Real-time FIR Filtering on a TI C5502 DSP Core Using BIOS

Real Time Digital Signal Processing - University of Nebraska

Daniel Shchur
62096122

Contents

List of Figures **2**

1 Introduction **3**

2 FIR Filters **3**
2.1 Low Pass Design 3
2.2 High Pass Design 4
2.3 Implementation 4

3 Buttons, LEDs, and LCD **5**
3.1 Functionality 5
3.2 Implementation 5

4 Efficiency **6**

5 Summary **6**

6 Appendix **6**
6.1 Youtube Video Demonstration 6
6.2 Main Code 6
6.3 Audio Processing Code 9
6.4 LED Code 12
6.5 LCD Code 13

List of Figures

1 Frequency response of the Low Pass FIR filter 3
2 Frequency response of the High Pass FIR filter 4

1 Introduction

A continuation of the previous project, Finite Impulse Response filters were investigated in their implementation for real time applications. As an improvement of the previous single filter design, multiple filters and modes of operation were employed and implemented with user control through switch buttons and LCD and LED output of state through real time. This was able to be performed through the use of DSP/BIOS Real Time Operating System (RTOS) which allows for the scheduling and management of threads on the DSP Core for multi tasking.

2 FIR Filters

Both filters were designed in MATLAB, utilizing an equiripple FIR design to give a high degree of attenuation on the desired frequency while leaving the other signals more or less in tact. They were designed with a lower coefficient count in mind which means that their attenuation degree is lower than would be ideal, but they still provide enough attenuation for this application.

2.1 Low Pass Design

The filter was designed to a length of 200 coefficients. A passband frequency of 800 Hz, a stopband frequency of 1200 Hz, and equal weighting on both bands was used. The filter's frequency response chart is shown in Figure 1. As can be seen from the plot, a stop band attenuation of more than 30 dB is achieved.

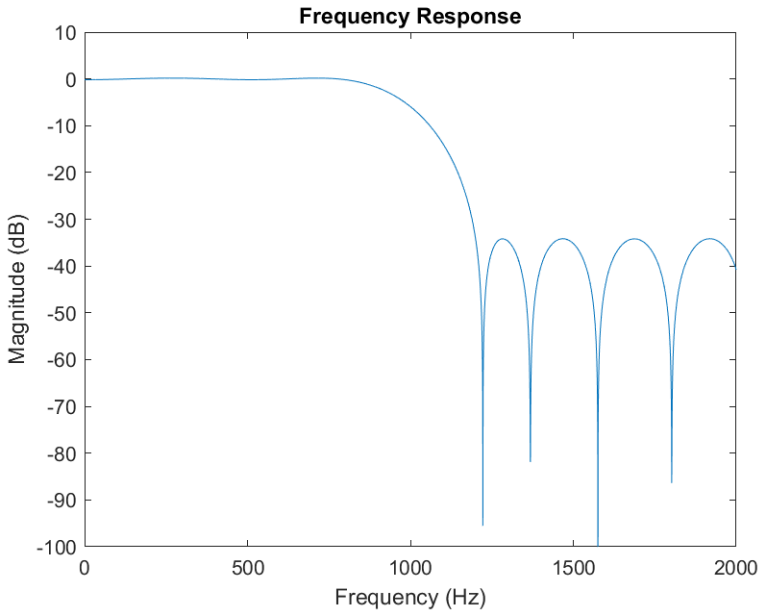

Figure 1: Frequency response of the Low Pass FIR filter

FIR filters have constant delay applied to the input signal based on the number of filter coefficients. This filter has a delay of 100 samples based on its order.

2.2 High Pass Design

The filter was designed to a length of 150 coefficients. A passband frequency of 2500 Hz, a stopband frequency of 2000 Hz, and equal weighting on both bands was used. The filter's frequency response chart is shown in Figure 2. As can be seen from the plot, a stop band attenuation of more than 30 dB is achieved.

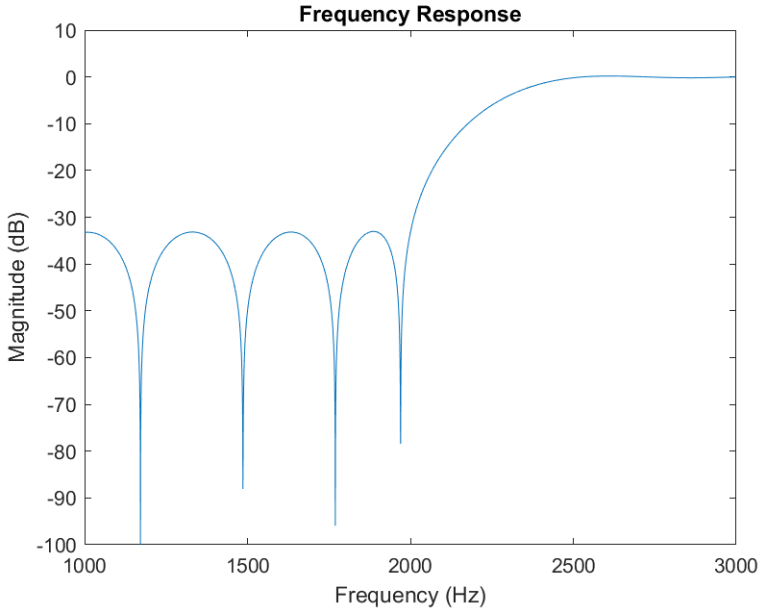

Figure 2: Frequency response of the High Pass FIR filter

This filter has a delay of 75 samples based on its order.

2.3 Implementation

The previous filtering function is employed to filter the incoming samples in real time. It is able to be used with minimal modification since it was implemented generalized enough to be able to be dropped in without issue. The only major change was making sure to avoid filtering if the number of coefficients are 0 so that when the filtering is disabled, it does not try to fill the delay line.

An audio processing section was created with receive and transmit functions to allow for capturing of incoming samples as well as filtering in real time before passing the filtered sample back to the output of the codec. These functions are implemented as hardware interrupt (HWI) threads to allow the BIOS scheduler to call on them when the audio codec generates hardware interrupts to let the system know when a sample is ready to be read or to be written to the output buffer. Utilizing threading in this way allows for alternate processing between samples, making use of the cycles wasted in the previous design.

While more computationally expensive, both channels of audio were filtered individually, allowing for stereo playback, as the filter complexity was low enough for the filtering to be performed in real time without distortions.

3 Buttons, LEDs, and LCD

3.1 Functionality

The two buttons, four LEDs, and the dot matrix display on the eZdsp board were both utilized within the design of this project to update the state of the system, allowing for user input to mute and to change filter type. The LCD and LEDs were used to give feed back to the user so that they can know what the current state of the system is.

3.2 Implementation

To make use of the unused cycles between audio samples and to allow the user to modify the state of the system in real time, both buttons were read in an idle (IDL) thread to perform muting and switching between filters. The LCD and LEDs were also updated accordingly with the state of the system when a button is pressed. The implementation simply polls the state for each button and calls on respective functions to update the LCD and LEDs, adjusting the state of the system. Because they are in an IDL thread, if the audio codec generates an interrupt, the IDL thread is put on hold while the respective function related to the interrupt is called and processed. This means that, as long as the HWI threads are implemented efficiently, the system will not drop samples and will achieve no distortion output while the IDL thread runs in the background, polling for button changes and updating the system between samples.

The writing of the LCD is implemented in such a way to make it easy to print text to the LCD. Columns of data have to be written to screen individually which means that a font face had to be created to be able to present the user information in a readable format. This was done with a switch statement which translates character input into a four column command to print to the screen a 4×8 formatted character. This was then put into a loop to allow for strings to be printed onto the screen with relative ease. One quirk with the system is that painting of pixels is performed from right to left, which means that the input of a string has to be reversed to print the characters in order; this is performed in the printing function so that strings can be formatted in normal English text order during design. This does incur some processing cost however, so if performance is key, the system could be reduced to a look up table format where all strings used in an implementation are preprocessed to be reversed before use which will save on processing cycles.

The turning on and off of LEDs is similarly implemented to ease the programmer in implementation, simply allowing for specification of what LED to turn turn on or off. This is done fairly efficiently and should incur minimal computation costs as only a single register needs to be updated to set the state of an LED. The LEDs simply cycle to show what filter is currently chosen and a single one is also dedicated to show that state of the mute.

To mute the system, a flag is simply set which in turn writes "0"s to the output of the audio codec. Filtering is still performed in the background, even though the system is muted. A better way to implement this would be to disable the HWI threads related to reading and writing of samples to prevent and processing of audio data if nothing is to be heard anyways.

To switch between filters, a type is passed into a function with a switch statement which updates a pointer to memory of filter coefficients, as well as a value which specifies the number of coefficients. This allows the filtering function to call on the same variables during filtering without knowing what filter is being passed in. This allows for easy extensibility if more filters were to be added in the future. When the filter type is set to "NONE", the coefficient

pointer is set to "NULL" and the coefficient size is updated to 0 to prevent the filtering from being performed. The filtering function therefore simply returns the input as is without any computation. This subtlety makes sure that cycles aren't wasted on filtering when there is no filter specified.

4 Efficiency

The filtering process had to be fairly efficient since the samples were not being buffered at all. The filtering was also being performed in an HWI thread which means the process had to run as fast as possible to allow other threads to run on the system in a timely manner. As such, the filter coefficients were kept low since both channels of audio were being filtered. O2 optimization was also utilized to compile the code to an efficient implementation. Using cycle analysis in Code Composer Studio, it was found that a single channel read without filtering would incur on average 96 cycles of computation. On the other hand, filtering took about 410 cycles on average. Given it takes about 3125 cycles for a single sample of audio to be read and written on the audio codec, only 13.12% of the CPU is utilized for a single channel of filtering or 26.24% for both, leaving plenty of headroom for the system to read the buttons and write to the LCD display and LEDs.

5 Summary

Threading through BIOS on a C5502 DSP core was explored in a real time application of filtering, updating LEDs, and writing to a display all at the same time. Since sample management and filtering was able to be only performed when needed through the use of hardware interrupts and HWI threads, other processing was able to happen in the background on IDL threads such as reading switches, updating LEDs, and writing text to the LCD in real time. No distortions were audible and response of the system input and output was perceivably instant due to the efficient implementation of the filtering process from the previous project, and the threading employed.

6 Appendix

6.1 Youtube Video Demonstration

https://youtu.be/LxuazLVjuxA

6.2 Main Code

```
6  #include <std.h>
7
8  #include <log.h>
9  #include <clk.h>
10 #include <tsk.h>
11 #include <gbl.h>
12
13 #include "bioscfg.h"
14 #include "c55.h"
```

```
15 #include "ezdsp5502.h"
16 #include "stdint.h"
17 #include "aic3204.h"
18 #include "ezdsp5502_mcbsp.h"
19 #include "ezdsp5502_i2cgpio.h"
20 #include "csl_mcbsp.h"
21 #include "graphics.h"
22 #include "led.h"
23
24 extern void audioProcessingInit(void);
25 extern void ToggleMute();
26 extern uint16_t GetMute();
27 extern void SetFilter(Uint8 type);
28
29 volatile int counter = 0;
30
31 Uint8 sw1State = 0;       // SW1 state
32 Uint8 sw2State = 0;       // SW2 state
33 Uint8 filterType = 0;
34
35 void main(void)
36
37 {
38     /* Initialize BSL */
39     EZDSP5502_init( );
40
41     /* Initialize LEDS */
42     InitLeds();
43     TurnOnLed(0);
44
45     /* Setup I2C GPIOs for Switches */
46     EZDSP5502_I2CGPIO_configLine(SW0, IN);
47     EZDSP5502_I2CGPIO_configLine(SW1, IN);
48
49     LOG_enable(&trace);
50
51     // configure the Codec chip
52     setup_aic3204();
53
54     /* Initialize I2S */
55     EZDSP5502_MCBSP_init();
56
57     screen_start();
58     select_screen(0);
59     screen_string("FILTER TYPE: NONE");
60     select_screen(1);
61     screen_string("MUTE: OFF ");
62
```

```
63      /* enable the interrupt with BIOS call */
64      C55_enableInt(7); // reference technical manual, I2S2 tx interrupt
65      C55_enableInt(6); // reference technical manual, I2S2 rx interrupt
66
67      audioProcessingInit();
68
69      // after main() exits the DSP/BIOS scheduler starts
70      LOG_printf(&trace, "Finished Main");
71  }
72
73
74  Void FXN_IDL_SWITCH(void)
75  {
76      if(!(EZDSP5502_I2CGPIO_readLine(SW0))) // Is SW1 pressed?
77         {
78              if(sw1State)          // Was previous state not pressed?
79              {
80                  filterType++;
81                  if(filterType >= 3)
82                      filterType = 0;
83                  SetFilter(filterType);     // Change filter
84                  select_screen(0);
85                  switch(filterType)
86                  {
87                      case 0:
88                          TurnOnLed(0);
89                          TurnOffLed(1);
90                          TurnOffLed(2);
91                          screen_string("NONE");
92                          break;
93                      case 1:
94                          TurnOffLed(0);
95                          TurnOnLed(1);
96                          TurnOffLed(2);
97                          screen_string("HPF ");
98                          break;
99                      case 2:
100                         TurnOffLed(0);
101                         TurnOffLed(1);
102                         TurnOnLed(2);
103                         screen_string("LPF ");
104                         break;
105                     default:
106                         TurnOnLed(0);
107                         TurnOffLed(1);
108                         TurnOffLed(2);
109                         screen_string("UNK ");
110                         break;
```

```
111                 }
112                 sw1State = 0;     // Set state to 0 to allow only single press
113             }
114         }
115         else                      // SW1 not pressed
116             sw1State = 1;         // Set state to 1 to allow timer change
117 
118         /* Check SW2 */
119         if(!(EZDSP5502_I2CGPIO_readLine(SW1))) // Is SW2 pressed?
120         {
121             if(sw2State)          // Was previous state not pressed?
122             {
123                 ToggleMute();     // Mute
124                 select_screen(1);
125                 switch(GetMute())
126                 {
127                     case 0:
128                         TurnOffLed(3);
129                         screen_string("OFF ");
130                         break;
131                     case 1:
132                         TurnOnLed(3);
133                         screen_string("ON  ");
134                         break;
135                     default:
136                         TurnOffLed(3);
137                         screen_string("UNK ");
138                         break;
139                 }
140 
141                 sw2State = 0;     // Set state to 0 to allow only single press
142             }
143         }
144         else                      // SW2 not pressed
145             sw2State = 1;         // Set state to 1 to allow tone change
146 }
```

6.3 Audio Processing Code

```
5  #include "bioscfg.h"
6  #include "ezdsp5502.h"
7  #include "stdint.h"
8  #include "aic3204.h"
9  #include "ezdsp5502_mcbsp.h"
10 #include "csl_mcbsp.h"
11 #include "string.h"
12 #include "myfir.h"
13 
```

```
14 extern MCBSP_Handle aicMcbsp;
15
16 extern int16_t lpfCoeff[];
17 extern int16_t hpfCoeff[];
18
19 int16_t* filter_coeff;
20 int16_t filter_length = 0;
21
22 int16_t rxRightSample = 0;
23 int16_t rxLeftSample = 0;
24 int16_t leftRightFlag = 0;
25 int16_t txleftRightFlag = 0;
26 uint16_t muteFlag = 0;
27 int16_t delayLineL[LPF_COEFF_LEN], delayLineR[LPF_COEFF_LEN];
28
29 void ToggleMute()
30 {
31     muteFlag = muteFlag ? 0 : 1;
32 }
33
34 uint16_t GetMute()
35 {
36     return muteFlag;
37 }
38
39 void audioProcessingInit(void)
40 {
41     memset(delayLineL, 0, LPF_COEFF_LEN);
42     memset(delayLineR, 0, LPF_COEFF_LEN);
43         rxRightSample = 0;
44         rxLeftSample = 0;
45
46 }
47
48 void SetFilter(Uint8 type)
49 {
50     switch(type)
51     {
52         case 0:
53             filter_coeff = NULL;
54             filter_length = 0;
55             break;
56         case 1:
57             filter_coeff = hpfCoeff;
58             filter_length = HPF_COEFF_LEN;
59             break;
60         case 2:
61             filter_coeff = lpfCoeff;
```

```
62              filter_length = LPF_COEFF_LEN;
63              break;
64          default:
65              filter_coeff = NULL;
66              filter_length = 0;
67              break;
68      }
69  }
70
71  void HWI_I2S_Rx(void)
72  {
73          if (leftRightFlag == 0)
74          {
75                  rxLeftSample = MCBSP_read16(aicMcbsp);
76                  myfir((const int16_t *) &rxLeftSample, filter_coeff,
                    ↪  &rxLeftSample, delayLineL, 1, filter_length);
77                  leftRightFlag = 1;
78          }
79          else
80          {
81                  rxRightSample = MCBSP_read16(aicMcbsp);
82                  myfir((const int16_t *) &rxRightSample, filter_coeff,
                    ↪  &rxRightSample, delayLineR, 1, filter_length);
83                  leftRightFlag = 0;
84          }
85  }
86
87  void HWI_I2S_Tx(void)
88  {
89      if (muteFlag)
90      {
91          rxLeftSample = 0;
92          rxRightSample = 0;
93      }
94          if (txleftRightFlag == 0)
95          {
96                  MCBSP_write16(aicMcbsp,rxLeftSample);
97                  txleftRightFlag = 1;
98          }
99          else
100         {
101                 MCBSP_write16(aicMcbsp,rxRightSample);
102                 txleftRightFlag = 0;
103         }
104 }
```

6.4 LED Code

```
8  #include "ezdsp5502_i2cgpio.h"
9  #include "led.h"
10
11 void InitLeds(void)
12 {
13     /* Setup I2C GPIO directions for LEDs */
14     EZDSP5502_I2CGPIO_configLine(LED0, OUT);
15     EZDSP5502_I2CGPIO_configLine(LED1, OUT);
16     EZDSP5502_I2CGPIO_configLine(LED2, OUT);
17     EZDSP5502_I2CGPIO_configLine(LED3, OUT);
18
19     /* Turn off all LEDs */
20     EZDSP5502_I2CGPIO_writeLine(LED0, HIGH);
21     EZDSP5502_I2CGPIO_writeLine(LED1, HIGH);
22     EZDSP5502_I2CGPIO_writeLine(LED2, HIGH);
23     EZDSP5502_I2CGPIO_writeLine(LED3, HIGH);
24 }
25
26 void TurnOnLed(Uint8 led)
27 {
28     switch(led)
29     {
30         case 0:
31             led = LED0;
32             break;
33         case 1:
34             led = LED1;
35             break;
36         case 2:
37             led = LED2;
38             break;
39         case 3:
40             led = LED3;
41             break;
42         default:
43             return;
44     }
45     EZDSP5502_I2CGPIO_writeLine(led, LOW);
46 }
47
48 void TurnOffLed(Uint8 led)
49 {
50     switch(led)
51     {
52         case 0:
53             led = LED0;
```

```
            break;
        case 1:
            led = LED1;
            break;
        case 2:
            led = LED2;
            break;
        case 3:
            led = LED3;
            break;
        default:
            return;
    }
    EZDSP5502_I2CGPIO_writeLine(led, HIGH);
}
```

6.5 LCD Code

```
#include "graphics.h"
#include "ezdsp5502.h"
#include "lcd.h"

void screen_start()
{

    /* Initialize  Display */
    osd9616_init( );

    osd9616_send(0x00,0x2e);     // Deactivate Scrolling

    clear_screen(0);
    clear_screen(1);

    return;
}

void select_screen(Uint8 screen)
{
    osd9616_send(0x00,0x00);   // Set low column address
    osd9616_send(0x00,0x10);   // Set high column address
    osd9616_send(0x00,0xb0+screen); // Set to page 0
}

void clear_screen(Uint8 screen)
{
    select_screen(screen);
    int i;
    for(i=0; i<128; i++) osd9616_send(0x40, 0x00);
```

```
}

uint16_t screen_char(char c)
{
    switch(c)
    {
        case 'A':
            printLetter(0x7C, 0x09, 0x0A, 0x7C);
            break;
        case 'B':
            printLetter(0x36, 0x49, 0x49, 0x7F);
            break;
        case 'C':
            printLetter(0x22, 0x41, 0x41, 0x3E);
            break;
        case 'D':
            printLetter(0x3E, 0x41, 0x41, 0x7F);
            break;
        case 'E':
            printLetter(0x41, 0x49, 0x49, 0x7F);
            break;
        case 'F':
            printLetter(0x01, 0x09, 0x09, 0x7F);
            break;
        case 'G':
        case 'H':
            printLetter(0x7F, 0x08, 0x08, 0x7F);
            break;
        case 'I':
            printLetter(0x00, 0x7F, 0x00, 0x00);
            break;
        case 'J':
        case 'K':
        case 'L':
            printLetter(0x40, 0x40, 0x40, 0x7F);
            break;
        case 'M':
            printLetter(0x7F, 0x06, 0x06, 0x7F);
            break;
        case 'N':
            printLetter(0x7F, 0x30, 0x0E, 0x7F);
            break;
        case 'O':
            printLetter(0x3E, 0x41, 0x41, 0x3E);
            break;
        case 'P':
            printLetter(0x06, 0x09, 0x09, 0x7F);
            break;
```

```
86          case 'Q':
87          case 'R':
88              printLetter(0x46, 0x29, 0x19, 0x7F);
89              break;
90          case 'S':
91              printLetter(0x32, 0x49, 0x49, 0x26);
92              break;
93          case 'T':
94              printLetter(0x01, 0x7F, 0x01, 0x01);
95              break;
96          case 'U':
97              printLetter(0x3F, 0x40, 0x40, 0x3F);
98              break;
99          case 'V':
100         case 'W':
101             printLetter(0x7F, 0x30, 0x30, 0x7F);
102             break;
103         case 'X':
104             printLetter(0x63, 0x1C, 0x1C, 0x63);
105             break;
106         case 'Y':
107             printLetter(0x07, 0x78, 0x08, 0x07);
108             break;
109         case 'Z':
110             printLetter(0x43, 0x4D, 0x51, 0x61);
111             break;
112         case ' ':
113             printLetter(0x00, 0x00, 0x00, 0x00);
114             break;
115         case '0':
116             printLetter(0x3E, 0x49, 0x45, 0x3E);
117             break;
118         case '1':
119             printLetter(0x40, 0x7F, 0x42, 0x00);
120             break;
121         case '2':
122             printLetter(0x47, 0x49, 0x51, 0x62);
123             break;
124         case '3':
125         case '4':
126         case '5':
127             printLetter(0x31, 0x49, 0x49, 0x2F);
128             break;
129         case '6':
130         case '7':
131         case '8':
132         case '9':
133         case '!':
```

```
134             printLetter(0x00, 0x00, 0x5F, 0x00);
135             break;
136         case '?':
137         case '.':
138             printLetter(0x00, 0x00, 0x40, 0x00);
139             break;
140         case ',':
141             printLetter(0x00, 0x20, 0x40, 0x00);
142             break;
143         case ':':
144             printLetter(0x00, 0x00, 0x14, 0x00);
145             break;
146         case '=':
147             printLetter(0x14, 0x14, 0x14, 0x14);
148             break;
149         default:
150             return 1;
151             break;
152     }
153     return 0;
154 }
155 
156 uint16_t screen_string(char* str)
157 {
158     int count = 0;
159     char* start = str - 1;
160     while (*str != '\0')
161     {
162         count++;
163         str++;
164     }
165     if(count > 19)
166     {
167         return 1;
168     }
169     while (str != start)
170     {
171         screen_char(*str);
172         str--;
173     }
174     return 0;
175 }
```