Laboratory Exercise 8

Audio and an Introduction to Multithreaded Applications

The purpose of this exercise is to create user-level Linux programs that produce audio output on the DE1-SoC board. The final application task for this exercise is to implement a digital piano that can play musical chords through the audio port of the board. In addition to listening to the music, you will also be able to visualize the sound as waveforms on a VGA display, and record/playback songs. As part of this exercise, you will also be introduced to writing *multithreaded* applications.

Using the Audio Port of the DE1-SoC Computer

The DE1-SoC Computer includes an audio port that is connected to the audio CODEC (COder/DECoder) chip on the DE1-SoC board. The default setting for the sample rate of the audio CODEC is 48,000 samples/second. The audio port provides audio-output capability via the line-out jack (3.5 mm green connector), as well as audio input via the microphone jack (pink connector). For this exercise you will use only the audio output feature. The audio port includes two *Write* FIFOs that are used to hold outgoing data, as well as two *Read* FIFOs (not used here) for incoming data. These FIFOs have a maximum depth of 128 32-bit words.

The audio port's programming interface consists of four 32-bit registers, as illustrated in Figure 1. The *Control* register, which has the address 0xFF203040, is readable to provide status information and writable to make control settings. Bits *RE* and *WE* are available for enabling the generation of processor interrupts from the audio port; you should set these bits to 0, as we are not using interrupts in this exercise. The bit *CW* can be set to 1 to clear the *Write* FIFO. The clear function remains active until *CW* is set back to 0. The read-only *Fifospace* register contains four 8-bit fields. The fields *WSRC* and *WSLC* give the number of words currently available for storing data in the right and left audio-out FIFOs. When the FIFOs are empty, the values are *WSRC* = *WSLC* = 128. The *Leftdata* and *Rightdata* registers are writable for audio out. When data is written into these registers it is loaded into the *Write* FIFOs.

Address	31 ··· 24	23 ··· 16	15 ··· 10	9	8	7 ··· 3	2	1	0	
0xFF203040	Unused			WI	RI		CW	CR	WE	RE	Control
0xFF203044	WSLC	WSRC	RALC			RARC					Fifospace
0xFF203048	Left data										Leftdata
0xFF20303C	Right data										Rightdata

Figure 1: Audio port registers.

Part I

You are to write a program that plays the tones of the middle C chromatic scale through the audio-out port of the DE1-SoC Computer system. The tones and their respective frequencies are shown in Table 1. When executed, your program should play these tones in sequence from the low C (261.63 Hz) to the higher C (523.25 Hz), then terminate. The tones should be outputted as sinusoidal waves, with each tone lasting 300 ms.

Note	Frequency (Hz)
C	261.626
C#/Db	277.183
D	293.665
D#/Eb	311.127
E	329.628
F	349.228
F#/Gb	369.994
G	391.995
G#/Ab	415.305
A	440.000
A#/Bb	466.164
B	493.883
C	523.251

Table 1: Notes of the middle C chromatic scale and their frequencies.

Perform the following:

1. Make a new folder to hold your C source code. Write a file *part1.c* for your `main` function, and any other source-code files that you require. To communicate with the audio port use the memory-mapped I/O addresses given in Figure 1. You will need to *map* the physical addresses of the audio port into virtual addresses by using Linux kernel functions like *mmap* and the */dev/mem* interface. These concepts are discussed in detail in the tutorial *Using Linux on the DE1-SoC*. To initialize the audio port you first should clear the *Write* FIFO by using the *CW* bit in the *Control* register.

   To output a sinusoidal wave to the audio port, use the $\sin(x)$ function provided by the library `<math.h>`. The code shown in Figure 2 demonstrates its use to output a 261.63 Hz tone (the middle C). To compile C code that uses the `<math.h>` library, you must append `-lm` to the end of your `gcc` command. The `-lm` flag instructs the compiler to include the math library as part of your program. Since the *sin* function uses radians to measure degrees, you will need to calculate the number of radians per sample for each tone, and then use a loop to output 300 ms of samples for each tone.

2. Connect headphones or a speaker to the audio-out jack (3.5 mm green connector) on the DE1-SoC board. Compile your program using a command such as `gcc -o part1 part1.c -lm`, and then run the program to hear the tones.

```
1   #include <math.h>
2
3   #define 2PI 6.28318531
4   #define SAMPLE_RATE 48000
5   ...
6   int main(void){
7       int nth_sample;
8       double freq = 261.626;
9       // Max volume when multiplied by sin() which ranges from -1 to 1
10      int vol = 0x7FFFFFFF;
11
12      // Write 48000 samples (one second's worth) of middle C (261.63~Hz)
13      for (nth_sample = 0; nth_sample < ...; nth_sample++){
14          write_to_audio_port(vol * sin(nth_sample * <some math goes here>));
15      }
16  }
```

Figure 2: Using the sin function to make samples for middle C (261.63 Hz).

Part II

In Part I you wrote a program that is capable of playing a single tone at a time. Now you will write a program that is capable of playing multiple tones simultaneously (chords). Your program should accept a single command-line input string of 13 '1's and '0's, which turn on and off the 13 tones of the chromatic scale. For example, running your program with `./part2 1000100100000` should play the notes C (261.63 Hz), E, and G. Your program should play the selected tones simultaneously for one second, and then terminate.

To output multiple tones simultaneously, you will have to generate samples that are a sum of multiple sinusoids. When doing so, you must be careful of possible overflows; the sum of multiple sinusoid samples should not overflow a signed 32-bit value. One solution is to set each tone's volume to be the maximum volume divided by 13, which ensures that the maximum possible value (when all 13 tones are being played) does not exceed the limit. Perform the following:

1. Make a new folder to hold your C source code. Augment your code from Part I to create a new file *part2.c*, in which the main program accepts one 13-character *string* argument. Create an integer array such as `int tone_volumes[13]` to store the volume for each tone, based the command-line argument of your program. Use a loop to create your sinusoidal waveform in which each sample is the sum of individual tones, for a duration of one second.
2. Compile your program using a command such as `gcc -o part2 part2.c -lm`, and then run the program to verify that each tone, and chords, are played properly.

Part III

In this part you will extend your code from Part II to create a simple digital piano. You will make use of the concept of *threads* to create a *multithreaded* program, as discussed below.

Introduction to Multithreaded Applications

A *thread* refers to an independent sequence of instructions executed by a processor. An application program can run as a single thread, or as multiple threads. When there are multiple threads running concurrently, it is the job of the operating system to schedule the threads for execution. In this way the CPU core(s) of your computer can be shared among the threads of your program(s).

When running on a computer with multiple *CPU cores*, breaking up a program into multiple threads allows parts of the program to run in parallel, on the multiple CPU cores. The possible benefits of multithreading include higher throughput and lower latency. In this part of the exercise we will explore how to use C-code to write a multithreaded program. This implementation will help to ensure that a CPU core is always available to fill the *Write* FIFOs of the audio port, so that there are no "empty spaces" in the sounds being played, and at the same time another CPU core is available to process user inputs.

The digital piano will have two continuous tasks. The first is to listen for user input and set the `tone_volume` array based on which tones are to be played. The second is to constantly write samples to the audio port. While it would be possible to have a single thread that does both tasks by interleaving them, doing so would have drawbacks. First, it could be detrimental to your program's responsiveness to user inputs as your program must write to the audio port in between reads of the keyboard. Likewise, since your program would have to read the keyboard in between writes to the audio port, the program may sometimes fail to supply the port with a sample every 1/48000 seconds. Having two threads solves these issues by doing both tasks simultaneously on the dual-core ARM processor of the DE1-SoC board.

Your program should have two threads:

- The **main thread**, which is responsible for listening to user input.
- The **audio thread**, which is responsible for writing samples to the audio port.

The main thread is created when your program is first executed (starting from `main(...)`). To provide piano "keys" you will connect a USB keyboard to the DE1-SoC board, following the mappings shown in Figure 3. The keyboard can be plugged into any USB port on the DE1-SoC board, and will be automatically recognized by the Linux system. Linux provides access to the keyboard through a device file located in */dev/input/*. For example, a keyboard plugged into the board might be represented by the device file */dev/input/by-id/generic-brand-USKeyboard-event-kbd*. Figure 4 shows C-language code that opens a keyboard device file and listens for user key strokes.

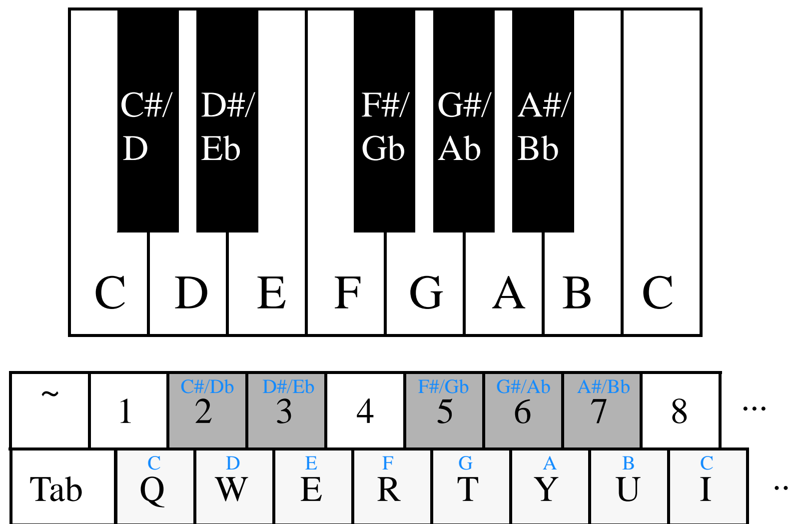

Figure 3: The mapping from piano keys to keyboard keys.

The audio thread can be created from within the main thread by using the *Pthreads* application programming interface (API), which is included as part of Linux. To use this API in your code, you must include the `<pthread.h>` header file and compile your code with the flag `-pthread` appended to your `gcc` command. Skeleton code for using this API is shown in Figure 5. In this code, the main program loops to read the USB keyboard, representing the piano keys (tones). When a key is pressed, the main program adjusts the volume of the corresponding tone in the `tone_volume` array. The audio thread loops to read the `tone_volume` array, and creates a sinusoid for each tone that is currently being played.

The audio thread is similar to the code from Part II, because it has to create, and possibly sum together, sinusoids corresponding to each tone. You may want to structure your audio thread so that it, in addition to generating the tone-sounds, slowly reduces the volume of each tone depending on whether or not the key for this tone is still being pressed. This approach simulates the way in which actual piano keys behave.

```
1  #include <stdio.h>
2  #include <fcntl.h>
3  #include <linux/input.h>
4
5  #define KEY_RELEASED 0
6  #define KEY_PRESSED 1
7
8  int main (int argc, char *argv[])
9  {
10     struct input_event ev;
11     int fd, event_size = sizeof (struct input_event);
12
13     // Get the keyboard device
14     if (argv[1] == NULL){
15         printf("Specify the path to the keyboard device ex.
               /dev/input/by-id/HP-KEYBOARD");
16         return -1;
17     }
18
19     // Open keyboard device
20     if ((fd = open (argv[1], O_RDONLY | O_NONBLOCK)) == -1){
21         printf ("Could not open %s\n", argv[1]);
22         return -1;
23     }
24
25     while (1){
26         // Read keyboard
27         if (read (fd, &ev, event_size) < event_size){
28             // No event
29             continue;
30         }
31         if (ev.type == EV_KEY && ev.value == KEY_PRESSED){
32             printf("Pressed key: 0x%04x\n", (int)ev.code);
33         } else if (ev.type == EV_KEY && ev.value == KEY_RELEASED){
34             printf("Released key: 0x%04x\n", (int)ev.code);
35         }
36     }
37     close(fd);
38     return 0;
39 }
```

Figure 4: Reading keyboard input.

Important lines of code in Figure 5 are discussed below:

- Lines 9 - 22 implement the function that is executed in the audio thread. As shown, this function has to have the type `void *`.
- Line 14 calls the function `pthread_testcancel`, which checks if the thread has been cancelled by the main thread. If it has been cancelled, the audio thread will halt execution at this point and terminate.
- Line 31 calls `pthread_create` to create the audio thread. The arguments are:
  - *&pid*: the variable in which the *thread id*, a unique integer identifier for the thread, will be stored.
  - NULL: a pointer to an *attributes* structure can be passed to specify some properties of the spawned thread. A NULL argument causes the thread to use default attributes.
  - *&audio_thread*: a pointer to the `audio_thread` function that we want to execute in the audio thread.
  - NULL: This argument can be any value, and in general practice would be a pointer to a structure containing all of the variables (arguments) that we want to pass into the thread. In our case the variables

that are needed by the `audio_thread`, such as `tone_volume`, are global variables, which do not need to be passed to the thread as arguments.

- Line 39: the call to the function `pthread_cancel` sends a *cancel* signal to the thread, which causes the audio thread to stop executing when it reaches `pthread_testcancel`.
- Line 41: the `pthread_join` function forces the main thread to wait until the audio thread has been terminated before continuing past this point.

```
1  #include <math.h>
2  #include <linux/input.h>
3  #include <pthread.h>
4  ... other include statements
5
6  int tone_volume[13];
7  ... other global variables (not shown)
8
9  void *audio_thread( ){
10     int i;
11     while(1){
12         // Check if this thread has been cancelled.
13         // If it was, the thread will halt at this point.
14         pthread_testcancel();
15         ...
16         // Code for writing to the audio port.
17         for (i=0; i < NUM_OF_TONES; i++){
18             ... tone_volume[i] ...
19         }
20         ...
21     }
22 }
23
24 int main (int argc, char *argv[])
25 {
26     int err;
27     pthread_t tid;
28     ...
29
30     // Spawn the audio thread.
31     if ((err = pthread_create(&tid, NULL, &audio_thread, NULL)) != 0)
32         printf("pthread_create failed:[%s]\n", strerror(err));
33
34     while (1){
35         // Read keyboard and adjust tone_volume[].
36         ...
37     }
38     // Cancel the audio thread.
39     pthread_cancel(tid);
40     // Wait until audio thread terminates.
41     pthread_join(tid, NULL);
42
43     return 0;
44 }
```

Figure 5: Spawning the audio thread.

Using a Mutex

If multiple threads in a program need to modify a single, shared, variable it is useful to synchronize access to that variable by using a *mutex*. For example, Figure 6 indicates how a mutex can be used to synchronize writes to the `tone_volume` array by both the main and audio threads. In this code the mutex is declared as a global variable, and then *locked/unlocked* by each thread. The main thread writes to `tone_volume` when a piano key is pressed, and the audio thread gradually reduces the volume of a tone depending on whether the key is still pressed. More details about mutexes and pthreads can be found by searching on the Internet.

Perform the following:

1. Make a new folder to hold your C source code. Augment your code from Part II so that it uses two threads as illustrated in Figure 5. In your main thread open the appropriate keyboard device in the folder */dev/input*, to provide your piano keys. When you press a key the tone should immediately sound, and then gradually fade away as indicated in Figure 6.
2. Compile your program using a command such as `gcc -o part3 part3.c -lm -pthread`

```c
...
int tone_volume[13];
pthread_mutex_t mutex_tone_volume;  // mutex for main and audio_thread
...

void *audio_thread( ){
    ... code for creating and outputting waveforms not shown
    // Fade the tones
    for (j = 0; j < 13; j++){
        if (tone_volume[j] > ...){
            pthread_mutex_lock (&mutex_tone_volume);    // lock the mutex
            tone_volume[j] *= tone_fade_factor[j];
            pthread_mutex_unlock (&mutex_tone_volume);  // unlock the mutex
        }
    }
}

int main (int argc, char *argv[])
{
    ...
    pthread_t tid;
    if ((err = pthread_create(&tid, NULL, &audio_thread, NULL)) != 0)
        printf("pthread_create failed:[%s]\n", strerror(err));
    while (1){
        // Read keyboard and adjust tone_volume[].
        ...
        pthread_mutex_lock (&mutex_tone_volume);    // lock the mutex
        tone_volume[tone_index] = ...
        pthread_mutex_unlock (&mutex_tone_volume);  // unlock the mutex
        tone_fade_factor[tone_index] = ...
    }
    ...
}
```

Figure 6: Using a mutex to control access to the `tone_volume` array.

Part IV

Figure 7 shows two sinusoidal waveforms. The top waveform represents middle C, also known as tone C4, which has the frequency 261.626 Hz, and the bottom waveform represents tone C5, which has the frequency 523.251 Hz. For this part of the exercise you are to augment your piano code such that it can draw waveforms on a VGA display corresponding to the sound that is currently being played. If a single tone is being played, then your waveform will be a sine wave, like the ones depicted in Figure 7, but if a chord is being played then you would display the waveform represented by the sum of the individual tones. Rather than drawing waveforms continuously, just draw a few cycles whenever the piano "keys" are pressed, or released. *Hint*: at the sampling rate of 48,000 samples/second, there are $48000 \div 261.626 \simeq 184$ samples per cycle for tone C4. Given that there are 320 columns in the VGA display for the DE1-SoC Computer, it is easy to display about $320 \div 184 \simeq 1.7$ cycles of this waveform. Using the same approach you could display about $320 \div (48000 \div 523.251) \simeq 3.5$ cycles of tone C5.

Perform the following:

1. Make a new folder to hold your C source code. Augment your code from Part III to create a third thread called *video_thread*. In this new thread you should provide the code for drawing waveforms on the VGA display. To communicate with the VGA controller in the DE1-SoC Computer you can use a character device driver, such as the ones described in Laboratory Exercise 6.

   Since there are now three threads in your program and only two CPU cores in the DE1-SoC Computer, the threads must share the processor cores. It is important to ensure that the audio thread always has access to a CPU core so that the audio-port's *Write* FIFOs do not become empty. One approach for handling the three threads is to assign the main and video threads to one CPU core, while assigning the audio thread to the other core. Figure 8 provides a function that can be called by a thread to assign it to a specific CPU core. In the main and video threads you could call `set_processor_affinity(0)`, assigning these threads to CPU core 0, and in the audio thread `set_processor_affinity(1)`, assigning this thread to CPU core 1. To use the `CPU_ZERO` and `CPU_SET` macros in Figure 8 you have to define `_GNU_SOURCE` and include the header file `<sched.h>`, as shown in the code.

2. Compile and test your program. Check that tones and chords that are played on the piano sound the same as they did in Part III, confirming that your threads are working properly. Check that your video thread draws appropriate waveforms for each tone and chord played on the piano.

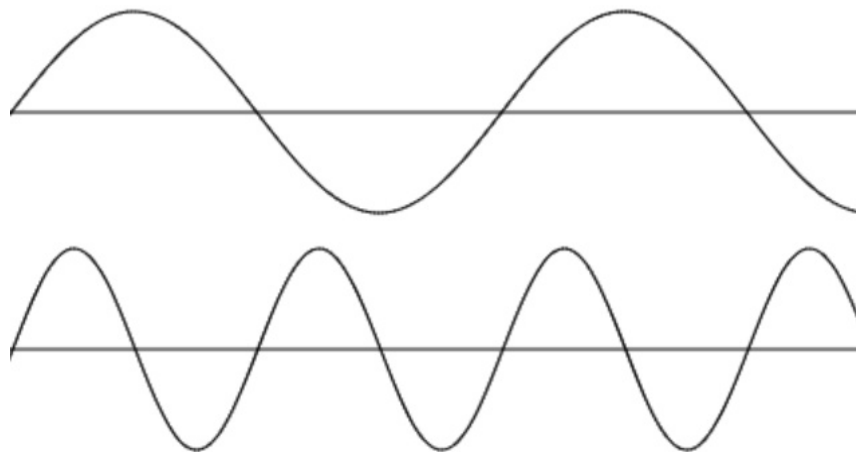

Figure 7: Sine waves for tones C4 and C5.

```c
#define _GNU_SOURCE
#include <pthread.h>
#include <sched.h>
...
// Set the current thread's affinity to the core specified
int set_processor_affinity(unsigned int core){
    cpu_set_t cpuset;
    pthread_t current_thread = pthread_self();

    if (core >= sysconf(_SC_NPROCESSORS_ONLN)){
        printf("CPU Core %d does not exist!\n", core);
        return -1;
    }

    // Zero out the cpuset mask
    CPU_ZERO(&cpuset);
    // Set the mask bit for specified core
    CPU_SET(core, &cpuset);

    return pthread_setaffinity_np(current_thread, sizeof(cpu_set_t),
        &cpuset);
}
```

Figure 8: Assigning a thread to a CPU core.

Part V

For this part you are to further enhance your code from Part IV so that it is possible to record, for later playback, music that is played on the piano.

Perform the following:

1. Augment your source code from Part IV to create a data structure that can be used to store recorded music. In this data structure you should record when each piano key is pressed, and when it is released. Both the recoding process and playback should be handled by the main thread in your program. To keep track of time you can use a timer, like the *stopwatch* that was discussed in Laboratory Exercise 4, Part 2. This *stopwatch* provides a character device driver via the file */dev/stopwatch*, which provides the current time when read, and can be written to set the time. The *stopwatch* decrements at a rate of $1/100^{th}$ seconds, which is sufficiently accurate for our purposes.

2. You should start recording notes when the user presses the pushbutton KEY_0 in the DE1-SoC Computer. Pressing this button again should stop the recording. The recording should be played back when KEY_1 is pressed. You should illuminate $LEDR_0$ while recording music, and $LEDR_1$ during playback. In addition to using the *stopwatch* to measure time during recording, you can also use it to control the time between notes during playback. Show the *stopwatch* time on the seven-segment displays HEX5 - HEX0.

3. You can use character devices drivers to communicate with the KEY pushbuttons, LEDR lights, and seven-segment displays. Suitable drivers were developed in Laboratory Exercise 3.

4. Compile and test your program. During both recording and playback of music you should be able to hear the music being played and see the corresponding waveforms on the VGA display.

Part VI

So far in this exercise you have used memory-mapped I/O to communicate with the audio-out port. For this part, you are to modify your approach so that your program communicates with the audio port through a character device driver.

Perform the following:

1. Create a Linux character device driver, in the form of a kernel module, that provides a file-based I/O interface to the audio-out port. Write the driver code, which must create the file */dev/audio* in the Linux filesystem, in a file *audio.c*. You should be able to write the following commands to /dev/audio: `init`, which clears the audio FIFOs, `wait`, which checks the depths of the *Write* FIFOs and waits for some free space, `left LLLLLLLL`, and `right RRRRRRRR`. These last two commands write the supplied data, which is specified as 8-digit hexadecimal numbers `LLLLLLLL` and `RRRRRRRR`, into the audio-port's left and right data registers. Refer to Exercise 3 for a detailed discussion about the functions and variables that are needed for character device drivers.
2. Create a Makefile to compile your character device driver, and then insert the driver into the Linux kernel.
3. Change your piano source code so that it uses your audio character device driver instead of using memory-mapped I/O. Compile the modified program and test to ensure that it works the same as it did in Part V.

Copyright © Intel Corporation.