Contents

1 AudioInterface.cpp

```
/**
 * @file AudioInterface.cpp
 *
 * @brief Audio interface
 *
 * This file sends and receives data from the Fe-Pi audio board. It also records
 * audio and plays it back, depending on which tracks have flags set.
 *
 * @author Bryan Cisneros
 */

#include "AudioInterface.h"
#include "portaudio.h"
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <sched.h>
#include <pthread.h>
#include "Globals.h"
#include <string.h>

#define SAMPLE_RATE (44100)
#define AUDIO_LENGTH (441000) // The max length of audio we can record is 10
    seconds
#define CHUNK_SIZE (512) // Handle audio in 512 sample chunks
#define MAX_NUMBER_OF_TRACKS (4) // Maximum number of tracks that are supported
#define CHANNELS (1) // Mono audio

#define AUDIO_THREAD_PRIORITY (80)
#define DEVICE_INDEX (2)

// Variables related to audio tracks
static Track* tracks[MAX_NUMBER_OF_TRACKS] = {};
static int number_of_tracks = 0;

// Array and variables to store and access audio
static int16_t audio[MAX_NUMBER_OF_TRACKS][AUDIO_LENGTH] = {};
static volatile long current_position = 0;
static volatile long write_position = 0;
static volatile long track_length = AUDIO_LENGTH;

void* audio_thread(void *arg);

static int paCallback( const void *inputBuffer, void *outputBuffer,
                       unsigned long framesPerBuffer,
                       const PaStreamCallbackTimeInfo* timeInfo,
                       PaStreamCallbackFlags statusFlags,
                       void *userData );



void audio_add_track(Track* track)
{
    if (number_of_tracks < MAX_NUMBER_OF_TRACKS)
    {
```

```
        tracks[number_of_tracks] = track;
        number_of_tracks++;
    }
    else
    {
        printf("Max number of tracks!\n");
    }

}

void audio_set_track_length(void)
{
    track_length = current_position;
}

void audio_set_track_position(int position)
{
    current_position = position;

    // write position should be one chunk before current, but make sure it's
    // not negative
    write_position = current_position - CHUNK_SIZE;
    if (write_position < 0)
    {
        write_position += track_length;
    }
}

void audio_init(void)
{
    // Create and start the audio thread!
    pthread_t audio_thread_id;
    pthread_create(&audio_thread_id, NULL, audio_thread, NULL);
}

void audio_reset(void)
{
    // Completely clear out the audio buffers
//    memset(audio, 0, MAX_NUMBER_OF_TRACKS * AUDIO_LENGTH * 2);
    for (int i = 0; i < MAX_NUMBER_OF_TRACKS; i++)
    {
        for (int j = 0; j < AUDIO_LENGTH; j++)
        {
            audio[i][j] = 0;
        }
    }
    track_length = AUDIO_LENGTH;
}


void* audio_thread(void *arg)
{
    // Set this thread as a high priority real time thread
    const struct sched_param priority = {AUDIO_THREAD_PRIORITY};
    sched_setscheduler(0, SCHED_FIFO, &priority);

    // Initialize PortAudio
```

```
    PaError err = Pa_Initialize();
    if( err != paNoError )
    {
        printf(  "PortAudio error: %s\n", Pa_GetErrorText( err ) );
    }
    else
    {
        // Create a PortAudio stream
        PaStream *stream;
        PaError err;
        PaDeviceIndex index = DEVICE_INDEX;
        PaStreamParameters input = {index, CHANNELS, paInt16, (Pa_GetDeviceInfo(
            index))->defaultLowInputLatency, NULL};
        PaStreamParameters output = {index, CHANNELS, paInt16, (Pa_GetDeviceInfo
            (index))->defaultLowOutputLatency, NULL};

        // Open the stream
        err = Pa_OpenStream     (&stream, &input, &output, SAMPLE_RATE,
            CHUNK_SIZE, paNoFlag, paCallback, NULL);
        if( err != paNoError )
        {
            printf(  "PortAudio error: %s\n", Pa_GetErrorText( err ) );
        }

        // Start the stream
        err = Pa_StartStream( stream );
        if( err != paNoError )
        {
            printf(  "PortAudio error: %s\n", Pa_GetErrorText( err ) );
        }

        // now that the stream is started, we don't need to do anything else in
        // this thread, other than keep the thread alive, so we'll just sit in a
        // while(1) and just yield anytime this thread may become active. All of
        // the audio processing will be handled in the callback function.
        while(1)
        {
            sched_yield();
        }

        err = Pa_StopStream( stream );
        if( err != paNoError )
        {
            printf(  "PortAudio error: %s\n", Pa_GetErrorText( err ) );
        }

        err = Pa_CloseStream( stream );
        if( err != paNoError )
        {
            printf(  "PortAudio error: %s\n", Pa_GetErrorText( err ) );
        }
    }

    err = Pa_Terminate();
    return NULL;
}
```

```
// This routine will be called by the PortAudio engine when audio is needed.
static int paCallback( const void *inputBuffer, void *outputBuffer,
                       unsigned long framesPerBuffer,
                       const PaStreamCallbackTimeInfo* timeInfo,
                       PaStreamCallbackFlags statusFlags,
                       void *userData )
{
    int16_t* in = (int16_t*) inputBuffer;
    int16_t* out = (int16_t*) outputBuffer;
    unsigned int i;

    // For each frame in the input buffer
    for( i=0; i<framesPerBuffer; i++ )
    {
        // Always pass the incoming audio through to the output
        *out = *in;

        // check each track. If the track is recording, add the current audio
        // to its audio buffer. If the track is playing, add the audio in its
        // buffer to the output
        for (int i = 0; i < number_of_tracks; i++)
        {
            if (tracks[i]->isRecording())
            {
                audio[i][write_position] += *in;
            }

            if (tracks[i]->isPlaying())
            {
                *out += audio[i][current_position];
            }
        }

        // Update the current position. If it has wrapped around, reset to zero
        // and set the 'rollover' flag (used by the main state machine)
        current_position++;
        if (current_position >= track_length || current_position >= AUDIO_LENGTH
            )
        {
            current_position = 0;
            rollover = true;
        }

        // Update the write position, and reset to 0 if necessary
        write_position++;
        if (write_position >= track_length || write_position >= AUDIO_LENGTH)
        {
            write_position = 0;
        }

        // Advance the in and out pointers to the next location
        out++;
        in++;
    }
    return 0;
}
```

2 AudioInterface.h

```
/**
 * @file AudioInterface.h
 *
 * @brief Audio interface
 *
 * This file provides the API to send and receives data from the Fe-Pi audio
     board.
 *
 * @author Bryan Cisneros
 */

#pragma once

#include "Track.h"

/**
 * @brief Audio init
 *
 * Initialize the audio interface and start the audio thread
 *
 * @return void
 */
void audio_init(void);

/**
 * @brief Add track to the audio interface
 *
 * @note Safe to call before audio_init()
 *
 * @param track track to add to audio interface
 *
 * @return void
 */
void audio_add_track(Track* track);

/**
 * @brief Set the track length to the current position
 *
 * @return void
 */
void audio_set_track_length(void);

/**
 * @brief Set the audio position
 *
 * @param position position to set the audio to
 *
 * @return void
 */
void audio_set_track_position(int position);

/**
 * @brief Reset audio
 *
 * Reset the audio interface back to default (and erase all tracks)
```

```
 *
 * @return void
 */
void audio_reset(void);
```

3 Button.cpp

```
/**
 * @file Button.cpp
 *
 * @brief Button class
 *
 * This file implements the Button class, including initialization, button
 * presses, and debouncing
 *
 * @author Bryan Cisneros
 */

#include <bcm2835.h>
#include "Button.h"
#include "Globals.h"
#include <stdio.h>

#define TIMEOUT_COUNT (250)

bool Button::bcm2835_initialized = false;

Button::Button(int pin)
{
    // Make sure the bcm2835 is initialized!
    if(!bcm2835_initialized)
    {
        if (!bcm2835_init())
        {
            printf("bcm2835␣init␣failed!/n");
        }
        bcm2835_initialized = true;
    }

    buttonPin = pin;

    // Set pin as an input
    bcm2835_gpio_fsel(buttonPin, BCM2835_GPIO_FSEL_INPT);

    // Enable pullup on pin
    bcm2835_gpio_set_pud(buttonPin, BCM2835_GPIO_PUD_UP);

    pressed = false;
}


Button::~Button()
{
}

void Button::tick()
{
    // Check for a press, and check that the previous press has already timed
        out
    if (!bcm2835_gpio_lev(buttonPin) && timeout == 0)
    {
        // Press detected! set the flag, and also set the timeout for debouncing
```

```
            pressed = true;
            timeout = TIMEOUT_COUNT;
        }

        // If the timeout is running, decrement it
        if (timeout > 0)
        {
            timeout--;
        }
    }

    bool Button::fell()
    {
        // If the button has been pressed, reset the flag and return true. Otherwise
        // return false;
        if (pressed)
        {
            pressed = false;
            return true;
        }
        else
        {
            return false;
        }

    }
```

4 Button.h

```
/**
 * @file Button.h
 *
 * @brief Button class
 *
 * This file implements the API for the Button class
 *
 * @author Bryan Cisneros
 */

#pragma once

class Button
{
public:
    /**
     * @brief Button constructor
     *
     * @param pin pin to initialize
     * @return void
     */
    Button(int pin);

    /**
     * @brief destructor
     */
    ~Button(void);

    /**
     * @brief Tick
     *
     * This function checks the state of the pin to detect a press. It should be
     * called frequently for proper function.
     *
     * @return void
     */
    void tick();

    /**
     * @brief reports if press was detected
     *
     * @return true if press detected, false otherwise
     */
    bool fell();

private:
    int buttonPin; // pin number
    bool pressed; // bool to record a press
    int timeout; // timeout for debouncing
    static bool bcm2835_initialized; // bool to track bcm2835 initialization
};
```

5 Globals.h

```
/**
 * @file Globals.h
 *
 * @brief Global variables
 *
 * This file contains variables that need to be seen by many modules
 *
 * @author Bryan Cisneros
 */

extern bool recordingMode;  // true if recording mode, false if playing mode
extern bool masterDone;     // true every time the master track starts over.
    will stay true only for one tick cycle
extern int waitingToStart;  // 0 until the first track starts recording, 1 while
     the first track is recording, 2 once the first track finishes
extern bool rollover;       // set by the audio interface every time the track
    wraps around
```

6 Led.cpp

```
/**
 * @file Led.cpp
 *
 * @brief Led clas
 *
 * This file implements the Led class
 *
 * @author Bryan Cisneros
 */

#include "Led.h"
#include "LedInterface.h"
#include <stdio.h>


#define FLASH_TIMEOUT (150)

Led::Led(int channel)
{
    this->channel = channel;
    count = 0;

    // initialize to off and not flashing
    flashing = false;
    led_on = false;
}

Led::~Led(void)
{
}


void Led::tick(void)
{
    // If the LED is flashing, increment the counter. If the timeout is reached,
    // toggle the LED and reset the counter
    if (flashing)
    {
        count++;
        if (count >= FLASH_TIMEOUT)
        {
            if (led_on)
            {
                LedInterface_turnOffLed(channel);
                led_on = false;
            }
            else
            {
                LedInterface_turnOnLed(channel);
                led_on = true;
            }
            count = 0;
        }
    }
}
```

```
void Led::turnOn(void)
{
    // Turn on the LED, and update internal status variables
    LedInterface_turnOnLed(channel);
    led_on = true;
    flashing = false;
}

void Led::turnOff(void)
{
    // Turn off the LED, and update internal status variables
    LedInterface_turnOffLed(channel);
    led_on = false;
    flashing = false;
}

void Led::flash(void)
{
    // Start with the LED on, and update internal status variables
    LedInterface_turnOnLed(channel);
    led_on = true;
    flashing = true;
    count = 0;
}
```

7 Led.h

```
/**
 * @file Led.h
 *
 * @brief Led class
 *
 * This file implements the API for the Led class
 *
 * @author Bryan Cisneros
 */

#pragma once

class Led
{
public:
    /**
     * @brief Led constructor
     *
     * @param channel LED channel to initialize
     * @return void
     */
    Led(int channel);

    /**
     * @brief destructor
     */
    ~Led(void);

    /**
     * @brief Tick
     *
     * This function checks if it is time for the LED to flash. It should be
     * called frequently for proper function.
     *
     * @return void
     */
    void tick(void);

    /**
     * @brief Turn on the LED
     * @return void
     */
    void turnOn(void);

    /**
     * @brief Turn off the LED
     * @return void
     */
    void turnOff(void);

    /**
     * @brief Start flashing the LED
     * @return void
     */
    void flash(void);
```

```
private:
    int channel; // LED channel
    int count; // count used for timing the flashing
    bool flashing; // true if flashing, false otherwise
    bool led_on; // records state of LED
};
```

8 LedInterface.cpp

```
/**
 * @file LedInterface.cpp
 *
 * @brief LedInterface module
 *
 * This file implements the LedInterface module, which acts as a bridge between
 * the Led class and the LedDriver of the other running process. This module
 * updates the shared memory used by both processes with the desired LED states
 *
 * @author Bryan Cisneros
 */

#include "LedInterface.h"
#include <stdint.h>
#include  <stdio.h>
#include <sys/shm.h>

#define ON ((uint8_t)0xFF)
#define OFF ((uint8_t)0x00)

typedef struct
{
    bool update;
    char led_values[12];
} shared_leds_t;

void* shared_pointer = NULL;
shared_leds_t* shared_leds;
int shared_leds_id;


void LedInterface_init(void)
{
    // get shared memory
    shared_leds_id = shmget((key_t)1234, sizeof(shared_leds_t), 0);
    if (shared_leds_id == -1)
    {
        printf("failed to get shared memory\n");
    }
    shared_pointer = shmat(shared_leds_id, (void*)0, 0);
    if (shared_pointer == (void*)-1)
    {
        printf("shmat() failed!\n");
    }
    shared_leds = (shared_leds_t*)shared_pointer;

    // Initialize all LEDs to off
    for (int i = 0; i < 12; i++)
    {
        shared_leds->led_values[i] = OFF;
    }
    shared_leds->update = true;
}

void LedInterface_turnOnLed(int channel)
```

```
{
    // Update value in shared memory and set update flag.
    // The LED driver will then update the LED accordingly
    shared_leds->led_values[channel] = ON;
    shared_leds->update = true;
}

void LedInterface_turnOffLed(int channel)
{
    // Update value in shared memory and set update flag.
    // The LED driver will then update the LED accordingly
    shared_leds->led_values[channel] = OFF;
    shared_leds->update = true;
}
```

9 LedInterface.h

```c
/**
 * @file LedInterface.h
 *
 * @brief LedInterface module
 *
 * This file implements the API for the LedInterface module, which acts as a
    bridge
 * between the Led class and the LedDriver of the other running process. This
    module
 * updates the shared memory used by both processes with the desired LED states
 *
 * @author Bryan Cisneros
 */

#pragma once

/**
 * @brief Initialize the LED interface
 *
 * @return void
 */
void LedInterface_init(void);

/**
 * @brief Turn on an LED
 *
 * This function sets the appropriate value in shared memory so that the LED
 * driver will turn on an LED
 *
 * @param channel LED channel to turn on
 *
 * @return void
 */
void LedInterface_turnOnLed(int channel);

/**
 * @brief Turn off an LED
 *
 * This function sets the appropriate value in shared memory so that the LED
 * driver will turn off an LED
 *
 * @param channel LED channel to turn off
 *
 * @return void
 */
void LedInterface_turnOffLed(int channel);
```

10 Looper.cpp

```
/**
 * @file Looper.cpp
 *
 * @brief Looper class
 *
 * This file implements the Looper class
 *
 * @author Bryan Cisneros
 */

#include "Looper.h"
#include "Globals.h"
#include "AudioInterface.h"
#include <stdio.h>

Looper::Looper(Button* recPlay, Button* startStop, Button* resetButton, Led*
    red_led, Led* green_led)
{
    // Point the member variables to the buttons and LEDs
    recPlayButton = recPlay;
    startStopButton = startStop;
    this->resetButton = resetButton;
    this->red_led = red_led;
    this->green_led = green_led;

    // Initialize to idle state, recording mode
    state = idle;
    recordingMode = true;
    masterTrack = NULL;
}

Looper::~Looper()
{
}

void Looper::tick()
{
    // masterDone is set to true when a rollover (wrap-around) occurs. Rollover
    // is set by the audio interface, so we want to clear it as soon as possible
        ,
    // but we want to make sure that masterDone is true for one (and only one)
    // whole tick process.
    masterDone = rollover;
    if (rollover)
    {
        rollover = false;
    }

    // Run the state machine for each track controller
    for (unsigned i = 0; i < trackControllers.size(); i++)
    {
        trackControllers[i]->tick();
    }

    // Run the state machine for each button
```

```
recPlayButton->tick();
startStopButton->tick();
resetButton->tick();

// Check if any of the buttons have been pressed
bool recPlayButtonPressed = recPlayButton->fell();
bool startStopButtonPressed = startStopButton->fell();
bool resetButtonPressed = resetButton->fell();

if (resetButtonPressed)
{
    resetPressed();
}


if (startStopButtonPressed)
{
    printf("Start/stop button pressed!\n");

    // If we were in the stopped state, call the startButton function.
    // If we were in any other state, call the stopButton function
    switch (state)
    {
    case Looper::stopped:
        //start playing again
        startButton();

        //move to normal operation
        state = normalOperation;
        break;
    default:
        stopButton();
        state = stopped;
        break;
    }
}


//state action
switch (state)
{
case Looper::idle:
    break;
case Looper::firstRecording:
    break;
case Looper::normalOperation:
    if (recPlayButtonPressed)
    {
        printf("recPlay button pressed!\n");
        recordingMode = !recordingMode;
        if (recordingMode)
        {
            // In recording mode, red should be on and green off
            green_led->turnOff();
            red_led->turnOn();
        }
        else
```

```
            {
                // In playing mode, red should be off and green on
                red_led->turnOff();
                green_led->turnOn();
            }
        }
        break;
    case Looper::stopped:
        break;
    default:
        break;
    }

    //state update
    switch (state)
    {
    case Looper::idle:
        // Check each track to see if any of them have moved into the recording
        // state. If they have, we have started our first recording. That track
        // becomes our master track, and we move into the first recording state
        for (unsigned i = 0; i < trackControllers.size(); i++)
        {
            if (trackControllers[i]->getState() == TrackController::recording)
            {
                masterTrack = trackControllers[i];
                state = firstRecording;
                printf("looper state: firstRecording\n");
                break;
            }
        }
        break;
    case Looper::firstRecording:
        if (masterTrack->getState() == TrackController::playing)
        {
            // If the master track has moved to the playing state, then our
                first
            // recording is complete. move to the normal operation state
            rollover = false; // make sure rollover is false before normal
                operation
            state = normalOperation;
            printf("looper state: normalOperation\n");
        }
        break;
    case Looper::normalOperation:
        break;
    case Looper::stopped:
        break;
    default:
        break;
    }
}

void Looper::addTrack(TrackController* track)
{
    // Just add the ttrack controller to the vector
    trackControllers.push_back(track);
}
```

```
void Looper::stopButton()
{
    // Call the stop button function on each of the track controllers
    for (unsigned i = 0; i < trackControllers.size(); i++)
    {
        trackControllers[i]->stopButton();
    }
}

void Looper::startButton()
{
    // Reset the audio position back to 0
    audio_set_track_position(0);

    // Call the start button function on each of the track controllers
    for (unsigned i = 0; i < trackControllers.size(); i++)
    {
        trackControllers[i]->startButton();
    }
}

void Looper::resetPressed()
{
    // Reset everything back to default!
    printf("Reset␣button␣pressed!\n");
    state = idle;
    recordingMode = true;
    masterTrack = NULL;
    waitingToStart = 0;

    // Call the reset button function on each of the track controllers
    for (unsigned i = 0; i < trackControllers.size(); i++)
    {
        trackControllers[i]->resetButton();
    }

    // Reset the audio interface
    audio_reset();

    // Reset the LEDs back to just the red one on
    green_led->turnOff();
    red_led->turnOn();
}
```

11 Looper.h

```
/**
 * @file Looper.h
 *
 * @brief Looper class
 *
 * This file implements the API for the Looper class
 *
 * @author Bryan Cisneros
 */

#include "TrackController.h"
#include "Button.h"
#include "Led.h"
#include <vector>

class Looper
{
public:
    /**
     * @brief Looper constructor
     *
     * @param recPlay record/play button
     * @param startStop start/stop button
     * @param resetButton reset button
     * @param red_led Red LED (for recording mode)
     * @param green_led Green LED (for playing mode)
     *
     * @return void
     */
    Looper(Button* recPlay, Button* startStop, Button* resetButton, Led* red_led
        , Led* green_led);

    /**
     * @brief destructor
     */
    ~Looper();

    /**
     * @brief Tick
     *
     * This function runs the state machines, as well as all of the sub state
     * machines (for buttons, leds, track controllers, etc). It should be called
     * frequently for proper function.
     *
     * @return void
     */
    void tick();

    /**
     * @brief add a track to the looper
     *
     * Adding a track will enable the looper to start recording/playing audio on
     * the track.
     *
     * @param track track to add
```

```
     *
     * @return void
     */
    void addTrack(TrackController* track);

private:
    std::vector<TrackController*> trackControllers; // vector of all the track
        controllers

    // Buttons
    Button* recPlayButton;
    Button* startStopButton;
    Button* resetButton;

    // LEDs
    Led* red_led;
    Led* green_led;

    // The master track is the first one recorded. It determines the track
        length, etc
    TrackController* masterTrack;

    // States for the state machine
    enum State { idle, firstRecording, normalOperation, stopped };
    State state;

    // Functions called when the associated buttons are pressed
    void stopButton();
    void startButton();
    void resetPressed();
};
```

12 main.cpp

```
/**
 * @file main.cpp
 *
 * @brief main for looper program
 *
 * This file sets up the looper, tracks, audio interface, buttons, etc, then
 * enters a while(1) loop to periodically run state machines and check for
 * button presses
 *
 * @author Bryan Cisneros
 */

#include "Looper.h"
#include "Button.h"
#include "TrackController.h"
#include "Track.h"
#include "AudioInterface.h"
#include "Led.h"
#include "LedInterface.h"
#include <stdio.h>
#include <bcm2835.h>
#include <time.h>
#include <sched.h>

// Button gpio pin assignements
#define REC_PLAY_BUTTON     (16)
#define RESET_BUTTON        (12)
#define START_STOP_BUTTON   (4)
#define TRACK_1_BUTTON      (25)
#define TRACK_2_BUTTON      (17)
#define TRACK_3_BUTTON      (24)
#define TRACK_4_BUTTON      (23)

#define TWO_MS (2000000L)

bool recordingMode; //true if recording mode, false if playing mode
bool masterDone; //true every time the master track starts over. will stay true
    only for one tick cycle
int waitingToStart; //0 until the first track starts recording, 1 while the
    first track is recording, 2 once the first track finishes
volatile bool rollover; //set by the audio interface every time the track wraps
    around

//Buttons
Button recPlayButton(REC_PLAY_BUTTON);
Button startStopButton(START_STOP_BUTTON);
Button resetButton(RESET_BUTTON);
Button track1Button(TRACK_1_BUTTON);
Button track2Button(TRACK_2_BUTTON);
Button track3Button(TRACK_3_BUTTON);
Button track4Button(TRACK_4_BUTTON);

//Tracks
Track track1;
Track track2;
```

```
Track track3;
Track track4;

//LEDs
Led red1(6);
Led green1(7);
Led red2(5);
Led green2(4);
Led red3(3);
Led green3(2);
Led red4(1);
Led green4(0);
Led record_led(8);
Led play_led(9);

//TrackControllers
TrackController track1Controller(&track1, &track1Button, &red1, &green1);
TrackController track2Controller(&track2, &track2Button, &red2, &green2);
TrackController track3Controller(&track3, &track3Button, &red3, &green3);
TrackController track4Controller(&track4, &track4Button, &red4, &green4);

// Looper
Looper looper(&recPlayButton, &startStopButton, &resetButton, &record_led, &
    play_led);

int main()
{
    printf("Starting␣setup...\n");

    // Add tracks to the looper
    looper.addTrack(&track1Controller);
    looper.addTrack(&track2Controller);
    looper.addTrack(&track3Controller);
    looper.addTrack(&track4Controller);

    // Initialize the audio and LED interfaces
    audio_init();
    LedInterface_init();

    // Set up a struct to sleep between state machine ticks
    struct timespec sleep_time, time2;
    sleep_time.tv_sec = 0;
    sleep_time.tv_nsec = TWO_MS;

    rollover = false; // make sure rollover is initialized (to false)

    //turn on the record LED to signify that setup is complete!
    record_led.turnOn();
    printf("Setup␣complete!\n");

    while (1)
    {
        // Run the state machine, then sleep for ~2ms. The timing here is very
            soft.
        // As long as we're running the state machine every few ms (or somewhat
        // close to it), we'll be able to detect button presses and update our
            state
```

```
        // machines fast enough for acceptable performance. The audio handling
            is
        // all done in a separate, high-priority, real-time thread, so there is
        // nothing critical depending on a tight 2ms timing. Extra delays will
            occur
        // occasionaly as the OS is running other tasks, but this is not
            noticable
        // to the user.
        looper.tick();
        nanosleep(&sleep_time, &time2);
    }

    return 0;
}
```

13 Makefile

```
looper: *.cpp
        g++ *.cpp -Wall -lrt -lasound -lpthread -lportaudio -lbcm2835 -o looper
            -O3

clean:
        rm looper
```

14 tlc59711/hal_spi.h

```
/**
 * @file hal_spi.h
 *
 * This file was found on github in a library provided by Arjan van Vught. It
 * is a header file that enables SPI on the raspberry pi zero. Source code here:
 * https://github.com/vanvught/rpidmx512/blob/master/lib-hal/include/hal_spi.h
 *
 */


/* Copyright (C) 2019 by Arjan van Vught mailto:info@raspberrypi-dmx.nl
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:

 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.

 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */

#ifndef HAL_SPI_H_
#define HAL_SPI_H_

#if defined(__linux__)
#include "bcm2835.h"
#elif defined(H3)
#include "h3_spi.h"
#else
#include "bcm2835_spi.h"
#endif

#if defined(H3)
#define SPI_BIT_ORDER_MSBFIRST  H3_SPI_BIT_ORDER_MSBFIRST
#define SPI_MODE0                               H3_SPI_MODE0
#define SPI_MODE3                               H3_SPI_MODE3
#define SPI_CS0                         H3_SPI_CS0
#define SPI_CS_NONE                     H3_SPI_CS_NONE
#else
#define SPI_BIT_ORDER_MSBFIRST  BCM2835_SPI_BIT_ORDER_MSBFIRST
#define SPI_MODE0                               BCM2835_SPI_MODE0
#define SPI_MODE3                               BCM2835_SPI_MODE3
#define SPI_CS0                         BCM2835_SPI_CS0
#define SPI_CS_NONE                     BCM2835_SPI_CS_NONE
#endif
```

```
#if defined(H3)
#define FUNC_PREFIX(x) h3_##x
#else
#define FUNC_PREFIX(x) bcm2835_##x
#endif

#endif /* HAL_SPI_H_ */
```

15 tlc59711/LedDriver.cpp

```
/**
 * @file LedDriver.cpp
 *
 * @brief LED driver
 *
 * This file updates the LEDs based on what is stored in shared memory
 *
 * @author Bryan Cisneros
 */

#include "LedDriver.h"
#include "tlc59711.h"
#include "bcm2835.h"
#include <sys/shm.h>
#include <stdio.h>
#include <string.h>

#define ON (0xFF)
#define OFF (0x00)
#define BRIGHTNESS (0x30)

typedef volatile struct
{
    bool update;
    uint8_t led_values[12];
} shared_leds_t;

void* shared_pointer = NULL;
shared_leds_t* shared_leds;
int shared_leds_id;

uint8_t local_led_values[12];

TLC59711* leds;

void LedDriver_init(void)
{
    bcm2835_init();

    leds = new TLC59711;

    // Set the brightness of the LEDs
    leds->SetGbcRed(BRIGHTNESS);
    leds->SetGbcGreen(BRIGHTNESS);
    leds->SetGbcBlue(BRIGHTNESS);

    // Turn off all LEDs
    for (int i = 0; i < TLC59711_OUT_CHANNELS; i++)
    {
        leds->Set(i, (uint8_t)OFF);
    }
    leds->Update();

    // Get or create shared memory
    shared_leds_id= shmget((key_t)1234, sizeof(shared_leds_t), 0666 | IPC_CREAT)
```

```
        ;
    if (shared_leds_id == -1)
    {
        printf("shmget() failed!\n");
    }

    shared_pointer = shmat(shared_leds_id, (void*)0, 0);
    if (shared_pointer == (void*)-1)
    {
        printf("shmat() failed!\n");
    }

    shared_leds = (shared_leds_t*)shared_pointer;

    // Initialize shared memory. Start with all the LEDs off and the update flag
         false
    shared_leds->update = false;
    for (int i = 0; i < TLC59711_OUT_CHANNELS; i++)
    {
        shared_leds->led_values[i] = 0;
    }
}

void LedDriver_checkForUpdates(void)
{
    if (shared_leds->update)
    {
        // Quickly copy the values into a local array.
        // This will minimize the time for a potential shared data problem
        memcpy(local_led_values, (void*)(shared_leds->led_values), 12);
        shared_leds->update = false;

        // Update the LEDs with the new values
        for (int i = 0; i < TLC59711_OUT_CHANNELS; i++)
        {
            leds->Set(i, local_led_values[i]);
            printf("%d, ", local_led_values[i]);
        }
        printf("\n");
        leds->Update();
    }
}
```

16 tlc59711/LedDriver.h

```
/**
 * @file LedDriver.h
 *
 * @brief LED driver
 *
 * This file contains the API to update the LEDs
 *
 * @author Bryan Cisneros
 */

#pragma once

/**
 * @brief Initialize the LED driver
 *
 * @return void
 */
void LedDriver_init(void);

/**
 * @brief Check for updates
 *
 * This function checks the shared memory for any updates. If there is an update
    ,
 * the LED driver then updates the LEDs with the new values.
 *
 * @return void
 */
void LedDriver_checkForUpdates(void);
```

17 tlc59711/main.cpp

```
/**
 * @file main.cpp
 *
 * @brief main for LED driver program
 *
 * This file initializes the LED driver, then enters a while(1) loop to
 * periodically check/update the LEDs
 *
 * @author Bryan Cisneros
 */

#include "LedDriver.h"
#include <time.h>
#include <stdio.h>

#define FIFTEEN_MS (15000000)

int main(void)
{
    // Set up a struct to be able to sleep for 15 ms
    struct timespec sleep_time, time2;
    sleep_time.tv_sec = 0;
    sleep_time.tv_nsec = FIFTEEN_MS;

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

    LedDriver_init();

    printf("Done!\n");

    while(1)
    {
        // Check for updates, then sleep for ~15 ms
        LedDriver_checkForUpdates();
        nanosleep(&sleep_time, &time2);
    }
}
```

18 tlc59711/Makefile

```
led_driver: main.cpp LedDriver.cpp tlc59711.cpp
	g++ *.cpp -DNDEBUG -Wall -o led_driver -lbcm2835

clean:
	rm led_driver
```

19 tlc59711/tlc59711.cpp

```
/**
 * @file tlc59711.cpp
 *
 * This file was found on github in a library provided by Arjan van Vught. It
 * provides an interface to interact with the TLC59711 LED driver. A link to the
 * repository is here:
 * https://github.com/vanvught/rpidmx512/tree/master/lib-tlc59711
 *
 */


/* Copyright (C) 2018-2019 by Arjan van Vught mailto:info@raspberrypi-dmx.nl
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:

 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.

 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */

#include <stdint.h>
#if !defined(NDEBUG) || defined(__linux__)
#include <stdio.h>
#endif
#include <string.h>
#include <assert.h>

#include "tlc59711.h"

#include "hal_spi.h"

#define TLC59711_COMMAND                        0x25
#define TLC59711_COMMAND_SHIFT  26

#define TLC59711_OUTTMG_DEFAULT         0
#define TLC59711_OUTTMG_SHIFT   25

#define TLC59711_EXTGCK_DEFAULT         0
#define TLC59711_EXTGCK_SHIFT   24

#define TLC59711_TMGRST_DEFAULT         1
#define TLC59711_TMGRST_SHIFT   23
```

```
#define TLC59711_DSPRPT_DEFAULT         1
#define TLC59711_DSPRPT_SHIFT   22

#define TLC59711_BLANK_DEFAULT          0
#define TLC59711_BLANK_SHIFT    21

#define TLC59711_GS_DEFAULT                     0x7F
#define TLC59711_GS_MASK                0x7F
#define TLC59711_GS_RED_SHIFT   0
#define TLC59711_GS_GREEN_SHIFT 7
#define TLC59711_GS_BLUE_SHIFT  14

TLC59711::TLC59711(uint8_t nBoards, uint32_t nSpiSpeedHz):
    m_nBoards(nBoards),
    m_nSpiSpeedHz(nSpiSpeedHz == 0 ? TLC59711_SPI_SPEED_DEFAULT : nSpiSpeedHz),
    m_nFirst32(0),
    m_pBuffer(0),
    m_pBufferBlackout(0),
    m_nBufSize(0)
{
    FUNC_PREFIX(spi_begin());

    if (m_nSpiSpeedHz > TLC59711_SPI_SPEED_MAX)
    {
        m_nSpiSpeedHz = TLC59711_SPI_SPEED_MAX;
    }

    if (nBoards == 0)
    {
        nBoards = 1;
    }

    m_nBufSize = nBoards * TLC59711_16BIT_CHANNELS;

    m_pBuffer = new uint16_t[m_nBufSize];
    assert(m_pBuffer != 0);

    m_pBufferBlackout = new uint16_t[m_nBufSize];
    assert(m_pBufferBlackout != 0);

    for (uint32_t i = 0; i < m_nBufSize; i++)
    {
        m_pBuffer[i] = (uint16_t) 0;
    }

    m_nFirst32 |= (uint32_t) TLC59711_COMMAND << TLC59711_COMMAND_SHIFT ;

    SetOnOffTiming(TLC59711_OUTTMG_DEFAULT);
    SetExternalClock(TLC59711_EXTGCK_DEFAULT);
    SetDisplayTimingReset(TLC59711_TMGRST_DEFAULT);
    SetDisplayRepeat(TLC59711_DSPRPT_DEFAULT);
    SetBlank(TLC59711_BLANK_DEFAULT);
    SetGbcRed(TLC59711_GS_DEFAULT);
    SetGbcGreen(TLC59711_GS_DEFAULT);
    SetGbcBlue(TLC59711_GS_DEFAULT);

    memcpy(m_pBufferBlackout, m_pBuffer, m_nBufSize * 2);
```

```
}

TLC59711::~TLC59711(void)
{
    delete[] m_pBuffer;
    m_pBuffer = 0;
}

bool TLC59711::Get(uint8_t nChannel, uint16_t &nValue)
{
    const uint8_t nBoardIndex = nChannel / TLC59711_OUT_CHANNELS;

    if (nBoardIndex < m_nBoards)
    {
        const uint32_t nIndex = 2 + (nBoardIndex * TLC59711_16BIT_CHANNELS) +
            ((12 * nBoardIndex) + 11 - nChannel);
        nValue =  __builtin_bswap16(m_pBuffer[nIndex]);
        return true;
    }

    return false;
}

void TLC59711::Set(uint8_t nChannel, uint16_t nValue)
{
    const uint8_t nBoardIndex = nChannel / TLC59711_OUT_CHANNELS;

    if (nBoardIndex < m_nBoards)
    {
        const uint32_t nIndex = 2 + (nBoardIndex * TLC59711_16BIT_CHANNELS) +
            ((12 * nBoardIndex) + 11 - nChannel);
        m_pBuffer[nIndex] = __builtin_bswap16(nValue);
    }
#ifndef NDEBUG
    else
    {
        printf("\t\tm_nBoards=%d, nBoardIndex=%d, nChannel=%d\n", (int)
            m_nBoards, (int) nBoardIndex, (int) nChannel);
    }
#endif
}

bool TLC59711::GetRgb(uint8_t nOut, uint16_t& nRed, uint16_t& nGreen, uint16_t&
    nBlue)
{
    const uint8_t nBoardIndex = nOut / 4;

    if (nBoardIndex < m_nBoards)
    {
        uint32_t nIndex = 2 + (nBoardIndex * TLC59711_16BIT_CHANNELS) + (((4 *
            nBoardIndex) + 3 - nOut) * 3);
        nBlue = __builtin_bswap16(m_pBuffer[nIndex++]);
        nGreen = __builtin_bswap16(m_pBuffer[nIndex++]);
        nRed = __builtin_bswap16(m_pBuffer[nIndex]);
        return true;
    }
```

```
    return false;
}

void TLC59711::Set(uint8_t nChannel, uint8_t nValue)
{
    const uint8_t nBoardIndex = nChannel / TLC59711_OUT_CHANNELS;

    if (nBoardIndex < m_nBoards)
    {
        const uint32_t nIndex = 2 + (nBoardIndex * TLC59711_16BIT_CHANNELS) +
            ((12 * nBoardIndex) + 11 - nChannel);
        m_pBuffer[nIndex] = (uint16_t) nValue << 8 | (uint16_t) nValue;
    }
#ifndef NDEBUG
    else
    {
        printf("\t\tm_nBoards=%d, nBoardIndex=%d, nChannel=%d\n", (int)
            m_nBoards, (int) nBoardIndex, (int) nChannel);
    }
#endif
}

void TLC59711::SetRgb(uint8_t nOut, uint16_t nRed, uint16_t nGreen, uint16_t
    nBlue)
{
    const uint8_t nBoardIndex = nOut / 4;

    if (nBoardIndex < m_nBoards)
    {
        uint32_t nIndex = 2 + (nBoardIndex * TLC59711_16BIT_CHANNELS) + (((4 *
            nBoardIndex) +3 - nOut) * 3);
        m_pBuffer[nIndex++] = __builtin_bswap16(nBlue);
        m_pBuffer[nIndex++] = __builtin_bswap16(nGreen);
        m_pBuffer[nIndex] = __builtin_bswap16(nRed);
    }
#ifndef NDEBUG
    else
    {
        printf("m_nBoards=%d, nBoardIndex=%d, nOut=%d\n", (int) m_nBoards, (int)
             nBoardIndex, (int) nOut);
    }
#endif
}

void TLC59711::SetRgb(uint8_t nOut, uint8_t nRed, uint8_t nGreen, uint8_t nBlue)
{
    const uint8_t nBoardIndex = nOut / 4;

    if (nBoardIndex < m_nBoards)
    {
        uint32_t nIndex = 2 + (nBoardIndex * TLC59711_16BIT_CHANNELS) + (((4 *
            nBoardIndex) + 3 - nOut) * 3);
        m_pBuffer[nIndex++] = (uint16_t) nBlue << 8 | (uint16_t) nBlue;
        m_pBuffer[nIndex++] = (uint16_t) nGreen << 8 | (uint16_t) nGreen;
        m_pBuffer[nIndex] = (uint16_t) nRed << 8 | (uint16_t) nRed;
    }
#ifndef NDEBUG
```

```
    else
    {
        printf("m_nBoards=%d, nBoardIndex=%d, nOut=%d\n", (int) m_nBoards, (int)
            nBoardIndex, (int) nOut);
    }
#endif
}

int TLC59711::GetBlank(void) const
{
    return (int)(m_nFirst32 & ((uint32_t) 1 << TLC59711_BLANK_SHIFT)) == (
        uint32_t) 1 << TLC59711_BLANK_SHIFT;
}

void TLC59711::SetBlank(bool pBlank)
{
    m_nFirst32 &= ~((uint32_t) 1 << TLC59711_BLANK_SHIFT);

    if (pBlank)
    {
        m_nFirst32 |= (uint32_t) 1 << TLC59711_BLANK_SHIFT;
    }

    UpdateFirst32();
}

int TLC59711::GetDisplayRepeat(void) const
{
    return (int)(m_nFirst32 & ((uint32_t) 1 << TLC59711_DSPRPT_SHIFT)) == (
        uint32_t) 1 << TLC59711_DSPRPT_SHIFT;
}

void TLC59711::SetDisplayRepeat(bool pDisplayRepeat)
{
    m_nFirst32 &= ~((uint32_t) 1 << TLC59711_DSPRPT_SHIFT);

    if (pDisplayRepeat)
    {
        m_nFirst32 |= (uint32_t) 1 << TLC59711_DSPRPT_SHIFT;
    }

    UpdateFirst32();
}

int TLC59711::GetDisplayTimingReset(void) const
{
    return (int)(m_nFirst32 & ((uint32_t) 1 << TLC59711_TMGRST_SHIFT)) == (
        uint32_t) 1 << TLC59711_TMGRST_SHIFT;
}

void TLC59711::SetDisplayTimingReset(bool pDisplayTimingReset)
{
    m_nFirst32 &= ~((uint32_t) 1 << TLC59711_TMGRST_SHIFT);

    if (pDisplayTimingReset)
    {
        m_nFirst32 |= (uint32_t) 1 << TLC59711_TMGRST_SHIFT;
```

```
    }

    UpdateFirst32();
}

int TLC59711::GetExternalClock(void) const
{
    return (int)(m_nFirst32 & ((uint32_t) 1 << TLC59711_EXTGCK_SHIFT)) == (
        uint32_t) 1 << TLC59711_EXTGCK_SHIFT;
}

void TLC59711::SetExternalClock(bool pExternalClock)
{
    m_nFirst32 &= ~((uint32_t) 1 << TLC59711_EXTGCK_SHIFT);

    if (pExternalClock)
    {
        m_nFirst32 |= (uint32_t) 1 << TLC59711_EXTGCK_SHIFT;
    }

    UpdateFirst32();
}

int TLC59711::GetOnOffTiming(void) const
{
    return (int)(m_nFirst32 & ((uint32_t) 1 << TLC59711_OUTTMG_SHIFT)) == (
        uint32_t) 1 << TLC59711_OUTTMG_SHIFT;
}

void TLC59711::SetOnOffTiming(bool pOnOffTiming)
{
    m_nFirst32 &= ~((uint32_t) 1 << TLC59711_OUTTMG_SHIFT);

    if (pOnOffTiming)
    {
        m_nFirst32 |= (uint32_t) 1 << TLC59711_OUTTMG_SHIFT;
    }

    UpdateFirst32();
}

uint8_t TLC59711::GetGbcRed(void) const
{
    return (uint8_t) (m_nFirst32 >> TLC59711_GS_RED_SHIFT) & TLC59711_GS_MASK;
}

void TLC59711::SetGbcRed(uint8_t nValue)
{
    m_nFirst32 &= ~((uint32_t) TLC59711_GS_MASK << TLC59711_GS_RED_SHIFT);
    m_nFirst32 |= (uint32_t)(nValue & TLC59711_GS_MASK) << TLC59711_GS_RED_SHIFT
        ;

    UpdateFirst32();
}

uint8_t TLC59711::GetGbcGreen(void) const
{
```

```
    return (uint8_t) (m_nFirst32 >> TLC59711_GS_GREEN_SHIFT) & TLC59711_GS_MASK;
}

void TLC59711::SetGbcGreen(uint8_t nValue)
{
    m_nFirst32 &= ~((uint32_t) TLC59711_GS_MASK << TLC59711_GS_GREEN_SHIFT);
    m_nFirst32 |= (uint32_t)(nValue & TLC59711_GS_MASK) <<
        TLC59711_GS_GREEN_SHIFT;

    UpdateFirst32();
}

uint8_t TLC59711::GetGbcBlue(void) const
{
    return (uint8_t) (m_nFirst32 >> TLC59711_GS_BLUE_SHIFT) & TLC59711_GS_MASK;
}

void TLC59711::SetGbcBlue(uint8_t nValue)
{
    m_nFirst32 &= ~((uint32_t) TLC59711_GS_MASK << TLC59711_GS_BLUE_SHIFT);
    m_nFirst32 |= (uint32_t)(nValue & TLC59711_GS_MASK) <<
        TLC59711_GS_BLUE_SHIFT;

    UpdateFirst32();
}

void TLC59711::UpdateFirst32(void)
{
    for (uint32_t i = 0; i < m_nBoards; i++)
    {
        const uint32_t nIndex = TLC59711_16BIT_CHANNELS * i;
        m_pBuffer[nIndex] = __builtin_bswap16((uint16_t) (m_nFirst32 >> 16));
        m_pBuffer[nIndex + 1] = __builtin_bswap16((uint16_t) m_nFirst32);
    }
}

void TLC59711::Dump(void)
{
#ifndef NDEBUG
    printf("Command:0x%.2X\n", m_nFirst32 >> TLC59711_COMMAND_SHIFT);
    printf("\tOUTTMG:%d␣(default=%d)\n", GetOnOffTiming(),
        TLC59711_OUTTMG_DEFAULT);
    printf("\tEXTGCK:%d␣(default=%d)\n", GetExternalClock(),
        TLC59711_EXTGCK_DEFAULT);
    printf("\tTMGRST:%d␣(default=%d)\n", GetDisplayTimingReset(),
        TLC59711_TMGRST_DEFAULT);
    printf("\tDSPRPT:%d␣(default=%d)\n", GetDisplayRepeat(),
        TLC59711_DSPRPT_DEFAULT);
    printf("\tBLANK:%d␣␣(default=%d)\n", GetBlank(),  TLC59711_BLANK_DEFAULT);
    printf("\nGlobal␣Brightness\n");
    printf("\tRed:0x%.2X␣(default=0x%.2X)\n", GetGbcRed(), TLC59711_GS_DEFAULT);
    printf("\tGreen:0x%.2X␣(default=0x%.2X)\n", GetGbcGreen(),
        TLC59711_GS_DEFAULT);
    printf("\tBlue:0x%.2X␣(default=0x%.2X)\n", GetGbcBlue(), TLC59711_GS_DEFAULT
        );
    printf("\nBoards:%d\n", (int) m_nBoards);
```

```
    uint8_t nOut = 0;

    for (uint32_t i = 0; i < m_nBoards; i++)
    {
        for (uint32_t j = 0; j < TLC59711_RGB_CHANNELS; j ++)
        {
            uint16_t nRed = 0, nGreen = 0, nBlue = 0;
            if (GetRgb(nOut, nRed, nGreen, nBlue))
            {
                printf("\tOut:%-2d, Red=0x%.4X, Green=0x%.4X, Blue=0x%.4X\n",
                    nOut, nRed, nGreen, nBlue);
            }
            nOut++;
        }
    }

    printf("\n");

    for (uint32_t i = 0; i < m_nBoards * TLC59711_OUT_CHANNELS; i++)
    {
        uint16_t nValue = 0;
        if (Get((uint8_t) i, nValue))
        {
            printf("\tChannel:%-3d, Value=0x%.4X\n", (int) i, nValue);
        }
    }

    printf("\n");
#endif
}

void TLC59711::Update(void)
{
    assert(m_pBuffer != 0);

    FUNC_PREFIX(spi_chipSelect(SPI_CS_NONE));
    FUNC_PREFIX(spi_set_speed_hz(m_nSpiSpeedHz));
    FUNC_PREFIX(spi_setDataMode(SPI_MODE0));
    FUNC_PREFIX(spi_writenb((char *) m_pBuffer, m_nBufSize * 2));
}

void TLC59711::Blackout(void)
{
    assert(m_pBufferBlackout != 0);

    FUNC_PREFIX(spi_chipSelect(SPI_CS_NONE));
    FUNC_PREFIX(spi_set_speed_hz(m_nSpiSpeedHz));
    FUNC_PREFIX(spi_setDataMode(SPI_MODE0));
    FUNC_PREFIX(spi_writenb((char *) m_pBufferBlackout, m_nBufSize * 2));
}
```

20 tlc59711/tlc59711.h

```
/**
 * @file tlc59711.h
 *
 * This file was found on github in a library provided by Arjan van Vught. It
 * provides an interface to interact with the TLC59711 LED driver. A link to the
 * repository is here:
 * https://github.com/vanvught/rpidmx512/tree/master/lib-tlc59711
 *
 */


/* Copyright (C) 2018-2019 by Arjan van Vught mailto:info@raspberrypi-dmx.nl
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:

 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.

 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */

#include <stdint.h>

#ifndef TLC59711_H_
#define TLC59711_H_

#define TLC59711_SPI_SPEED_DEFAULT      5000000
#define TLC59711_SPI_SPEED_MAX          10000000

#define TLC59711_16BIT_CHANNELS 14
#define TLC59711_OUT_CHANNELS   12
#define TLC59711_RGB_CHANNELS   4

#define TLC59711_RGB_8BIT_VALUE(x)      ((uint8_t)(x))
#define TLC59711_RGB_16BIT_VALUE(x)     ((uint16_t)(x))

#include <stdint.h>

class TLC59711
{
public:
    TLC59711(uint8_t nBoards = 1, uint32_t nSpiSpeedHz =
        TLC59711_SPI_SPEED_DEFAULT);
    ~TLC59711(void);
```

```
    int GetBlank(void) const;
    void SetBlank(bool pBlank = false);

    int GetDisplayRepeat(void) const;
    void SetDisplayRepeat(bool pDisplayRepeat = true);

    int GetDisplayTimingReset(void) const;
    void SetDisplayTimingReset(bool pDisplayTimingReset = true);

    int GetExternalClock(void) const;
    void SetExternalClock(bool pExternalClock = false);

    int GetOnOffTiming(void) const;
    void SetOnOffTiming(bool pOnOffTiming = false);

    uint8_t GetGbcRed(void) const;
    void SetGbcRed(uint8_t nValue = 0x7F);

    uint8_t GetGbcGreen(void) const;
    void SetGbcGreen(uint8_t nValue = 0x7F);

    uint8_t GetGbcBlue(void) const;
    void SetGbcBlue(uint8_t nValue = 0x7F);

    bool Get(uint8_t nChannel, uint16_t &nValue);
    void Set(uint8_t nChannel, uint16_t nValue);

    void Set(uint8_t nChannel, uint8_t nValue);

    bool GetRgb(uint8_t nOut, uint16_t &nRed, uint16_t &nGreen, uint16_t &nBlue)
        ;
    void SetRgb(uint8_t nOut, uint16_t nRed, uint16_t nGreen, uint16_t nBlue);

    void SetRgb(uint8_t nOut, uint8_t nRed, uint8_t nGreen, uint8_t nBlue);

    void Update(void);
    void Blackout(void);

    void Dump(void);

private:
    void UpdateFirst32(void);

private:
    uint8_t m_nBoards;
    uint32_t m_nSpiSpeedHz;
    uint32_t m_nFirst32;
    uint16_t *m_pBuffer;
    uint16_t *m_pBufferBlackout;
    uint32_t m_nBufSize;
};

#endif /* TLC59711_H_ */
```

21 TrackController.cpp

```
/**
 * @file TrackController.cpp
 *
 * @brief TrackController class
 *
 * This file implements the TrackController class. This class handles
 * everything related to the track, meaning the track itself, the button
    associated
 * with the track, and the LEDs related to the track.
 *
 * @author Bryan Cisneros
 */

#include "TrackController.h"
#include "Globals.h"
#include "AudioInterface.h"
#include <stdio.h>

TrackController::TrackController(Track* track, Button* button, Led* red, Led*
    green)
{
    // Initialize member variables with inputs
    this->track = track;
    this->button = button;
    led_red = red;
    led_green = green;

    // Initialize to idle
    state = idle;
    waiting_to_play = false;
    waiting_to_stop = false;

    // Add the track to the audio interface
    audio_add_track(track);
}


TrackController::~TrackController()
{
}

void TrackController::tick()
{
    // Run the button state machine
    button->tick();
    bool buttonPressed = button->fell();

    // Run the LED state machines
    led_red->tick();
    led_green->tick();

    //state action
    switch (state)
    {
    case TrackController::idle:
```

```
    break;
case TrackController::recording:
    break;
case TrackController::playing:
    if (buttonPressed && !recordingMode)
    {
        // If we've pressed the button and we're in playing mode, we want to
        // stop playing the track at the next wrap around. Start flashing
        // the LED and set the flag
        led_green->flash();
        waiting_to_stop = true;
    }
    break;
case TrackController::waiting:
    if (buttonPressed && !recordingMode)
    {
        // If we've pressed the button and we're in playing mode, we want to
        // start playing the track at the next wrap around. Start flashing
        // the LED and set the flag
        led_green->flash();
        waiting_to_play = true;
    }
    break;
default:
    break;
}


//state update
switch (state)
{
case TrackController::idle:
    if (buttonPressed && (waitingToStart == 0))
    {
        // start recording (and make sure the audio starts at position 0)
        audio_set_track_position(0);
        track->startRecording();
        track->startPlaying(); // this will be empty audio for now...

        // Turn on both LEDs. Technically we are playing audio too, but it's
             empty
        led_red->turnOn();
        led_green->turnOn();

        // Increment waiting to start variable. This will let everyone know
        // that we are in the first recording state
        waitingToStart++;

        // move to recording state
        state = recording;
    }
    else if(buttonPressed && recordingMode)
    {
        track->startRecording(); //start recording
        track->startPlaying(); // if we weren't playing, start playing too

        // Turn both LEDs on
```

```
            led_red->turnOn();
            led_green->turnOn();

            // move to recording state
            state = recording;
        }
        break;
    case TrackController::recording:
        if (buttonPressed)
        {
            //stop recording and turn off the red LED
            track->stopRecording();
            led_red->turnOff();

            // If this was the first recording, set the length of the track
            // and increment waiting to start variable
            if (waitingToStart == 1)
            {
                audio_set_track_length();
                waitingToStart++;
            }

            // move to playing state
            state = playing;
        }
        break;
    case TrackController::playing:
        if (waiting_to_stop && masterDone)
        {
            // stop playing and turn off green LED
            track->stopPlaying();
            led_green->turnOff();

            // move to waiting state and clear flag
            state = waiting;
            waiting_to_stop = false;
        }
        if (buttonPressed && recordingMode)
        {
            // If we've pressed the button and are in recording mode, we don't
            // need to wait for a wrap around. Start recording immediately
            track->startRecording();
            led_red->turnOn();

            // move to recording state
            state = recording;
        }
        break;
    case TrackController::waiting:
        if (waiting_to_play && masterDone)
        {
            // If we were waiting to start playing and the wrap around happened,
            // start playing and turn on the green LED
            track->startPlaying();
            led_green->turnOn();

            // move to playing state and clear flag
```

```
            state = playing;
            waiting_to_play = false;
        }
        if (buttonPressed && recordingMode)
        {
            // If we've pressed the button and are in recording mode, we don't
            // need to wait for a wrap around. Start recording (and also playing
                )
            // immediately. Also turn on both LEDs
            track->startPlaying();
            track->startRecording();
            led_red->turnOn();
            led_green->turnOn();

            // move to recording state
            state = recording;
        }
        break;
    default:
        break;
    }
}

TrackController::State TrackController::getState()
{
    return state;
}

void TrackController::stopButton()
{
    // store the last state so we can come back to it
    lastState = state;

    // based on what the state was, stop recording/playing and turn off the LEDs
    switch (state)
    {
    case idle:
        break;
    case recording:
        track->stopRecording();
        track->stopPlaying();
        led_red->turnOff();
        led_green->turnOff();
        break;
    case playing:
        track->stopPlaying();
        led_green->turnOff();
        break;
    case waiting:
        break;
    default:
        break;
    }

    // clear flags
    waiting_to_play = false;
    waiting_to_stop = false;
```

```
    // move to stopped state
    state = stopped;
}

void TrackController::startButton()
{
    // Restore the saved state, and start playing again if necessary. Note that
    // if we were previously recording, we don't keep recording after a stop
    switch (lastState)
    {
    case idle:
        state = idle;
        break;
    case recording:
        state = idle;
        break;
    case playing:
        track->startPlaying();
        led_green->turnOn();
        state = playing;
        break;
    case waiting:
        state = waiting;
        break;
    default:
        state = idle;
        printf("Hit default case in TrackController::startButton()\n");
        break;
    }
}

void TrackController::resetButton()
{
    // based on what state we were in, stop playing/recording and turn off LEDs
    switch (state)
    {
    case idle:
        break;
    case recording:
        track->stopRecording();
        track->stopPlaying();
        led_red->turnOff();
        led_green->turnOff();
        break;
    case playing:
        track->stopPlaying();
        led_green->turnOff();
        break;
    case waiting:
        break;
    default:
        break;
    }

    // clear flags
    waiting_to_play = false;
```

```
    waiting_to_stop = false;

    // move to the idle state
    state = idle;
}
```

22 TrackController.h

```
/**
 * @file TrackController.h
 *
 * @brief TrackController class
 *
 * This file implements the API for the TrackController class. This class
    handles
 * everything related to the track, meaning the track itself, the button
    associated
 * with the track, and the LEDs related to the track.
 *
 * @author Bryan Cisneros
 */

#pragma once
#include "Track.h"
#include "Button.h"
#include "Led.h"

class TrackController
{
public:
    /**
     * @brief TrackController constructor
     *
     * @param track track object
     * @param button button associated with track
     * @param red red LED (used when recording)
     * @param green green LED (used when playing)
     *
     * @return void
     */
    TrackController(Track* track, Button* button, Led* red, Led* green);

    /**
     * @brief TrackController destructor
     *
     * @return void
     */
    ~TrackController();

    // States for the state machine
    enum State { idle, recording, playing, waiting, stopped };

    /**
     * @brief Tick
     *
     * This function runs the state machine, as well as all of the sub state
     * machines (for the button, and leds). It should be called frequently for
     * proper function.
     *
     * @return void
     */
    void tick();
```

```
    /**
     * @brief get the current state of the track controller
     *
     * @return the state of the track controller
     */
    State getState();

    /**
     * @brief handle the main stop button being pressed
     *
     * @return void
     */
    void stopButton();

    /**
     * @brief handle the main start button being pressed
     *
     * @return void
     */
    void startButton();

    /**
     * @brief handle the main reset button being pressed
     *
     * @return void
     */
    void resetButton();

private:
    State state; // Current state
    State lastState; // Remembers the last state on a stop
    Track* track; // keeps track of recording/playing
    Button* button; // input to the state machine
    Led* led_red; // turns on when recording
    Led* led_green; // turns on when playing

    // These variables are used when we are going to start or stop playing
    // the track, but we're waiting for a rollover to occur
    bool waiting_to_play;
    bool waiting_to_stop;
};
```

23 Track.cpp

```cpp
/**
 * @file Track.cpp
 *
 * @brief Track class
 *
 * This file implements the Track class. The Track class simply keeps track of
 * if the track is currently playing, recording, or both (or neither).
 *
 * @author Bryan Cisneros
 */

#include "Track.h"

Track::Track()
{
    // Initialize both playing and recording to false.
    playing = false;
    recording = false;
}

Track::~Track()
{
}

bool Track::isPlaying()
{
    return playing;
}

bool Track::isRecording()
{
    return recording;
}

void Track::startPlaying()
{
    playing = true;
}

void Track::startRecording()
{
    recording = true;
}

void Track::stopPlaying()
{
    playing = false;
}

void Track::stopRecording()
{
    recording = false;
}
```

24 Track.h

```
/**
 * @file Track.h
 *
 * @brief Track class
 *
 * This file implements the API for the Track class. The Track class simply
 * keeps track of if the track is currently playing, recording, or both (or
 * neither).
 *
 * @author Bryan Cisneros
 */

#pragma once

class Track
{
public:
    /**
     * @brief Track constructor
     *
     * @return void
     */
    Track();

    /**
     * @brief Track destructor
     *
     * @return void
     */
    ~Track();

    /**
     * @brief is track playing?
     *
     * @return true if playing, false otherwise
     */
    bool isPlaying();

    /**
     * @brief is track recording?
     *
     * @return true if recording, false otherwise
     */
    bool isRecording();

    /**
     * @brief start playing the track
     *
     * @return void
     */
    void startPlaying();

    /**
     * @brief start recording the track
     *
```

```
     * @return void
     */
    void startRecording();

    /**
     * @brief stop playing the track
     *
     * @return void
     */
    void stopPlaying();

    /**
     * @brief stop recording the track
     *
     * @return void
     */
    void stopRecording();

private:
    // variables to store playing and recording state
    bool playing;
    bool recording;
};
```