Lab Assignment 4
Audio Player

Luke Jaffe
jaffe.lu@husky.neu.edu

Brian Crafton
crafton.b@husky.neu.edu

Submit Date: 03/22/2016
Due Date: 03/22/2016

Instructor: Prof. Gunar Schirner
TA: Jinghan Zhang

Abstract

The purpose of Lab 4 was to understand, explore, and solve issues in streaming data on embedded systems. Most embedded systems face real time requirements like streaming data, so being able to do this efficiently and effectively is important. This application requires deterministic behavior, low power consumption, and the ability to run concurrently with other applications.

Introduction

In previous labs we have used techniques like reading and writing to peripherals, interrupts, and separate tasks. In this lab all of these techniques are used to perform the task of playing audio efficiently and effectively. In the first part of the lab the goal is to simply play audio and this was achieved by writing to the FIFO when there was space available. When there was no space available a task delay was used. Because the task delay was used it does allow the processor to do other things while the FIFO empties. This is important because otherwise the processor would waste cycles constantly checking whether the FIFO has space available or not. In the next section of the lab volume control was introduced. This was implemented by setting up an interrupt for when the button was pressed and linking it to a function that incremented or decremented the volume based on the button that was pressed.

In the second half of the lab chunked data was used rather than streaming samples one by one. This was implemented in a very similar fashion to how it was implemented the previous section, except that rather than check if there was any vacant space the idea was to check if there was enough vacant space for a chunk. In the next section of the lab, interrupts and a queue were used to help make the streaming data more efficient on the processor and also more deterministic. An interrupt was set to occur when the FIFO became empty. For this to happen, initially a chunk needs to be written to the FIFO so that an interrupt will occur. In addition to this, a queue containing more chunks needs to be filled so that when the ISR is called, it has chunks it can write to the FIFO. The audioTx_put function was implemented to write to the FIFO when the FIFO was empty and when it was not empty, to write to the queue. This way the ISR could write the data in queue when it was called.

In the last part of the lab both error handling and an example of what happens when data streaming is cut off by other applications was explored. By only running the audioTx_put every other 10 second period. Because it is writing by chunk the output stream simply picks up where it left off so the sound does stream sequentially, but with 10 second periods of now sound. So although the sound task only played every other 10 second period, it handled it correctly by keeping track of what chunk it left off at.

Lab Materials

- Hardware
  - Xilinx Zynq®-7000 All Programmable SoC
- Software
  - FreeRTOS
- Environment
  - Eclipse IDE
  - Xilinx SDK

Results and Analysis

I. Basic Audio
audioPlayer project

In this portion of the lab, we used the Zynq to play audio using the ADAU1761 audio codec. The audio samples were provided in the form of a large static array. The audio codec uses the I2C interface for configuration. The audio player task (audioPlayer_task) consists of initializing the audio player struct, and then looping to play the sample array repeatedly:

```
audioPlayer_init(pThis);
while (1)
      txData(snd_samples, snd_samples_nSamples);
```

The txData function loops over the length of the packet, and waits to push samples in the FIFO if there isn't enough room:

```
int i = 0;
while (i < len)
{
      if (*(volatile u32 *) (FIFO_BASE_ADDR + FIFO_TX_VAC))
      {
            u32 sample_32 = (sampleA[i] << 16) & 0xFFFF0000;
            *(volatile u32 *) (FIFO_BASE_ADDR + FIFO_TX_DATA) = sample_32;
            *(volatile u32 *) (FIFO_BASE_ADDR + FIFO_TX_LENGTH) = 1;
            *(volatile u32 *) (FIFO_BASE_ADDR + FIFO_TX_DATA) = sample_32;
            *(volatile u32 *) (FIFO_BASE_ADDR + FIFO_TX_LENGTH) = 1;
            i++;
      }
      else
      {
            vTaskDelay(1); //delay for 0-10 ms
      }
}
```

We can hear Gunar giving congratulations when the audio sample plays correctly. When the check for FIFO vacancy is commented out, we hear only a brief "blip" for the audio, instead of the correct message.

II. Audio Interrupts
audioInterrupt project

In this portion of the lab, we will control the volume of the audio using push button interrupts. First, we must make sure the ISR has access to the audio player struct. We could make it global, but we can do a better implementation by passing the audio player struct pointer as a void* to the ISR in the setup:

```
XScuGic_Connect(pGIC, GPIO_INTERRUPT_ID,
      (Xil_ExceptionHandler) gpio_intrHandler,(void *) pThis); //
audioPlayer_t *pThis
```

In the interrupt handler, we check which button was pressed, and increase or decrease the volume accordingly:

```
// read interrupt status
u32 int_assert = (*(volatile u32 *)GPIO_INT_STAT_2) & ~(*(volatile u32
*)GPIO_INT_MASK_2);

// clear interrupts
(*(volatile u32 *)GPIO_INT_STAT_2) = int_assert;

/* check which button was pressed */
if (int_assert & (0x01<<20))
{
      audioPlayer_volumeIncrease((audioPlayer_t *)pRef);
}
if (int_assert & (0x01<<18))
{
      audioPlayer_volumeDecrease((audioPlayer_t *)pRef);
}
```

The increase and decrease volume functions use the volume member variable of the audio player struct. Then, they call the codec setVolume function. We increment and decrement the volume by 4 because this is the lowest delta which does not cause interruptions in the audio. Other groups noticed this effect as well.

```
void audioPlayer_volumeIncrease(audioPlayer_t *pThis)
{
      /* insert code for volume increase here */
      /* for this first, implement adau1761_setVolume in adau1761.c */
      if (pThis->volume < VOLUME_MAX)
            pThis->volume += 4;
      adau1761_setVolume(&(pThis->codec), pThis->volume);
}

void audioPlayer_volumeDecrease(audioPlayer_t *pThis)
{
      /* insert code for volume decrease here */
      if (pThis->volume > 0)
            pThis->volume -= 4;
      adau1761_setVolume(&(pThis->codec), pThis->volume);
}
```

The codec setVolume function simply sets the left and right volume registers with the updated volume value:

```
void adau1761_setVolume(tAdau1761 *pThis, unsigned int newVol)
{
      adau1761_regWrite(pThis,
R31_PLAYBACK_LINE_OUTPUT_LEFT_VOLUME_CONTROL, newVol);
      adau1761_regWrite(pThis,
R32_PLAYBACK_LINE_OUTPUT_RIGHT_VOLUME_CONTROL, newVol);
}
```

Next, we added the following code to the ISR:

```
{
      unsigned int n;
      for ( n=0; n < 5000000; n++ )
      {
            asm("nop; ssync;");
      }
}
```

Doing so, we noticed an audible delay during interrupts. This makes logical sense, since the audio playing task cannot preempt the interrupt; the interrupt must finish executing all the nop instructions before the audio player task gets a chance to run again. This manifests as an audible delay.

III. Audio Chunks
audioChunk project

In Part B of the lab, we read audio from chunks populated by a buffer pool. The initial task is to stream these chunks to the TX FIFO manually. This is done in a very similar manner to the previous section. The two key additions are that we must iterate over the number of bytesUsed/4, since there are 4 bytes per sample, and that we must release the chunk to the buffer pool after we finish pushing its bytes to the TX FIFO:

```
int i = 0;
while (i < pChunk->bytesUsed/4) // iterated over number of bytes used/4
{
      if (*(volatile u32 *) (FIFO_BASE_ADDR + FIFO_TX_VAC))
      {
            u32 sample_32 = (pChunk->u32_buff[i] << 16) & 0xFFFF0000;
            *(volatile u32 *) (FIFO_BASE_ADDR + FIFO_TX_DATA) = sample_32;
            *(volatile u32 *) (FIFO_BASE_ADDR + FIFO_TX_LENGTH) = 1;
            i++;
      }
}
bufferPool_d_release( pThis->pBuffP, pChunk ); // release the chunk to the
buffer pool
```

To see how long it takes to loop through 15 chunks, we can add the following code to the beginning of the audioTx_put function:

```
static int n = 0;
static int ticks = 0;

if (n == 0)
{
      ticks = xTaskGetTickCount();
}
      else if (n == 15)
{
      int ticks_elapsed = xTaskGetTickCount() - ticks;
      int time = ticks_elapsed;
      printf("Time passed (ticks): %d\n", time);
}
n++;
```

We found that 15 chunks executes very quickly, in 1 tick or less (10 ms or less). Since this is the smallest granularity available on the platform (we tried to find smaller and couldn't), we cannot truly say how long it takes to play 15 audio chunks.

IV. Audio ISR
audioInterruptStream project

The audioTx_start function is already set up to configure the ISR and trigger IRQs on FIFO near-empty. In the audioTx_put, function, we now have a conditional to check if the automatic ISR playing mechanism has started. If the mechanism has not started, we manually place the chunk in the FIFO. Else, we push a pointer to the chunk into the queue, to be played in the ISR.

```
if (!pThis->running)
{

      int i = 0;
      while (i < pChunk->bytesUsed/4)
      {
            if (*(volatile u32 *) (FIFO_BASE_ADDR + FIFO_TX_VAC))
            {
                  u32 sample_32 = (pChunk->u32_buff[i] << 16) & 0xFFFF0000;
                  *(volatile u32 *) (FIFO_BASE_ADDR + FIFO_TX_DATA) =
sample_32;
                  *(volatile u32 *) (FIFO_BASE_ADDR + FIFO_TX_LENGTH) = 1;
                  *(volatile u32 *) (FIFO_BASE_ADDR + FIFO_TX_DATA) =
sample_32;
                  *(volatile u32 *) (FIFO_BASE_ADDR + FIFO_TX_LENGTH) = 1;
```

```
                i++;
            }
        }
        bufferPool_d_release( pThis->pBuffP, pChunk );
        pThis->running = 1;
}
else
{
        xQueueSend(pThis->queue, &pChunk, 0);
}
```

In the ISR, we first check and clear interrupts:

```
unsigned int flags = (*(volatile u32 *)FIFO_BASE_ADDR); // read the status
register
(*(volatile u32 *)FIFO_BASE_ADDR) = flags;
```

Then we play the sample in the same way as in the put function. Again, we release the chunk to the buffer pool when finished.

```
/* copy samples in chuck into FIFO */
int i = 0;
while (i < pChunk->bytesUsed/4)
{
        if (*(volatile u32 *) (FIFO_BASE_ADDR + FIFO_TX_VAC))
        {
                u32 sample_32 = (pChunk->u32_buff[i] << 16) & 0xFFFF0000;
                *(volatile u32 *) (FIFO_BASE_ADDR + FIFO_TX_DATA) = sample_32;
                *(volatile u32 *) (FIFO_BASE_ADDR + FIFO_TX_LENGTH) = 1;
                i++;
        }
}
bufferPool_d_release_from_ISR( pThis->pBuffP, pChunk );
```

Again, it takes less than 1 tick to loop through 15 audio chunks, so we cannot be certain of the true time for this few chunks.

V. Audio Delay

audio delay project

In the final part of the lab, we implement error handling to handle when the sound stops. We check if the queue is empty before proceeding. If it is, we set the running flag to false, to indicate that a new chunk must be manually placed in the TX FIFO to restart the transfer mechanism.

```
if(xQueueIsQueueEmptyFromISR(pThis->queue))
{
      pThis->running = 0;
      return;
}
```

To test this functionality, we alternate 1 second delay with 1 second audio in the main loop:

```
int count;
while(1)
{
      for (count = xTaskGetTickCount(); xTaskGetTickCount() < count+100;)
      {
            /** get audio chunk */
            status = audioRx_get(&pThis->rx, &pThis->chunk);

            /** If we have chunks that can be played then we provide them
             * to the audio TX */
            if ( 1 == status ) {
              /** play audio chunk through speakers */
              audioTx_put(&pThis->tx, pThis->chunk);
            }
      }
      vTaskDelay(100);
}
```

This code worked correctly: the audio delayed and restarted during each iteration of the loop.

To calculate how long it will take to drain the filled queue and the filled FIFO, we need the following figures:
Sampling rate = 44.1k samples/second
Chunk size = 512 bytes = 128 samples
Queue size = 30 chunks
FIFO contains 1 chunk = 128 samples

Total samples = 30*128 + 1*128 = 3968 samples
Total time to transmit = 3,968 samples / 44,100 samples/second = 89.977 ms

So it takes about 90 ms for this scenario to complete.

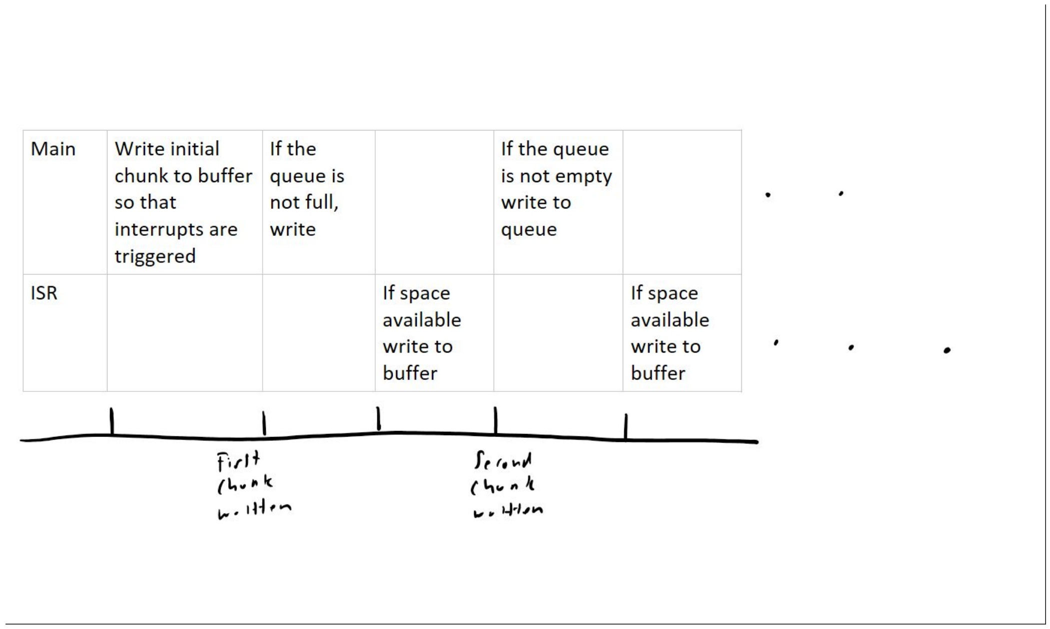
Conclusion

In this lab the goal was to investigate the problems and solutions to streaming data in real time efficiently and effectively. In order to do this effectively, it is necessary to consider how to limit the amount of time the processor is doing unproductive instructions. In the first part of the lab this was accomplished by using a task delay so that the processor would not spend all its time checking whether there was free space in the FIFO. This is effective because it allows the processor to do something else for 10 milliseconds while the FIFO works on pushing data out. After this a strategy was used where the data was written to the FIFO in chunks. This is a good choice because rather when the processor switches tasks it is more efficient to write large chunks at a time rather than writing a small number of samples. Switching tasks is expensive and it is best to write as many samples as possible at once. In the last improvement of the audio player a queue and interrupt was used. This was effective because rather than polling the status register for a signal to write data to the FIFO, an interrupt was triggered when the FIFO was empty. This gave the processor even more freedom since it would not have to check back when after each task delay, but only when it was interrupted. In order for this implementation to work, it was necessary that a queue be implemented so that the when the interrupt was called it would not write one chunk at a time, but however many chunks were able to fit in the queue. It was clear after completing the lab that although the first implementation of the sound player was effective, it was much less efficient than the last version. The last version used concepts that had been covered in previous labs and allowed for much more efficient use of the processor.

As well as improvement of a basic sound player, this lab covered error handling. It was necessary for the application that if an error occurred, the sound player should pick up playing where it left off. This case was tested by only calling the audioTx_put function every other period of 10 seconds. Because the index of the last chunk was maintained it was easy for the sound to play exactly where it had left off. This section of the lab displayed the important of error handling and what will happen if higher priority task was to interrupt the audio player task.

Post Lab Questions:

1. The audio could have been stored in ROM as well. There might need to be some sort of pre-fetching setup to deal with the long latency to ROM, but otherwise it would be better of stored in ROM because there is less RAM. If the sound data was stored in ROM, then the data would need to be loaded to a buffer in RAM prior to being used. This would allow for less RAM to be used. After it is in RAM the sound is written to the FIFO directly by the processor 1 32 bit word (1 sample) at a time. Once it is in the FIFO, the audio codec pulls them out at the frequency the sound is being played.
2. The reason the audio data is read in individual chunks is because the audio data is larger than the memory access bandwidth.
3. If the size of the chunk was increased, then the number of interrupts would be reduced. If it was decreased the number of interrupts would increase.The memory consumption would remain the same regardless of the size of the chunk, there would be more or less memory fetches per interrupt but the amount of memory used to store the data would remain the same.
4.



Main	Write initial chunk to buffer so that interrupts are triggered	If the queue is not full, write		If the queue is not empty write to queue	
ISR			If space available write to buffer		If space available write to buffer

5. The queue is not overflowed because if the queue is full and audioTx_put is called the queue add function will block so until a chunk from the queue is removed. Blocking when the queue is full prevents chunks from being dropped if the queue had been full.

References

[1] Xilinx, Zynq-7000 All Programmable SoC Software Developers Guide, v12.0 09/30/2015
http://www.xilinx.com/support/documentation/user_guides/ug821-zynq-7000-swdev.pdf

[2] Xilinx, Zynq-7000 All Programmable SoC Technical Reference Manual, v1.10, 02/23/2015
http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf