Sean O’Day

ECE 4220

Final project

5 June 2015

# Guitar looping station

**Abstract**

The looping pedal takes an iput audio sound and repeats it. Ideally, these samples can be layered and played over with various instruments. The device built here utilizes a Beaglebone Black and USB soundcard to create the looping effect. OpenAL was used to manipulate the audio. Buttons were wired to three of the GPIO pins and there is a kernel module that loads a device driver to drive the buttons and interface them with the user space program.

**Introduction**

The goal of this project was to create a guitar looping pedal using I2C controlled ADC and DAC to input sounds and repeat and layer them. In practice, the timing was difficult to nail down for these so a USB sound card was used instead. OpenAL was used to interface with the soundcard to create the loop. The intention was to use a summing amplifier to layer the sound of the guitar playing with the looped output. This circuit could not be completed due to a lack of resources (access to an oscilloscope was crucial and all of the labs are in the process of moving). The looping device has three buttons, one to record, one to playback and a third to exit the program cleanly.

**Background**

Looping pedals and digital effects are not a new concept, however most looping devices for guitar are quite expensive (on the order of $200-300). Even this low functionality is very difficult to achieve and commercial devices are very robust and have a lot of access to multiple samples.

The Beaglebone pedal is relatively cheap in comparison, with the Beaglebone at around $45 and a soundcard about $10. It is a much more affordable option and given the full operating system on board the Beaglebone, there is a lot of room for multiple effects beyond simple looping that can be programmed using the OpenAL libraries.

**Implementation**

The hardware of the finished project consists of a Beaglebone Black with a USB soundcard and a biased non inverting amplifier. Buttons are wired in a pull up configuration to the GPIO pins. This was accmomplished by wiring the 3.3 V supply on the Beaglebone through one end of the buttons. The other side of the buttons were connected through using 100 Ω resistors to the GPIO pins and 10 kΩ resistors to ground on the board.

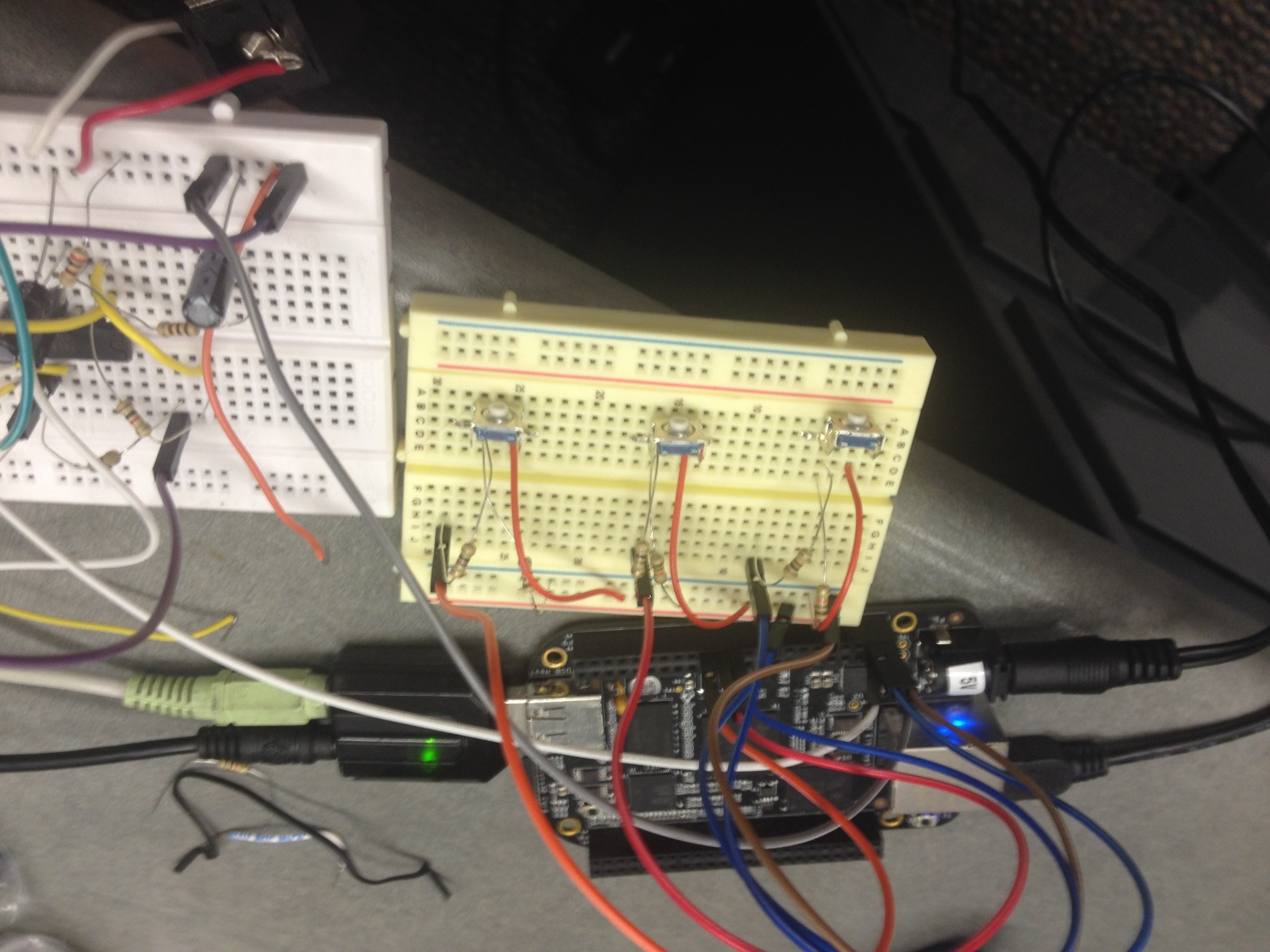


Figure : Picture of the button GPIO configuration

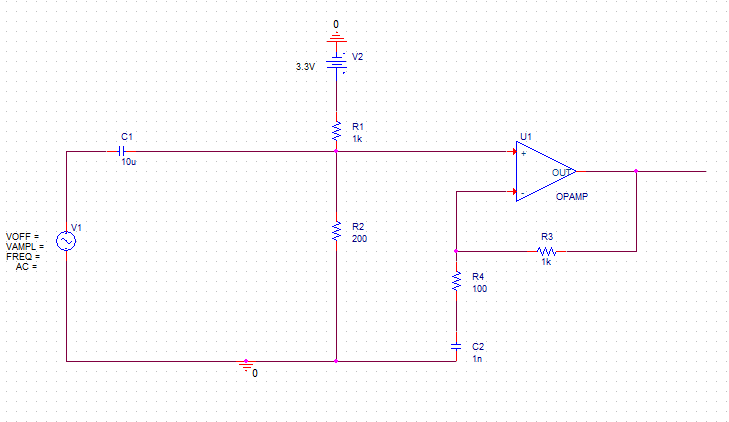
Guitars are high impedance devices (~1 MΩ), that produce a relatively small AC voltage. The sound card however can only read voltages above -0.3 V and is low impedance (30-50 Ω). To solve this problem, a biased non inverting amplifier was built using a TL974 op amp to first bias the guitar signal and then amplify it so that the line in of the sound card could read the voltages supplied by the guitar. This circuit schematic is shown below:

Figure : Schematic of biasing non-inverting amplifier

The guitar is wired into the circuit using a cheap jack purchased at Radioshack. This signal is decoupled using a 10 μF capacitor and through a combined 166 Ω resistor. This effectively creates a high pass filter, so these values were chosen such that the cutoff frequency would be relatively low, about 95 Hz:

The feedback resistor was adjusted such that the gain of the circuit would be about 11, this was necessary as the guitar output voltage was approximately 250 mV at the highest it was measured and the soundcard needed a range from -0.3 V to 3.3 V for optimum usage.

The output of the amplifier was attached to the headphone jack of the USB soundcard. After the soundcard, an attempt was made to create a summing amplifier to sum the guitar signal and the sound card output. This was unsuccessful, but can still be seen in the photograph of the actual circuit seen below.

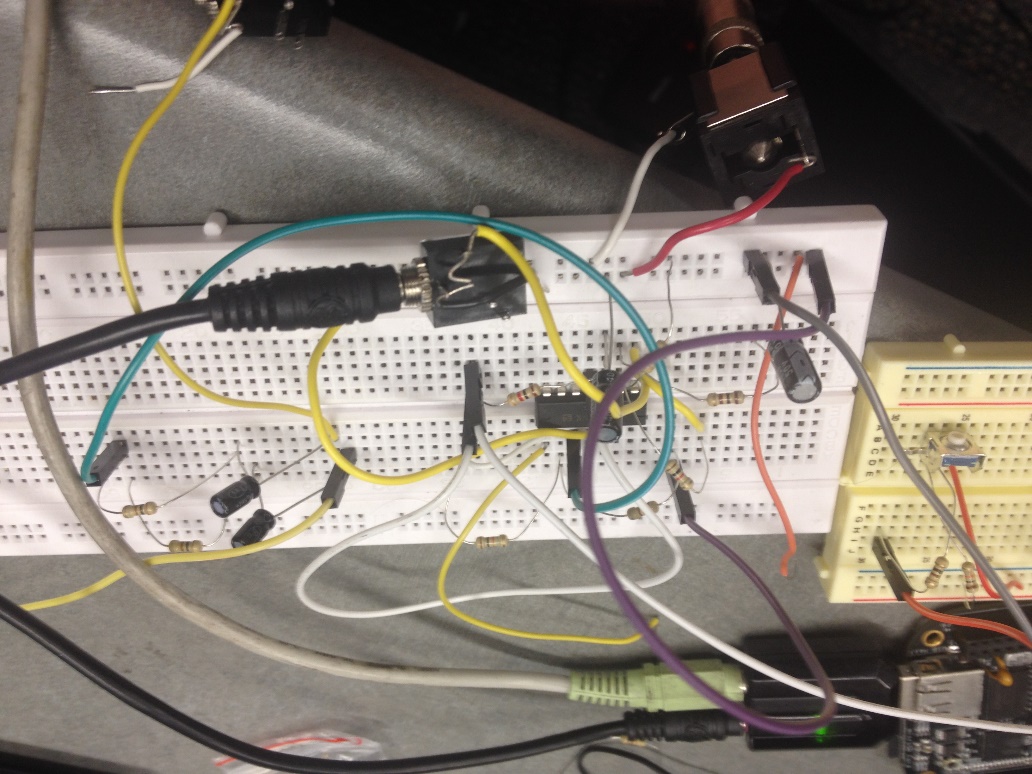


Figure : Photo of amplifier circuit. Bottom left, the guitar input can be seen, with the headphone out to its right

The software implementation of this project is fairly straightforward and is loosely outlined below:



Figure : Basic implementation of entire system.

A user space program was created that opens the USB sound card and uses OpenAL to create loops. This program spawns a thread that reads the button inputs and sets flags for the main thread to check and determine what function should be used. The main thread then spawns either a playback or record thread depending on which button was pressed. The third button is simply used to exit the program.

Meanwhile, a kernel module interfaces the buttons with the GPIO pins and handles the associated interrupts. The kernel module initializes the GPIO pins and gets IRQs for them. These are then associated with the handlers that will determine which button is pushed.

During initialization, the module also registers a device in the operating system that will be opened by the user space program. This character device driver takes the output of the interrupts handlers and passes it into the user space program using this device. During the read function of this driver, the calling process is placed in a wait queue until the interrupts are fired and signal that the wait is over and data is ready.

These interrupts register which button was pressed and pass a letter to the device to be read on the other side indicating the button that was pushed. Once the data is set by the interrupt, the handler then wakes up the read process and sets a Boolean value indicating the device read can continue,

**Experiments and Results**

There was a lot of trial and error involved in the making of this project. I initially attempted to create my own sort of soundcard using an I2C controlled ADC. While I was able to interface with this card it was difficult to use both it and the DAC I had initially planned to use. Both of these devices output data using I2S and there is only one such bus on the Beaglebone Black.

Early in the project, the Beaglebone I had malfunctioned and I attempted to use a replacement board: the Odroid C1. This is a powerful Linux computer with a real time clock and quite a few options for configuration. Unfortunately, it is a fairly new piece of equipment and, as such, there was little documentation that could be found on how to perform basic tasks using it. The datasheet released for the processor contained almost no information, and made it very difficult to map the registers for the GPIOs and for the interrupts I desired to use. The information must exist somewhere though, as GPIO libraries exist for the device. I began reverse engineering these, but gave up before I sank too much time into it.

There was quite a bit of experimentation that went into writing the device driver for the buttons. The main experimentation involved using various methods of synchronization so that the device would not read uncontrollably, but also would not block in the read function and starve the remaining threads.

Without any synchronization the button would simply be read continuously, which led to record and playback threads from running correctly. This also meant much more of the CPU was being spent reading the buttons, as it had the effect of polling them.

An attempt was then made to use semaphores to simply block the read portion of code until data was available, however this seemed to prevent the other threads from executing while the device was being read. In retrospect, this approach may have worked had I been more careful.

I also experimented with the summing amplifier somewhat extensively, however I was not ever able to get it to work the way I wanted it to. This was largely unsuccessful due to a lack of measurement instruments. I spent some time attempting to create a rudimentary oscilloscope using LabView and a DAQ unit that I had access to. I rightly decided this was not worth the effort and scrapped the summing amplifier.

**Discussion and Conclusion**

Overall this project went fairly well, though I can think of quite a few improvements I would make to the device given more time and resources to implement them. I believe the use of the USB sound card was a good decision, as it allowed the use of the OpenAL library. This library has a lot of features that could be used in the future to implement some digital effects into the design.

Without a doubtm the most difficult part of this project was writing and synchronizing the device driver for the buttons. This required using quite a few libraries and methods that I was entirely unfamiliar with to interface with the kernel module. I feel like this was the most valuable learning experience for me in this project.

The other part of the project that provided a great learning experience was the attempt at using the I2C devices. This was also something I had not done before and offered quite a bit of insight into how to map the registers for these devices and set them to do the intended task. While these devices were not part of the final product, it still provided quite an experience.

Given more time I would certainly attempt to get the summing amplifier working, as this was simply limited access to measurement tools that prevented me from implementing this correctly. While I knew roughly what the input was to the sound card, I had no idea what type of signal was being played out of it. This meant that I may have been attempting to sum two very disparate signals that may have required some manipulation before being added together. I would also implement a way to pipe the output of the sound card back into its input so that layers of loops could be recorded. This provide a lot more usefulness to the device for multi-tracking loops and samples.

Other improvements might include trying to use some of the real time kernels available for the Beaglebone. I would also attempt to become more familiar with the Odroid C1, as it is a powerful board. It is a shame that I could not find more information on how to interface it.

Appendix

CODE

User space program

#include <stdio.h>

#include <unistd.h>

#include <AL/al.h>

#include <AL/alc.h>

#include <sys/time.h>

#include <sys/stat.h>

#include <sys/mman.h>

#include <ctime>

#include <fstream>

#include <string>

#include <cstring>

#include <cstdlib>

#include <iostream>

#include "SimpleGPIO.h"

#include <pthread.h>

#include <fcntl.h>

#define MAX\_NUM\_BUFFERS 10 // Maximum number of bufffers in a bufferArray

#define BYTES\_PER\_BUFFER 1048576 // Number of ALubytes per sample.

#define FREQUENCY 44100 // Sampling frequency.

#define MAX\_CAPTURE\_CHUNK 1024 // Maximum chunk of samples to each time in the while loops.

using namespace std;

unsigned int recGPIO = 15; // GPIO0\_15 = (0x32) + 15 = 15

unsigned int playGPIO = 60; // GPIO1\_28 = (1x32) + 28 = 60

int stopPressed = 0;

int fd;

class openAL{

private:

const ALCchar \*devices;

const ALCchar \*ptr;

ALCcontext \*mainContext;

ALCdevice \*mainDev;

ALCdevice \*captureDev;

ALubyte \*\*bufferArray;

ALubyte \*captureBufPtr;

ALint samplesAvailable;

ALint samplesCaptured;

time\_t currentTime;

time\_t lastTime;

ALuint \*buffer;

ALint playState;

int final\_capture\_index;

int bufferCounter;

public:

unsigned int NUM\_ACTIVE\_BUFFERS;

ALuint source;

openAL();

~openAL();

void listPlaybackDevices(void);

void listCaptureDevices(void);

void playbackSound(void);

int captureSound(void);

void endOpenAL(void);

int preparePlayback(void);

unsigned int playPB, recPB;

void reallocateBuffers(void);

};

openAL::openAL(){

// Initialize GPIO ports

gpio\_export(recGPIO);

gpio\_set\_dir(recGPIO, INPUT\_PIN);

gpio\_export(playGPIO);

gpio\_set\_dir(playGPIO, INPUT\_PIN);

recPB = HIGH;

playPB = HIGH;

// Initialize bufferArray

bufferArray = new ALubyte\*[MAX\_NUM\_BUFFERS];

int i;

for(i = 0; i < MAX\_NUM\_BUFFERS; i++){

bufferArray[i] = new ALubyte[BYTES\_PER\_BUFFER];

}

NUM\_ACTIVE\_BUFFERS = 0;

final\_capture\_index = 0;

}

openAL::~openAL(){

int i;

for(i = 0; i < MAX\_NUM\_BUFFERS; i++){

delete[] bufferArray[i];

}

delete[] bufferArray;

}

// Prints the List of Playback Devices

void openAL::listPlaybackDevices(void){

printf("Available playback devices:\n");

devices = alcGetString(NULL, ALC\_DEVICE\_SPECIFIER);

ptr = devices;

while(\*ptr){

printf(" %s\n", ptr);

ptr += strlen(ptr) + 1;

}

}

// Print the list of capture devices.

void openAL::listCaptureDevices(void){

printf("Available capture devices:\n");

devices = alcGetString(NULL, ALC\_CAPTURE\_DEVICE\_SPECIFIER);

ptr = devices;

while (\*ptr){

printf(" %s\n", ptr);

ptr += strlen(ptr) + 1;

}

}

// Opens the default playback device and prepares it for playback.

int openAL::preparePlayback(void){

// Open the default playback device.

printf("Opening playback device...\n");

mainDev = alcOpenDevice(NULL);

if (mainDev == NULL) {

printf("Error. Unable to open playback device.\n");

return 0;

}

// Create a playback context.

devices = alcGetString(mainDev, ALC\_DEVICE\_SPECIFIER);

printf("Opened device '%s'\n", devices);

mainContext = alcCreateContext(mainDev, NULL);

if (mainContext == NULL)

{

printf("Error. Unable to create playback context.\n");

return 0;

}

// Make the playback context current

alcMakeContextCurrent(mainContext);

alcProcessContext(mainContext);

// Generate an array of openAL buffers and a single source to attatch them to.

buffer = new ALuint[NUM\_ACTIVE\_BUFFERS];

alGenBuffers(NUM\_ACTIVE\_BUFFERS, buffer);

alGenSources(1, &source);

// Fill the openAL buffer array with the data from the captured bufferArray.

int i;

for(i=0; i < NUM\_ACTIVE\_BUFFERS; i++){

if(i == NUM\_ACTIVE\_BUFFERS - 1)

alBufferData(buffer[i], AL\_FORMAT\_MONO16, bufferArray[i], final\_capture\_index , 44100);

else

alBufferData(buffer[i], AL\_FORMAT\_MONO16, bufferArray[i], BYTES\_PER\_BUFFER , FREQUENCY);

}

// Attatch the buffers to the source.

alSourceQueueBuffers(source, NUM\_ACTIVE\_BUFFERS, buffer);

return 1;

}

// Plays back the previously captured sound.

void openAL::playbackSound(void){

printf("Starting playback...\n");

fflush(stdout);

// Play the source, stop if the playback button is pressed again.

alSourcePlay(source);

playState = AL\_PLAYING;

playPB = HIGH;

alSourcei(source, AL\_LOOPING,1);

while((playState == AL\_PLAYING) && (playPB != LOW))

{

printf("Source is playing...\r");

fflush(stdout);

alGetSourcei(source, AL\_SOURCE\_STATE, &playState);

gpio\_get\_value(playGPIO, &playPB);

}

alSourceStop(source);

}

// Captures audio from the defualt device.

int openAL::captureSound(void){

// Open the default device

printf("Opening capture device:\n");

captureDev = alcCaptureOpenDevice(NULL, FREQUENCY, AL\_FORMAT\_MONO16, MAX\_CAPTURE\_CHUNK);

if(captureDev == NULL){

printf("Error. Unable to open device!\n");

return 0;

}

devices = alcGetString(captureDev, ALC\_CAPTURE\_DEVICE\_SPECIFIER);

printf("Opened device %s\n.", devices);

// Cowntdown to capture start.

int i;

for (i = 3; i > 0; i--) {

printf("Starting capture in %d...\r", i);

fflush(stdout);

lastTime = time(NULL);

currentTime = lastTime;

while(currentTime == lastTime){

currentTime = time(NULL);

usleep(100000);

}

}

// Initialize capture device and variables.

printf("Starting capture...\n");

fflush(stdout);

alcCaptureStart(captureDev);

samplesCaptured = 0;

recPB = HIGH;

captureBufPtr = bufferArray[0];

NUM\_ACTIVE\_BUFFERS++;

int buffer\_index = 0;

double MAX\_TIME = BYTES\_PER\_BUFFER / (2 \* FREQUENCY); // Maximum recording time.

//MAX\_TIME -= NUM\_ACTIVE\_BUFFERS;

cout << "The maximum capture time is " << MAX\_TIME << " seconds." << endl;

// Capture audio until the button is pressed or we run out of memory.

currentTime = time(NULL);

time\_t startTime = currentTime;

while((currentTime < (startTime + MAX\_TIME))){

printf("Capture time: %d\r", currentTime - startTime);

fflush(stdout);

// Copy the samples to the current capture buffer.

alcGetIntegerv(captureDev, ALC\_CAPTURE\_SAMPLES, 1, &samplesAvailable); // Get the number of samples available

if(samplesAvailable > 0){

alcCaptureSamples(captureDev, captureBufPtr, samplesAvailable);

samplesCaptured += samplesAvailable;

// Advance the buffer and buffer\_index (two bytes per sample \* number of samples)

captureBufPtr += samplesAvailable \* 2;

buffer\_index += samplesAvailable \* 2;

}

currentTime = time(NULL);

//gpio\_get\_value(recGPIO, &recPB);

if(stopPressed){

break;

}

}

// Stop capture.

printf("\nCapture Stopped.\n");

cout << "There are " << NUM\_ACTIVE\_BUFFERS << " buffers active." << endl;

alcCaptureStop(captureDev);

final\_capture\_index = buffer\_index;

return 1;

}

// Deletes and reallocates the buffers, effectively clearing them.

void openAL::reallocateBuffers(void){

int i;

for(i=0; i < MAX\_NUM\_BUFFERS; i++){

delete[] bufferArray[i];

}

delete[] bufferArray;

bufferArray = new ALubyte\*[MAX\_NUM\_BUFFERS];

for(i = 0; i < MAX\_NUM\_BUFFERS; i++){

bufferArray[i] = new ALubyte[BYTES\_PER\_BUFFER];

}

NUM\_ACTIVE\_BUFFERS = 0;

final\_capture\_index = 0;

cout << "Buffers reallocated." << endl;

}

int recordFlag = 0, playFlag = 0 ,exitFlag = 0;

openAL looper;

//thread to check buttons

void \*buttonThread(void \*arg){

char buffer;

while(1){

read(fd,&buffer,sizeof(buffer));

printf("%c\n",buffer);

if(buffer=='R'){

if(recordFlag){

stopPressed = 1;

recordFlag = 0;

}

}

if(buffer=='P'){

if(playFlag){

playFlag = 0;

alSourceStop(looper.source);

}

}

if(buffer=='X'){

exitFlag = 1;

}

}

}

//record thread calls capture

void \*recordSample(void \*arg){

if(looper.captureSound() == 0){

return 0;

}

if(looper.preparePlayback() == 0){

return 0;

}

pthread\_exit(0);

}

//main is pretty sparse, infinite loop that calls different functions based on interrupts provided

int main(void) {

pthread\_t recordThread,playbackThread,buttonCheck;

pthread\_create(&buttonCheck,NULL,buttonThread,NULL);

time\_t timeoutCurr, timeoutEnd;

bool flag = true;

int check=mkfifo("button\_presses",S\_IWUSR|S\_IRUSR|S\_IWOTH|S\_IROTH);

fd = open("button\_presses",O\_RDONLY);

looper.listCaptureDevices();

while(1){

if(recordFlag){

pthread\_create(&recordThread,NULL,recordSample,NULL);

}

if(playFlag){

looper.playbackSound();

}

if(exitFlag){

return 0;

}

}

}

Kernel Module:

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/gpio.h> // Required for the GPIO functions

#include <linux/interrupt.h> // Required for the IRQ code

#include <linux/fs.h>

#include <asm/segment.h>

#include <asm/uaccess.h>

#include <linux/buffer\_head.h>

#include <linux/wait.h>

#define DEVICE\_NAME "looperFIFO" ///< The device will appear at /dev/looperFIFO using this value

#define CLASS\_NAME "loops" ///< The device class -- this is a character device driver

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Sean ODay");

MODULE\_DESCRIPTION("A Button driver for the BBB looper");

MODULE\_VERSION("0.1");

static unsigned int gpioRecord = 49;

static unsigned int gpioPlay = 117;

static unsigned int gpioExit = 115; //GPIOs used for buttons

static unsigned int irqRecord;//irqs to be set on init

static unsigned int irqPlay;

static unsigned int irqExit;

//this wait queue is for synchronization of the read function and will put the calling thread into a sleep until data is available

wait\_queue\_head\_t my\_queue;

int intflag;//to be used as boolean value for the wait cue

static int majorNumber; ///< Stores the device number -- determined automatically

static char message[256] = {0}; ///< Memory for the string that is passed from userspace

static short size\_of\_message; ///< Used to remember the size of the string stored

static int numberOpens = 0; ///< Counts the number of times the device is opened

static struct class \*loopClass = NULL; ///< The device-driver class struct pointer

static struct device \*loopDevice = NULL; ///< The device-driver device struct pointer

// The prototype functions for the character driver -- must come before the struct definition

static int dev\_open(struct inode \*, struct file \*);

static int dev\_release(struct inode \*, struct file \*);

static ssize\_t dev\_read(struct file \*, char \*, size\_t, loff\_t \*);

static ssize\_t dev\_write(struct file \*, const char \*, size\_t, loff\_t \*);

/// Function prototype for the custom IRQ handler function -- see below for the implementation

static irq\_handler\_t record\_handler(unsigned int irq, void \*dev\_id, struct pt\_regs \*regs);

static irq\_handler\_t play\_handler(unsigned int irq, void \*dev\_id, struct pt\_regs \*regs);

static irq\_handler\_t exit\_handler(unsigned int irq, void \*dev\_id, struct pt\_regs \*regs);

//maps the functions into the fops struct, this ties the functions created here to the functions in user space

static struct file\_operations fops =

{

.open = dev\_open,

.read = dev\_read,

.write = dev\_write,

.release = dev\_release,

};

static int \_\_init buttongpio\_init(void){

int result = 0;

//check that the gpios are valid

printk(KERN\_INFO "GPIO\_TEST: Initializing the GPIO\_TEST LKM\n");

// Is the GPIO a valid GPIO number (e.g., the BBB has 4x32 but not all available)

if (!gpio\_is\_valid(gpioRecord)){

printk(KERN\_INFO "Invalid GPIO\n");

return -ENODEV;

}

if (!gpio\_is\_valid(gpioPlay)){

printk(KERN\_INFO "Invalid GPIO\n");

return -ENODEV;

}

if (!gpio\_is\_valid(gpioExit)){

printk(KERN\_INFO "Invalid GPIO\n");

return -ENODEV;

}

//sets the direction and debounce of the gpios, also exports a low value for them

gpio\_request(gpioRecord, "sysfs");

gpio\_direction\_input(gpioRecord);

gpio\_set\_debounce(gpioRecord, 200);

gpio\_export(gpioRecord, false);

gpio\_request(gpioPlay, "sysfs");

gpio\_direction\_input(gpioPlay);

gpio\_set\_debounce(gpioPlay, 200);

gpio\_export(gpioPlay, false);

gpio\_request(gpioExit, "sysfs");

gpio\_direction\_input(gpioExit);

gpio\_set\_debounce(gpioPlay, 200);

gpio\_export(gpioPlay, false);

// returns the irq numbers for the gpios

irqRecord = gpio\_to\_irq(gpioRecord);

irqPlay = gpio\_to\_irq(gpioPlay);

irqExit = gpio\_to\_irq(gpioExit);

printk(KERN\_INFO "GPIO\_TEST: The button is mapped to IRQ: %d\n", irqRecord);

// This next call requests an interrupt line

result = request\_irq(irqRecord,(irq\_handler\_t) record\_handler,IRQF\_TRIGGER\_RISING,"recordGPIO\_handler",NULL);

result = request\_irq(irqPlay,(irq\_handler\_t) play\_handler,IRQF\_TRIGGER\_RISING,"playGPIO\_handler",NULL);

result = request\_irq(irqExit,(irq\_handler\_t) exit\_handler,IRQF\_TRIGGER\_RISING,"exitGPIO\_handler",NULL);

printk(KERN\_INFO "Loop: Initializing the Loop LKM\n");

// allocate major number for device driver

majorNumber = register\_chrdev(0, DEVICE\_NAME, &fops);

if (majorNumber<0){

printk(KERN\_ALERT "Loop failed to register a major number\n");

return majorNumber;

}

printk(KERN\_INFO "Loop: registered correctly with major number %d\n", majorNumber);

// Register the device class

loopClass = class\_create(THIS\_MODULE, CLASS\_NAME);

if (IS\_ERR(loopClass)){ // Check for error and clean up if there is

unregister\_chrdev(majorNumber, DEVICE\_NAME);

printk(KERN\_ALERT "Failed to register device class\n");

return PTR\_ERR(loopClass); // Correct way to return an error on a pointer

}

printk(KERN\_INFO "loop: device class registered correctly\n");

// Register the device driver

loopDevice = device\_create(loopClass, NULL, MKDEV(majorNumber, 0), NULL, DEVICE\_NAME);

if (IS\_ERR(loopDevice)){ // Clean up if there is an error

class\_destroy(loopClass); // Repeated code but the alternative is goto statements

unregister\_chrdev(majorNumber, DEVICE\_NAME);

printk(KERN\_ALERT "Failed to create the device\n");

return PTR\_ERR(loopDevice);

}

//initialize the wait queue variable

init\_waitqueue\_head(&my\_queue);

printk(KERN\_INFO "loop: device class created correctly\n"); // Made it! device was initialized

return 0;

return result;

}

static void \_\_exit buttongpio\_exit(void){

free\_irq(irqRecord, NULL);

free\_irq(irqPlay, NULL);

free\_irq(irqExit, NULL);

gpio\_unexport(gpioRecord);

gpio\_unexport(gpioPlay);

gpio\_unexport(gpioExit);

gpio\_free(gpioRecord);

gpio\_free(gpioPlay);

gpio\_free(gpioExit);

device\_destroy(loopClass, MKDEV(majorNumber, 0)); // remove the device

class\_unregister(loopClass); // unregister the device class

class\_destroy(loopClass); // remove the device class

unregister\_chrdev(majorNumber, DEVICE\_NAME); // unregister the major number

printk(KERN\_INFO "Goodbye!\n");

}

//interrupt handlers, pretty straightforward, send a character via the message to user space

static irq\_handler\_t record\_handler(unsigned int irq, void \*dev\_id, struct pt\_regs \*regs){

static char buffer = 'R';

sprintf(message,"%c",buffer);

//wake up read and set boolean so it will be read by the user space

wake\_up\_interruptible(&my\_queue);

intflag=1;

// printk(KERN\_INFO "Record Interrupt! (button state is %d)\n", gpio\_get\_value(gpioRecord));

//pipe\_write(fd, &buffer,sizeof(buffer));

return (irq\_handler\_t) IRQ\_HANDLED; // Announce that the IRQ has been handled correctly

}

static irq\_handler\_t play\_handler(unsigned int irq, void \*dev\_id, struct pt\_regs \*regs){

static char buffer = 'P';

sprintf(message,"%c",buffer);

wake\_up\_interruptible(&my\_queue);

intflag=1;

// printk(KERN\_INFO "Play Interrupt! (button state is %d)\n", gpio\_get\_value(gpioPlay));

return (irq\_handler\_t) IRQ\_HANDLED; // Announce that the IRQ has been handled correctly

}

static irq\_handler\_t exit\_handler(unsigned int irq, void \*dev\_id, struct pt\_regs \*regs){

static char buffer = 'X';

sprintf(message,"%c",buffer);

wake\_up(&my\_queue);

intflag=1;

// printk(KERN\_INFO "Exit Interrupt! (button state is %d)\n", gpio\_get\_value(gpioExit));

//pipe\_write(fd, &buffer,sizeof(buffer));

//numberPresses++; // Global counter, will be outputted when the module is unloaded

return (irq\_handler\_t) IRQ\_HANDLED; // Announce that the IRQ has been handled correctly

}

//function to be called when device is opened, nothing really happens here

static int dev\_open(struct inode \*inodep, struct file \*filep){

numberOpens++;

printk(KERN\_INFO "loop: Device has been opened %d time(s)\n", numberOpens);

return 0;

}

//read function, the synchronization using the wait queue is key here,

//calling process will be placed into wait queueuntil data is available

static ssize\_t dev\_read(struct file \*filep, char \*buffer, size\_t len, loff\_t \*offset){

int error\_count = 0;

// copy\_to\_user has the format ( \* to, \*from, size) and returns 0 on success

wait\_event\_interruptible(my\_queue,intflag!=0);

intflag=0;

error\_count = copy\_to\_user(buffer,&message[0],strlen(message));

//printk(KERN\_INFO "Buffer is %s\n", message);

if (error\_count==0){ // if true then have success

printk(KERN\_INFO "loop: Sent %d characters to the user\n", size\_of\_message);

return (size\_of\_message=0); // clear the position to the start and return 0

}

else {

printk(KERN\_INFO "loop: Failed to send characters to the user\n", error\_count);

return -EFAULT; // Failed -- return a bad address message (i.e. -14)

}

}

//write function, this will never be used by my program

static ssize\_t dev\_write(struct file \*filep, const char \*buffer, size\_t len, loff\_t \*offset){

//sprintf(message, "%s(%d letters)", buffer, len); // appending received string with its length

size\_of\_message = strlen(message); // store the length of the stored message

printk(KERN\_INFO "loop: Received %d characters from the user\n", len);

return len;

}

//release function just states the device is closed

static int dev\_release(struct inode \*inodep, struct file \*filep){

printk(KERN\_INFO "loop: Device successfully closed\n");

return 0;

}

//these calls are mandatory and tie in the init and exit functions

module\_init(buttongpio\_init);

module\_exit(buttongpio\_exit);