

BIOS Multi-thread FIR on a TI C5502 DSP Core

Real Time Digital Signal Processing - University of Nebraska

Daniel Shchur
62096122

Contents

1	Introduction	3
2	Tasks	3
2.1	Implementation	3
2.1.1	Audio Processing	3
2.1.2	User Interface	3
3	Queue Circular Buffer	4
3.1	Implementation	4
4	Summary	4
5	Appendix	4
5.1	Youtube Video Demonstration	4
5.2	Main Code	5
5.3	Audio Processing Code	7
5.4	Queue Code	11

1 Introduction

A continuation of the previous project, filtering and user interaction functionality were delegated to thread tasks on the BIOS API with cross thread communication facilitated through the use of mailbox queue objects. Previous functionality was kept and expected to stay the same with the ability to change filter modes and toggle mute through the use of buttons. 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 Tasks

Threads were initially implemented and introduced in the previous assignment with simple HWI threads for the reading and writing of the audio codec as well as an IDL thread to manage user interface button reads and screen/led writes. As an extension of this system to allow for better extensibility and higher performance, the filtering previously done in the the HWI thread of the sample read was pulled into an audio processing task function to filter in a blockable thread. User interface functionality was also moved to a task thread.

2.1 Implementation

2.1.1 Audio Processing

The audio processing was preformed in an audio processing task with a priority of 8, putting it at a middle priority, but higher than any other task thread. This was done to ensure that the thread was able to process the audio in a timely manner and prevent the output queue from ever starving the hardware transmit thread. The task was built in such a way that it would block on a receiving audio mailbox, waiting for 1 millisecond of audio from the hardware receiving thread. Once the audio was recieved, it would see if there were any command messages from the user interface task to control the state of the task such as the muting or the filter. The timeout was set to 0 to ensure that the task would not wait for new messages from the user interface before continuing to process the 1 millisecond of audio. As the task processes a sample, it pushes it onto the output queue one at a time to ensure the hardware transmit thread has samples to write to the codec. Once the audio was processed, it would again block on the audio mailbox, waiting for another millisecond of audio.

2.1.2 User Interface

The user interface task was lightly modified from the IDL version to work within a task. Its priority was set to 3 to be lower than the audio processing task, but higher than the IDL task since user input isn't as time critical as audio processing. The user interface was slightly modified to send commands through a mailbox instead of directly modifying the system state. This ensured that every millisecond of audio was processed in a consistent state without abrupt and non-atomic changes. A command structure was made to easily name different commands such as toggling mute or changing the filter type. Writing to the LEDs and display were still processed in the user interface task, taking care to post the respective command to the command mailbox before changing the LEDs and display.

3 Queue Circular Buffer

To pass processed audio from the audio processing task to the audio codec transmit interrupt thread, a circular buffer as a queue was implemented.

3.1 Implementation

The queue was implemented using a fixed sized array of the same type as the audio samples, a `sample_t struct` type with a `channel_t enum` for channel identification and `int16_t` for audio sample data storage. This incurred 2 16bit words of data per audio sample. The array was the same length as the output mailbox for the 1 millisecond audio codec read thread which read in 96 samples in total with 2 slots to give a buffer. Therefore, the array was 192 in length to accommodate all of the data if such a backup occurred. To manage indexing within the queue, 3 variables were used to keep track of the head, tail, and size of the current queue. When an element were to be added to the end of the queue, `q_push(sample)` would be called, incrementing the tail and adding the element to the new tail's location if there was still space left. When an element were to be removed from the queue, `sample = q_pop()` would be called which returned the element at the head and incremented it to the next element in the queue. The size variable would also increment and decrement with push and pop operations. This way, order of data was taken care of internally without need for complex structures such as linked lists or for implementing functions needing to take care of these variables. With the queue in a separate file and built generically with these calling functions, only minor changes would need to be made to implement a different type. It could also be extended to allow for heap memory allocation and usage, similar to the mailboxes.

To make sure data was pushed onto the queue in an atomic and threadsafe way, hardware interrupts were disabled during the push process and enabled after. This way, the receiving hardware thread wouldn't ever try to pop data off the queue when data was being pushed on.

4 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 read and writes 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 task threads such as audio filtering, reading switches, updating LEDs, and writing text to the LCD in real time. The use of mailboxes and queues allowed from cross-thread communication. 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.

5 Appendix

5.1 Youtube Video Demonstration

<https://youtu.be/WNJuxH9QvGQ>

5.2 Main Code

```
25 extern void audioProcessingInit(void);
26 extern uint16_t GetMute();
27
28 void main(void)
29 {
30     /* Initialize BSL */
31     EZDSP5502_init( );
32
33     /* Setup I2C GPIOs for Switches */
34     EZDSP5502_I2CGPIO_configLine(SW0, IN);
35     EZDSP5502_I2CGPIO_configLine(SW1, IN);
36
37     LOG_enable(&trace);
38
39     // configure the Codec chip
40     setup_aic3204();
41
42     /* Initialize I2S */
43     EZDSP5502_MCBSP_init();
44
45     /* enable the interrupt with BIOS call */
46     C55_enableInt(7); // reference technical manual, I2S2 tx interrupt
47     C55_enableInt(6); // reference technical manual, I2S2 rx interrupt
48
49     audioProcessingInit();
50
51     /* Initialize LEDS */
52     InitLeds();
53     TurnOnLed(0);
54     /* Initialize Screen */
55     screen_start();
56     select_screen(0);
57     screen_string("FILTER TYPE: TEST");
58     select_screen(1);
59     screen_string("MUTE: OFF ");
60
61     // after main() exits the DSP/BIOS scheduler starts
62     LOG_printf(&trace, "Finished Main");
63 }
64
65 void TSK_UserInterface(void)
66 {
67     Uint8 sw1State = 0;
68     Uint8 sw2State = 0;
69     Uint8 filterType = 1;
```

```
71     while(1)
72     {
73         /* Check SW1 */
74         if(!(EZDSP5502_I2CGPIO_readLine(SW0)))
75         {
76             if(sw1State)
77             {
78                 filterType++;
79                 if(filterType >= 4)
80                     filterType = 1;
81                 MBX_post(&mbx_command, &filterType, SYS_FOREVER);
82                 select_screen(0);
83                 switch(filterType)
84                 {
85                     case FILTER_NONE:
86                         TurnOnLed(0);
87                         TurnOffLed(1);
88                         TurnOffLed(2);
89                         screen_string("NONE");
90                         break;
91                     case FILTER_HPF:
92                         TurnOffLed(0);
93                         TurnOnLed(1);
94                         TurnOffLed(2);
95                         screen_string("HPF ");
96                         break;
97                     case FILTER_LPF:
98                         TurnOffLed(0);
99                         TurnOffLed(1);
100                        TurnOnLed(2);
101                        screen_string("LPF ");
102                        break;
103                    default:
104                        TurnOnLed(0);
105                        TurnOffLed(1);
106                        TurnOffLed(2);
107                        screen_string("UNK ");
108                        break;
109                }
110                sw1State = 0;
111            }
112        }
113        else
114            sw1State = 1;
115
116        /* Check SW2 */
117        if(!(EZDSP5502_I2CGPIO_readLine(SW1)))
118        {
```

```

119         if(sw2State)
120         {
121             sw2State = 0;
122             MBX_post(&mbx_command, &sw2State, SYS_FOREVER);
123             select_screen(1);
124             switch(GetMute())
125             {
126                 case 0:
127                     TurnOffLed(3);
128                     screen_string("OFF ");
129                     break;
130                 case 1:
131                     TurnOnLed(3);
132                     screen_string("ON ");
133                     break;
134                 default:
135                     TurnOffLed(3);
136                     screen_string("UNK ");
137                     break;
138             }
139         }
140     }
141     else
142         sw2State = 1;
143 }
144 }

```

5.3 Audio Processing Code

```

28 extern MCBSP_Handle aicMcbasp;
29
30 // Pulled from myfir
31 extern int16_t lpfCoeff[];
32 extern int16_t hpfCoeff[];
33
34 static int16_t* filter_coeff;
35 static int16_t filter_length = 0;
36 static int16_t leftRightFlag = 0;
37 static uint16_t muteFlag = 0;
38 static int16_t delayLineL[LPF_COEFF_LEN], delayLineR[LPF_COEFF_LEN];
39
40 /*
41  * Toggles the mute flag
42  */
43 void ToggleMute()
44 {
45     muteFlag = muteFlag ? 0 : 1;
46 }

```

```
47
48  /*
49  * Returns the mute flag state
50  */
51  uint16_t GetMute()
52  {
53      return muteFlag;
54  }
55
56  /*
57  * Initializes the audio processing task variables
58  */
59  void audioProcessingInit(void)
60  {
61      memset(delayLineL, 0, LPF_COEFF_LEN);
62      memset(delayLineR, 0, LPF_COEFF_LEN);
63  }
64
65  /*
66  * Set's the currently used filter pointers
67  */
68  void SetFilter(command_t type)
69  {
70      switch(type)
71      {
72          case FILTER_NONE:
73              filter_coeff = NULL;
74              filter_length = 0;
75              break;
76          case FILTER_HPF:
77              filter_coeff = hpfCoeff;
78              filter_length = HPF_COEFF_LEN;
79              break;
80          case FILTER_LPF:
81              filter_coeff = lpfCoeff;
82              filter_length = LPF_COEFF_LEN;
83              break;
84          default:
85              filter_coeff = NULL;
86              filter_length = 0;
87              break;
88      }
89  }
90
91  /*
92  * BIOS Task to process incoming audio data
93  * Receives 1 msec of audio from the receiver thread
94  * to then filter based on the system state.
```

```

95  *
96  * System state is modified by the command mailbox
97  * which is posted by the userInterface task.
98  */
99  void TSK_AudioProcessing()
100 {
101     sample_t processedSample = {LEFT, 0};
102     command_t state = 0;
103     uint16_t i = 0;
104     Uns old;
105     sample_t samples[MBX_AUDIO_LEN];
106     while(1)
107     {
108         // wait for data
109         MBX_pend(&mbx_audio, samples, SYS_FOREVER);
110         // Pull a command off of the buffer, only process if exists
111         if(MBX_pend(&mbx_command, &state, 0))
112         {
113             if(state == TOGGLE_MUTE)
114                 ToggleMute();
115             else
116                 SetFilter(state);
117         }
118         else
119             LOG_printf(&trace, "HWI_I2S_Rx: Failed to post to 'mbx_audio'");
120         for(i = 0; i < MBX_AUDIO_LEN; i++)
121         {
122             if (muteFlag)
123             {
124                 processedSample.data = 0;
125             }
126             else
127             {
128                 myfir((const int16_t *) &samples[i].data,
129                     filter_coeff,
130                     &processedSample.data,
131                     samples[i].channel ? delayLineL : delayLineR,
132                     1,
133                     filter_length
134                 );
135             }
136             processedSample.channel = samples[i].channel;
137             old = HWI_disable();
138             q_push(processedSample);
139             HWI_restore(old);
140         }
141     }
142 }

```

```
143
144 /**
145  * BIOS HWI Thread which triggers when audio codec
146  * has a new sample on its registers to read.
147  *
148  * Reads in 1msec of audio before posting it to a mailbox
149  */
150 void HWI_I2S_Rx(void)
151 {
152     static sample_t samples[MBX_AUDIO_LEN];
153     static uint16_t idx = 0;
154     if (leftRightFlag == 0)
155     {
156         samples[idx].data = MCBSP_read16(aicMcbasp);
157         samples[idx].channel = LEFT;
158         leftRightFlag = 1;
159         idx++;
160     }
161     else
162     {
163         samples[idx].data = MCBSP_read16(aicMcbasp);
164         samples[idx].channel = RIGHT;
165         leftRightFlag = 0;
166         idx++;
167     }
168     if(idx >= MBX_AUDIO_LEN)
169     {
170         if(!MBX_post(&mbx_audio, samples, 0))
171             LOG_printf(&trace, "HWI_I2S_Rx: Failed to post to 'mbx_audio'");
172         idx = 0;
173     }
174 }
175
176
177 /**
178  * BIOS HWI Thread which triggers when audio codec
179  * is ready to read a new sample.
180  *
181  * Pulls a single sample at a time from a queue
182  */
183 void HWI_I2S_Tx(void)
184 {
185     static sample_t sample = {LEFT, 0};
186     sample = q_pop();
187     MCBSP_write16(aicMcbasp, sample.data);
188 }
```

5.4 Queue Code

```
11 static sample_t sampleQueue[Q_MAX_SIZE] = {LEFT, -1};
12 static int16_t head = 0;
13 static int16_t tail = 0;
14 static int16_t size = 0;
15
16 sample_t q_pop()
17 {
18     sample_t popped = {LEFT, 0};
19     if (size == 0)
20         return popped;
21     popped = sampleQueue[head];
22     head++;
23     if (head >= Q_MAX_SIZE)
24         head = 0;
25     size--;
26     return popped;
27 }
28
29 bool q_push(sample_t sample)
30 {
31     if (size == Q_MAX_SIZE)
32         return 0;
33     tail++;
34     if (tail >= Q_MAX_SIZE)
35         tail = 0;
36     sampleQueue[tail] = sample;
37     size++;
38     return 1;
39 }
40
41 void q_empty()
42 {
43     head = 0;
44     tail = 0;
45     size = 0;
46 }
47
48 bool q_is_empty()
49 {
50     return size == 0;
51 }
52
53 bool q_is_full()
54 {
55     return size == Q_MAX_SIZE;
56 }
```