MIDI Pattern Visualizer

Final Project Report
December 14, 2018
E155: Microprocessor Systems

Vicki Moran
Spencer Rosen

Abstract:

Making music brings joy to many people, uniting individuals under a common desire to create something beautiful. Adding lights makes the experience even more enjoyable and visually pleasing. This project creates a system that plays and displays a pattern of notes recorded from a MIDI keyboard using a Raspberry Pi and an Altera Cyclone IV FPGA. The Pi receives user input from the keyboard and outputs the corresponding frequency to a speaker. Simultaneously, the Pi sends the note across SPI to the FPGA, which saves these values and interfaces to an LED matrix. By storing previous note values, the FPGA displays a scrolling pattern with various colors.

Introduction

People often experience music through hearing alone, limiting the experience to a single sense. Every note on a keyboard corresponds to a different frequency, just as colors do, so relating colors with sound can create a full audiovisual experience. Furthermore, fun and interactive projects make people happy, which is motivation enough for this MIDI pattern visualizer.

The system receives input from a MIDI keyboard and plays the corresponding note on a speaker while displaying the MIDI visualization of the input on an LED matrix. This involves receiving user input from the MIDI keyboard through the Raspberry Pi, outputting a square wave of appropriate frequency to a speaker, and sending data about which key is pressed or unpressed to the FPGA through SPI. The FPGA controls an LED matrix to display a scrolling effect of the notes pressed over time. A simplified block diagram of the overall system is shown below.

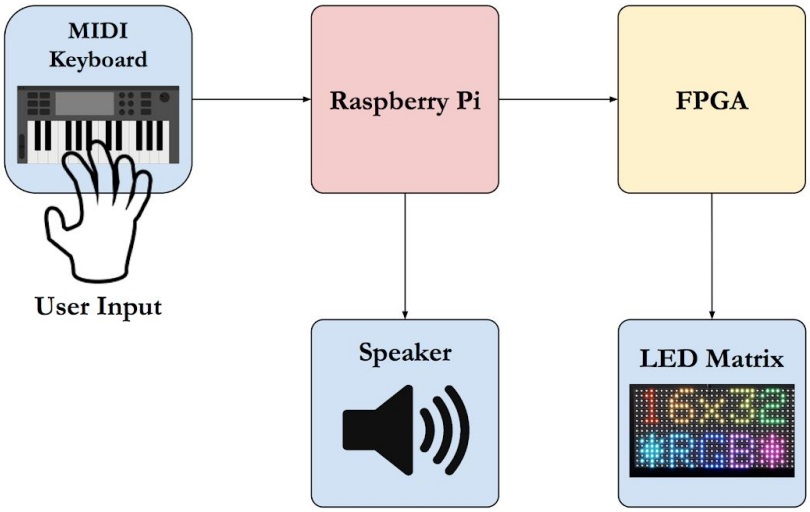

Figure 1: Simple Block Diagram

An additional feature allows the user to toggle between three modes: live mode, recording mode, and playback mode. In live mode, when the user presses a key on the keyboard, the system plays the note through the speaker and displays the corresponding color across the LED matrix. Recording mode has all the same functionality of live mode, with the additional feature of storing the pattern played by the user. In playback mode, the system plays and displays the stored pattern on repeat. A dedicated key on the keyboard starts and stops a recording, and an external push button clears the stored pattern and returns to live mode. A status LED indicates the current mode: off during live mode, blinking during recording mode, and solid during playback mode. The current implementation allows for only one key to be pressed, unpressed, and displayed at any given time.

New Hardware

In addition to hardware previously implemented in E155, this project uses the AKM320 midiplus MIDI controller. The MIDI keyboard consists of 32 keys and other functionality, including a pitch wheel, modulation wheel, octave buttons, and transpose buttons [2]. The keyboard plugs into the Pi via USB, obscuring the more complex aspects of the data transfer. A publicly available online resource, the Advanced Linux Sound Architecture (ALSA) project [4] has several open source libraries that provide support for various audio interfaces connected to the MIDI. This project took advantage of the functionality for reading MIDI input in the provided C library. Specifically, each time a user presses or unpressed a key, the original code prints out three hexadecimal numbers to the terminal: the first representing whether the key is pressed or unpressed, the second representing which key is pressed or unpressed, and the third representing the velocity of the press.

Figure 2: AKM320 MIDI Keyboard [2]

Another new hardware used in this project is the 16x32 RGB LED matrix panel. The panel has 512 RGB LEDs on the front and a PCB with IDC connectors for input and output on the back. With a power requirement of 5V, the panel draws a maximum current of 4A when all LEDs are set to display white. Because the display lacks built in PWM control, using high speed processors such as FPGAs for control generates optimal results [3]. To display patterns, each row receives RGB inputs for the 32 columns within that row. Two additional signals, output enable and latch, determine when the RGB values stored in the shift registers display. The panel is split between the upper eight rows and the lower eight rows, and only one row in each half can be controlled at a time; thus, the display is multiplexed with a ⅛ duty cycle [5]. The FPGA design section of this report specifies the details of implementation.

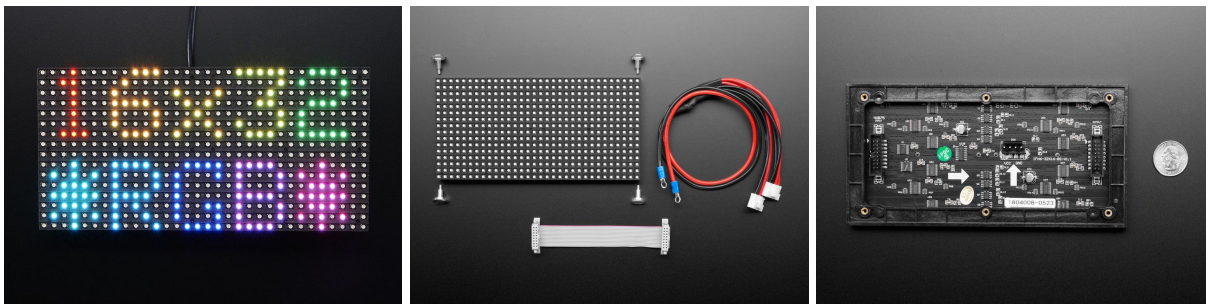

Figure 3: 16x32 RGB LED Matrix [3]

Schematic

The schematic shown below includes of all circuity on the breadboard connecting each subsystem and interfacing to external hardware.

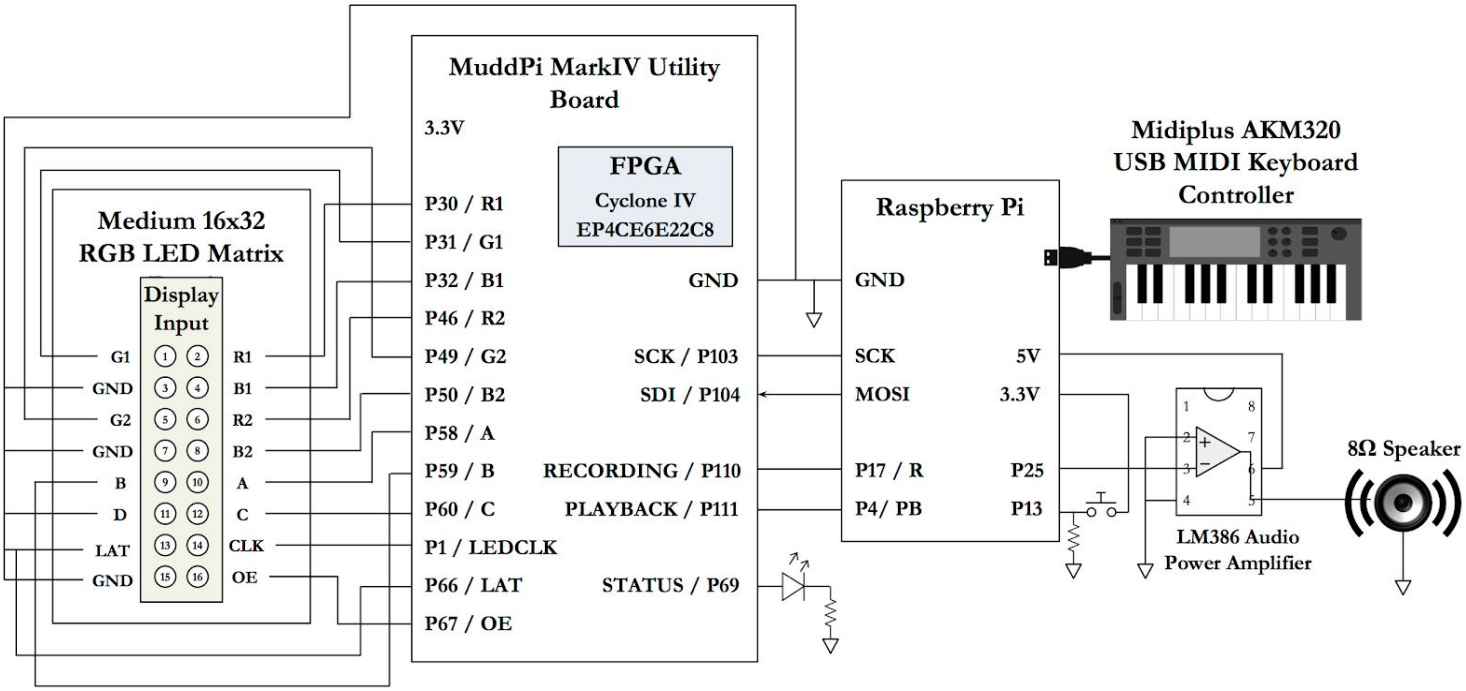

Figure 4: Full Schematic

The Raspberry Pi receives input from the MIDI keyboard through USB and outputs a square wave to a GPIO pin. This signal passes through the LM386 audio amplifier for a gain that is internally set to 20 in order to output a wave with a sufficiently large amplitude to the 8Ω speaker. With the Raspberry Pi acting as the master and the FPGA as the slave, the Pi sends notes pressed and unpressed across SPI. The Pi also sends recording and playback signals to the FPGA over GPIO pins indicating the current mode. The FPGA then outputs data to the LED Matrix through several GPIO pins; this includes the serial clock, the current row, RGB values for the top and bottom half of the matrix, a blank signal, and a latch signal. Finally, the FPGA outputs a signal to a GPIO pin connected to an LED indicating the current mode.

Microcontroller Design

On the software side, the amidi.c code from the ALSA project was updated to meet the project specifications. Before any modifications were made, the original code could print information about each received keyboard input. From there, new code added functionality for three modes, with distinct variables keeping track of the current mode.

This project only provides functionality to play one note at a time, so variables monitor the current key and its press status as well as the next key and its pressed status. Keeping track of two keys acts as a buffer to ignore an input if something is already pressed. Modifications to the code account for several possible situations. When a new input is received, the value of next key is updated. If there is no key currently pressed and then a key is pressed, the value of current key is updated to hold the next key value as well as its pressed status. Mathematical manipulations convert the updated current key value to a 4-bit note within an octave, which is then sent to the FPGA via SPI. While a key is pressed, the Pi outputs a square wave of the frequency corresponding to the current key and waits for that key to be unpressed. When the key is unpressed, the current key and its press status are updated and the value zero is sent to the FPGA via SPI. After the code exits the while loop, it waits for the next keyboard input.

The program starts in live mode, where it functions as described in the previous paragraph. A dedicated toggle key on the keyboard shifts from live mode to recording mode and then playback mode. When the toggle key is pressed for the first time, the program skips over updating the current key and playing the corresponding sound. Instead, a starting timestamp marks the beginning of the pause and the mode signals are updated from live mode to recording mode. In recording mode, when a new key is pressed, an ending timestamp marks the end of the pause and the calculated duration between timestamps is stored in a dynamically allocated array. Another starting timestamp marks the start of the pressed note, and when the key is unpressed, an ending timestamp marks the end of that note. Similarly, the duration and value of the note that just played are stored in a dynamically allocated array. This process continues until the stop key is pressed for the second time, when the duration of the final pause is stored in a dynamically allocated array and the mode signals are updated from recording mode to playback mode.

When the program is in playback mode, all inputs from from the keyboard are ignored and instead the program loops over the previously recorded array of notes and durations. The loop skips over the first note in the recording because the recorded pattern always starts with a pause. When the pushbutton is pressed, the program exits the loop and the mode signals are updated from playback mode to live mode. The array resets to allow the next recording to overwrite the previous pattern.

To generate sound, the code from lab 5 was adapted to produce a square wave of appropriate frequency. This square wave passes to the input of the LM386 audio amplifier, and an 8Ω speaker connects to the output of this power amplification circuit.

FPGA Design

A simple SPI module shifts in eight bits during each transfer, but only saves the four least significant bits. These correspond to the note within an octave, from 1 to 12, which then determines the row and color to display.

The shift clock signal that controls the display oscillates for 32 cycles while the RGB values are actively passed in to two rows. These rows are indicated by the three bits A, B, and C, and eight plus that value. For example, when C = 1, B = 1, and A = 0, the values R1, G1, and B1 are passed in to row 6 and R2, G2, and B2 to row 14. Then the shift clock remains low while the display holds before incrementing to the next row and shifting in new values. Just before the shift clock turns on to load the RGB contents of each row, the output enable signal pulses for three clock cycles. This blanks the display, during which the latch signal pulses for a single clock cycle to latch the contents of the shift registers to the output registers.

Several output GPIO pins on the FPGA connect to corresponding input headers on the ribbon cable connecting to the LED display. These signals include three row select bits, three upper RGB bits, three lower RGB bits, latch, output enable, and the shift clock.

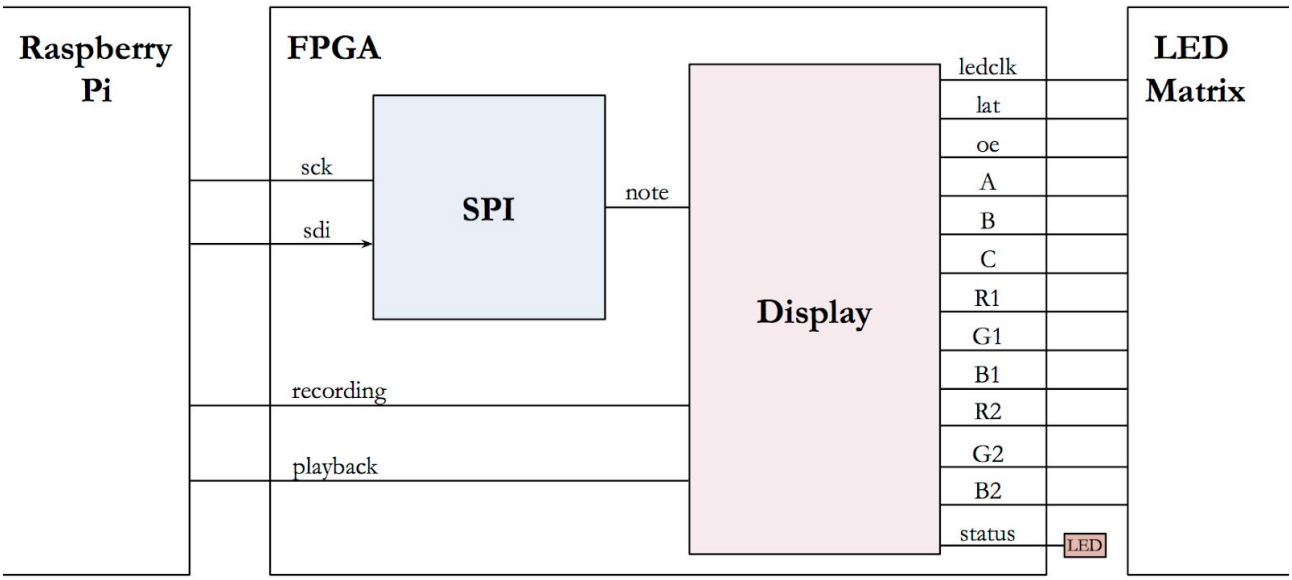

Figure 5: FPGA Block Diagram

Each pair of rows takes a total of 32 clock cycles to shift in RGB values to each column in those rows and another 32 clock cycles before moving to the next rows. The clock passed to the LED matrix slows the 40 MHz FPGA clock to ~300 kHz with an 8-bit counter, so the two rows update every ~0.2 ms, and the full display updates every ~1.7 ms. Using time multiplexing, the refresh rate for the entire process is ~600 Hz, more than fast enough to prevent flickering.

To create the scrolling effect, the FPGA clock is slowed even further by using a 22-bit counter to generate a ~20 Hz scrolling clock. The 4-bit notes sent over the last 32 cycles of the scrolling clock, or the last 1.6 s, are stored in 32 shift registers. For each note, the sequence of pressed and unpressed signals over these 32 clock cycles are stored in twelve 32-bit logic elements. While updating each row, the desired RGB values are sent when there is a one in the note sequence corresponding to the current row.

Each row displays a distinct color, with red corresponding to lower frequencies and violet corresponding to higher frequencies. To display colors beyond the 6 possible 1-bit RGB combinations (excluding white and black), we essentially layered multiple colors by using another counter. The counter increments to five, then resets to zero because the size of this counter is not a multiple of 32. To mix two colors, one color is passed in to the display when the counter is less than three and the other is passed in otherwise.

Finally, to control the status LED, the FPGA clock is slowed with a 24-bit counter to create a blinking signal with a frequency of ~5 Hz when in recording mode. In playback mode, the status bit goes high to emit a constant output to the LED, and in live mode, the status bit goes low to keep the LED off.

Results

We successfully accomplished everything that we set out to achieve in our project proposal, though we failed to reach certain stretch goals. First, we hoped to eliminate the toggle key and replace it with an external push button. Accomplishing this would require a deeper understanding of the more obscure functions of the ALSA project code. Because this modification would rely on additional work with software alone, it seemed out of the scope of a project for a class focused on hardware, so we settled with using a designated key on the keyboard. Second, we wanted to change the orientation of scrolling on the LED matrix to allow for 32 rows of scrolling as opposed to the 12 that we used. The MIDI keyboard has 32 keys, so it may have been more intuitive if each key had its own row on the LED matrix. We chose to only display 12 different rows because the rows that we used correspond better to the internal shift registers in the LED matrix, so displaying a scrolling pattern would be much more difficult with a rotated orientation. Furthermore, displaying 32 distinct colors would call for a binary coded modulation technique, which would entail much more thought and research.

We did run into an infrequent problem that we claim is a feature, not a bug. On rare occasion, if multiple keys are pressed or unpressed simultaneously, the system will continuously play a note even though no keys are pressed. When this happens, the system turns into a game of musical ear training. The user has to listen to the note and guess which note it is, then press the corresponding key to make it stop playing. This problem likely occurs due to the timing of receiving inputs in the code and the failure to receive an unpressed signal.

Ultimately, the system worked well and fulfilled its motivation by bringing happiness to many users during demo day. Figuring out the implementation necessary to control specific signals sent to the LED matrix proved especially difficult, but in the end the display worked as expected. The hard work and long hours in the lab proved worthwhile when we demonstrated our finished project.

References

[1] E155: Microprocessor-Based Systems, http://pages.hmc.edu/harris/class/e155/

[2] midiplus Midi Controller, 32-Key (AKM320), *Amazon*,

https://www.amazon.com/midiplus-AKM320-MIDI-Keyboard-Controller/dp/B00VHKMK64

[3] Medium 16x32 RGB LED matrix panel, *Adafruit*, https://www.adafruit.com/product/420

[4] Advanced Linux Sound Architecture (ALSA) project homepage, https://www.alsa-project.org/main/index.php/Main_Page

[5] Adkins, Glen. RGB LED Panel Driver Tutorial, 2014, https://bikerglen.com/projects/lighting/led-panel-1up/

Parts List

Part	**Source**	**Vendor Part #**	**Price**
AKM320 Midiplus MIDI Controller	Amazon		$35.32
Medium 16x32 RGB LED Matrix Panel	Adafruit	Product ID: 420	$24.95
8Ω Speaker	Digital Lab		
LM386 Low Voltage Audio Power Amplifier	Digital Lab		
Total			**$60.27**

Appendix A: Verilog Modules

```
// top level module
module finalProject(input  logic clk,
                    input  logic sck,
                    input  logic sdi,
                    input  logic recording,
                    input  logic playback,
                    output logic ledclk,
                    output logic lat, oe,
                    output logic A, B, C,
                    output logic R1, G1, B1,
                    output logic R2, G2, B2,
                    output logic [3:0] note,
                    output logic status);

   spi     spi(sck, sdi, note);
   display led(clk, note, recording, playback, ledclk, lat, oe, A, B, C, R1, G1, B1, R2, G2, B2, status);

endmodule


// spi module
module spi(input  logic sck,
           input  logic sdi,
           output logic [3:0] note);

   // shift eight bits from spi into note logic
   // only need to save last four bits sent
   always_ff @(posedge sck) begin
      note <= {note[2:0], sdi};
   end

endmodule


// display module
module display(input  logic clkin,
               input  logic [3:0] note,
               input  logic recording, playback,
               output logic ledclk,
               output logic lat, oe,
               output logic A, B, C,
               output logic R1, G1, B1,
               output logic R2, G2, B2,
               output logic status);

   logic [5:0] c;
   logic [2:0] d;
   logic [23:0] q;
   logic D;
   logic [127:0] notes;
   logic clk, scrollclk;
```

```
logic [4:0] n0, n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11,
logic [4:0] n12, n13, n14, n15, n16, n17, n18, n19, n20, n21,
logic [4:0] n22, n23, n24, n25, n26, n27, n28, n29, n30, n31;
logic [31:0] one, two, three, four, five, six;
logic [31:0] seven, eight, nine, ten, eleven, twelve;

// slowing 40 MHz FPGA clock with a counter
slowClock sc(clkin, q);

// shift clock sent to LED matrix has frequency ~300 kHz
assign clk = q[7];

// clock controlling speed of scrolling has frequency ~20 Hz
assign scrollclk = q[21];

// 32 shift registers to store 4-bit notes sent over last 32 0.05 second intervals
flop reg0(scrollclk, note, n0);
flop reg1(scrollclk, n0, n1);
flop reg2(scrollclk, n1, n2);
flop reg3(scrollclk, n2, n3);
flop reg4(scrollclk, n3, n4);
flop reg5(scrollclk, n4, n5);
flop reg6(scrollclk, n5, n6);
flop reg7(scrollclk, n6, n7);
flop reg8(scrollclk, n7, n8);
flop reg9(scrollclk, n8, n9);
flop reg10(scrollclk, n9, n10);
flop reg11(scrollclk, n10, n11);
flop reg12(scrollclk, n11, n12);
flop reg13(scrollclk, n12, n13);
flop reg14(scrollclk, n13, n14);
flop reg15(scrollclk, n14, n15);
flop reg16(scrollclk, n15, n16);
flop reg17(scrollclk, n16, n17);
flop reg18(scrollclk, n17, n18);
flop reg19(scrollclk, n18, n19);
flop reg20(scrollclk, n19, n20);
flop reg21(scrollclk, n20, n21);
flop reg22(scrollclk, n21, n22);
flop reg23(scrollclk, n22, n23);
flop reg24(scrollclk, n23, n24);
flop reg25(scrollclk, n24, n25);
flop reg26(scrollclk, n25, n26);
flop reg27(scrollclk, n26, n27);
flop reg28(scrollclk, n27, n28);
flop reg29(scrollclk, n28, n29);
flop reg30(scrollclk, n29, n30);
flop reg31(scrollclk, n30, n31);

// sequence of pressed signals for each note over last 32 0.5 second intervals
// stored in 12 different logic elements for each note in an octave
assign one = {n30==1, n29==1, n28==1, n27==1, n26==1, n25==1, n24==1, n23==1,
              n22==1, n21==1, n20==1, n19==1, n18==1, n17==1, n16==1, n15==1,
              n14==1, n13==1, n12==1, n11==1, n10==1, n9==1,  n8==1,  n7==1,
```

```
          n6==1,  n5==1,  n4==1,  n3==1,  n2==1,  n1==1,  n0==1,  n31==1};

assign two = {n30==2, n29==2, n28==2, n27==2, n26==2, n25==2, n24==2, n23==2,
          n22==2, n21==2, n20==2, n19==2, n18==2, n17==2, n16==2, n15==2,
          n14==2, n13==2, n12==2, n11==2, n10==2, n9==2,  n8==2,  n7==2,
          n6==2,  n5==2,  n4==2,  n3==2,  n2==2,  n1==2,  n0==2,  n31==2};

assign three = {n30==3, n29==3, n28==3, n27==3, n26==3, n25==3, n24==3, n23==3,
                n22==3, n21==3, n20==3, n19==3, n18==3, n17==3, n16==3, n15==3,
                n14==3, n13==3, n12==3, n11==3, n10==3, n9==3,  n8==3,  n7==3,
                n6==3,  n5==3,  n4==3,  n3==3,  n2==3,  n1==3,  n0==3,  n31==3};

assign four = {n30==4, n29==4, n28==4, n27==4, n26==4, n25==4, n24==4, n23==4,
          n22==4, n21==4, n20==4, n19==4, n18==4, n17==4, n16==4, n15==4,
          n14==4, n13==4, n12==4, n11==4, n10==4, n9==4,  n8==4,  n7==4,
          n6==4,  n5==4,  n4==4,  n3==4,  n2==4,  n1==4,  n0==4,  n31==4};

assign five = {n30==5, n29==5, n28==5, n27==5, n26==5, n25==5, n24==5, n23==5,
          n22==5, n21==5, n20==5, n19==5, n18==5, n17==5, n16==5, n15==5,
          n14==5, n13==5, n12==5, n11==5, n10==5, n9==5,  n8==5,  n7==5,
          n6==5,  n5==5,  n4==5,  n3==5,  n2==5,  n1==5,  n0==5,  n31==5};

assign six = {n30==6, n29==6, n28==6, n27==6, n26==6, n25==6, n24==6, n23==6,
          n22==6, n21==6, n20==6, n19==6, n18==6, n17==6, n16==6, n15==6,
          n14==6, n13==6, n12==6, n11==6, n10==6, n9==6,  n8==6,  n7==6,
          n6==6,  n5==6,  n4==6,  n3==6,  n2==6,  n1==6,  n0==6,  n31==6};

assign seven = {n30==7, n29==7, n28==7, n27==7, n26==7, n25==7, n24==7, n23==7,
          n22==7, n21==7, n20==7, n19==7, n18==7, n17==7, n16==7, n15==7,
          n14==7, n13==7, n12==7, n11==7, n10==7, n9==7,  n8==7,  n7==7,
          n6==7,  n5==7,  n4==7,  n3==7,  n2==7,  n1==7,  n0==7,  n31==7};

assign eight = {n30==8, n29==8, n28==8, n27==8, n26==8, n25==8, n24==8, n23==8,
          n22==8, n21==8, n20==8, n19==8, n18==8, n17==8, n16==8, n15==8,
          n14==8, n13==8, n12==8, n11==8, n10==8, n9==8,  n8==8,  n7==8,
          n6==8,  n5==8,  n4==8,  n3==8,  n2==8,  n1==8,  n0==8,  n31==8};

assign nine = {n30==9, n29==9, n28==9, n27==9, n26==9, n25==9, n24==9, n23==9,
          n22==9, n21==9, n20==9, n19==9, n18==9, n17==9, n16==9, n15==9,
          n14==9, n13==9, n12==9, n11==9, n10==9, n9==9,  n8==9,  n7==9,
          n6==9,  n5==9,  n4==9,  n3==9,  n2==9,  n1==9,  n0==9,  n31==9};

assign ten = {n30==10, n29==10, n28==10, n27==10, n26==10, n25==10, n24==10,
          n23==10, n22==10, n21==10, n20==10, n19==10, n18==10, n17==10,
          n16==10, n15==10, n14==10, n13==10, n12==10, n11==10, n10==10,
          n9==10,  n8==10,  n7==10, n6==10,  n5==10,  n4==10,  n3==10,
          n2==10,  n1==10,  n0==10,  n31==10};

assign eleven = {n30==11, n29==11, n28==11, n27==11, n26==11, n25==11, n24==11,
          n23==11, n22==11, n21==11, n20==11, n19==11, n18==11, n17==11,
          n16==11, n15==11, n14==11, n13==11, n12==11, n11==11, n10==11,
          n9==11,  n8==11,  n7==11, n6==11,  n5==11,  n4==11,  n3==11,
          n2==11,  n1==11,  n0==11,  n31==11};
```

```
assign twelve = {n30==12, n29==12, n28==12, n27==12, n26==12, n25==12, n24==12,
                 n23==12, n22==12, n21==12, n20==12, n19==12, n18==12, n17==12,
                 n16==12, n15==12, n14==12, n13==12, n12==12, n11==12, n10==12,
                 n9==12,  n8==12,  n7==12, n6==12,  n5==12,  n4==12,  n3==12,
                 n2==12,  n1==12,  n0==12,  n31==12};

// two counters
// c determines LED shift clock schedule and shift register loading
// d allows for offset to layer multiple colors beyond 8 RGB combinations
always_ff @(posedge clk) begin
   c <= c + 1;
   d <= d + 1;
   if (d >= 6) d <= 0;
   // row increments when counter c fills
   if (c == 0) {D,C,B,A} = {D,C,B,A} + 1;
end

// LED shift clock only on when loading RGB values into column shift registers
assign ledclk = (clk & !c[5]);
// display blanks before loading next row
assign oe = (c == 62 | c == 63 | c == 0);
// RGB values latch from shift registers to output registers during blank
assign lat = (c == 63);

// updates colors based on current row and scrolling note positions
always_comb begin
   case({D,C,B,A})
      // rows 1 through 6 controlled by R1, G1, and B1
      1: begin
            {R2,G2,B2} = 3'b000;
            if (one[c]) {R1,G1,B1} = 3'b100;
            else {R1,G1,B1} = 3'b000;
         end
      2: begin
            {R2,G2,B2} = 3'b000;
            if (two[c] & d<3) {R1,G1,B1} = 3'b100;
            else if (two[c] & d>=3) {R1,G1,B1} = 3'b110;
            else {R1,G1,B1} = 3'b000;
         end
      3: begin
            {R2,G2,B2} = 3'b000;
            if (three[c]) {R1,G1,B1} = 3'b110;
            else {R1,G1,B1} = 3'b000;
         end
      4: begin
            {R2,G2,B2} = 3'b000;
            if (four[c] & d<3) {R1,G1,B1} = 3'b110;
            else if (four[c] & d>=3) {R1,G1,B1} = 3'b010;
            else {R1,G1,B1} = 3'b000;
         end
      5: begin
            {R2,G2,B2} = 3'b000;
            if (five[c]) {R1,G1,B1} = 3'b010;
            else {R1,G1,B1} = 3'b000;
```

```
            end
        6: begin
            {R2,G2,B2} = 3'b000;
            if (six[c] & d<3) {R1,G1,B1} = 3'b010;
            else if (six[c] & d>=3) {R1,G1,B1} = 3'b011;
            else {R1,G1,B1} = 3'b000;
           end
        // rows 7 through 12 controlled by R2, G2, and B2
        7: begin
            {R1,G1,B1} = 3'b000;
            if (seven[c]) {R2,G2,B2} = 3'b011;
            else {R2,G2,B2} = 3'b000;
           end
        8: begin
            {R1,G1,B1} = 3'b000;
            if (eight[c] & d<3) {R2,G2,B2} = 3'b011;
            else if (eight[c] & d>=3) {R2,G2,B2} = 3'b001;
            else {R2,G2,B2} = 3'b000;
           end
        9: begin
            {R1,G1,B1} = 3'b000;
            if (nine[c]) {R2,G2,B2} = 3'b001;
            else {R2,G2,B2} = 3'b000;
           end
        10: begin
            {R1,G1,B1} = 3'b000;
            if (ten[c] & d<3) {R2,G2,B2} = 3'b001;
            else if (ten[c] & d>=3) {R2,G2,B2} = 3'b101;
            else {R2,G2,B2} = 3'b000;
           end
        11: begin
            {R1,G1,B1} = 3'b000;
            if (eleven[c]) {R2,G2,B2} = 3'b101;
            else {R2,G2,B2} = 3'b000;
           end
        12: begin
            {R1,G1,B1} = 3'b000;
            if (twelve[c] & d<3) {R2,G2,B2} = 3'b101;
            else if (twelve[c] & d>=3) {R2,G2,B2} = 3'b100;
            else {R2,G2,B2} = 3'b000;
           end
        default: begin
            {R1,G1,B1} = 3'b000;
            {R2,G2,B2} = 3'b000;
           end
      endcase
   end

   // status LED blinks at ~5 Hz in recording mode,
   // on in playback mode, and off in live mode
   assign status = recording ? (playback ? 0 : q[23]) : (playback ? 1 : 0);

endmodule
```

```
// slow clock module
module slowClock(input logic clk,
                 output logic [21:0] q);

   always_ff @(posedge clk) begin
      q <= q + 1;
   End

endmodule


// flip flop module
module flop(input logic clk,
            input logic [3:0] in,
            output logic [3:0] out);

   always_ff @(posedge clk) begin
      out <= in;
   end

endmodule


// test bench
module testbench();
   logic clkin, clk, lat, oe, A, B, C, R1, B1, G1, R2, B2, G2;

   counter ctr(clkin, clk, lat, oe, A, B, C, R1, B1, G1, R2, B2, G2);

   initial begin
      forever begin
         clkin = 1'b0; #5;
         clkin = 1'b1; #5;
      end
   End

endmodule
```

Appendix B: C Code

```
/*
 *  amidi.c - read from/write to RawMIDI ports
 *
 *  Copyright (c) Clemens Ladisch <clemens@ladisch.de>
 *
 *  Modified by Spencer Rosen and Vicki Moran
 *  December 11, 2018
 *  sarosen@hmc.edu and vmoran@hmc.edu
 *
 *
 *   This program is free software; you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation; either version 2 of the License, or
 *   (at your option) any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *   along with this program; if not, write to the Free Software
 *   Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA
 */

#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <getopt.h>
#include <errno.h>
#include <signal.h>
#include <sys/timerfd.h>
#include <sys/types.h>
#include <sys/poll.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
```

```
#include <alsa/asoundlib.h>
#include "aconfig.h"
#include "version.h"
#include <sys/mman.h>
#include <math.h>
#include "EasyPIO.h"

#define GENOUT 25
#define PUSH 13
#define RECORDING 17
#define PLAYBACK 4
#define OUTPUT 1
#define INPUT 0
#define STOPKEY 0x54 //C5

#define NSEC_PER_SEC 1000000000L

static int do_device_list, do_rawmidi_list;
static char *port_name = "default";
static char *send_file_name;
static char *receive_file_name;
static char *send_hex;
static char *send_data;
static int send_data_length;
static int receive_file;
static int dump;
static float timeout;
static int stop;
static int sysex_interval;
static snd_rawmidi_t *input, **inputp;
static snd_rawmidi_t *output, **outputp;

static void error(const char *format, ...)
{
   va_list ap;

   va_start(ap, format);
   vfprintf(stderr, format, ap);
   va_end(ap);
   putc('\n', stderr);
}

static void usage(void)
```

```
{
    printf(
        "Usage: amidi options\n"
        "\n"
        "-h, --help!!!!!                this help\n"
        "-V, --version                  print current version\n"
        "-l, --list-devices             list all hardware ports\n"
        "-L, --list-rawmidis            list all RawMIDI definitions\n"
        "-p, --port=name                select port by name\n"
        "-s, --send=file                send the contents of a (.syx) file\n"
        "-r, --receive=file             write received data into a file\n"
        "-S, --send-hex=\"...\"           send hexadecimal bytes\n"
        "-d, --dump                     print received data as hexadecimal bytes\n"
        "-t, --timeout=seconds          exits when no data has been received\n"
        "                               for the specified duration\n"
        "-a, --active-sensing           include active sensing bytes\n"
        "-c, --clock                    include clock bytes\n"
        "-i, --sysex-interval=mseconds   delay in between each SysEx message\n");
}

static void version(void)
{
    puts("amidi version " SND_UTIL_VERSION_STR);
}

static void *my_malloc(size_t size)
{
    void *p = malloc(size);
    if (!p) {
        error("out of memory");
        exit(EXIT_FAILURE);
    }
    return p;
}

static void list_device(snd_ctl_t *ctl, int card, int device)
{
    snd_rawmidi_info_t *info;
    const char *name;
    const char *sub_name;
    int subs, subs_in, subs_out;
    int sub;
    int err;
```

```
snd_rawmidi_info_alloca(&info);
snd_rawmidi_info_set_device(info, device);

snd_rawmidi_info_set_stream(info, SND_RAWMIDI_STREAM_INPUT);
err = snd_ctl_rawmidi_info(ctl, info);
if (err >= 0)
    subs_in = snd_rawmidi_info_get_subdevices_count(info);
else
    subs_in = 0;

snd_rawmidi_info_set_stream(info, SND_RAWMIDI_STREAM_OUTPUT);
err = snd_ctl_rawmidi_info(ctl, info);
if (err >= 0)
    subs_out = snd_rawmidi_info_get_subdevices_count(info);
else
    subs_out = 0;

subs = subs_in > subs_out ? subs_in : subs_out;
if (!subs)
    return;

for (sub = 0; sub < subs; ++sub) {
    snd_rawmidi_info_set_stream(info, sub < subs_in ?
                SND_RAWMIDI_STREAM_INPUT :
                SND_RAWMIDI_STREAM_OUTPUT);
    snd_rawmidi_info_set_subdevice(info, sub);
    err = snd_ctl_rawmidi_info(ctl, info);
    if (err < 0) {
        error("cannot get rawmidi information %d:%d:%d: %s\n",
                card, device, sub, snd_strerror(err));
        return;
    }
    name = snd_rawmidi_info_get_name(info);
    sub_name = snd_rawmidi_info_get_subdevice_name(info);
    if (sub == 0 && sub_name[0] == '\0') {
        printf("%c%c  hw:%d,%d    %s",
                sub < subs_in ? 'I' : ' ',
                sub < subs_out ? 'O' : ' ',
                card, device, name);
        if (subs > 1)
            printf(" (%d subdevices)", subs);
        putchar('\n');
```

```
            break;
        } else {
            printf("%c%c  hw:%d,%d,%d  %s\n",
                        sub < subs_in ? 'I' : ' ',
                         sub < subs_out ? 'O' : ' ',
                        card, device, sub, sub_name);
        }
    }
}

static void list_card_devices(int card)
{
    snd_ctl_t *ctl;
    char name[32];
    int device;
    int err;

    sprintf(name, "hw:%d", card);
    if ((err = snd_ctl_open(&ctl, name, 0)) < 0) {
        error("cannot open control for card %d: %s", card, snd_strerror(err));
        return;
    }
    device = -1;
    for (;;) {
        if ((err = snd_ctl_rawmidi_next_device(ctl, &device)) < 0) {
            error("cannot determine device number: %s", snd_strerror(err));
            break;
        }
        if (device < 0)
            break;
        list_device(ctl, card, device);
    }
    snd_ctl_close(ctl);
}

static void device_list(void)
{
    int card, err;

    card = -1;
    if ((err = snd_card_next(&card)) < 0) {
        error("cannot determine card number: %s", snd_strerror(err));
        return;
```

```
    }
    if (card < 0) {
        error("no sound card found");
        return;
    }
    puts("Dir Device         Name");
    do {
        list_card_devices(card);
        if ((err = snd_card_next(&card)) < 0) {
            error("cannot determine card number: %s", snd_strerror(err));
            break;
        }
    } while (card >= 0);
}

static void rawmidi_list(void)
{
    snd_output_t *output;
    snd_config_t *config;
    int err;

    if ((err = snd_config_update()) < 0) {
        error("snd_config_update failed: %s", snd_strerror(err));
        return;
    }
    if ((err = snd_output_stdio_attach(&output, stdout, 0)) < 0) {
        error("snd_output_stdio_attach failed: %s", snd_strerror(err));
        return;
    }
    if (snd_config_search(snd_config, "rawmidi", &config) >= 0) {
        puts("RawMIDI list:");
        snd_config_save(config, output);
    }
    snd_output_close(output);
}

static int send_midi_interleaved(void)
{
    int err;
    char *data = send_data;
    size_t buffer_size;
    snd_rawmidi_params_t *param;
    snd_rawmidi_status_t *st;
```

```
    snd_rawmidi_status_alloca(&st);

    snd_rawmidi_params_alloca(&param);
    snd_rawmidi_params_current(output, param);
    buffer_size = snd_rawmidi_params_get_buffer_size(param);

    while (data < (send_data + send_data_length)) {
        int len = send_data + send_data_length - data;
        char *temp;

        if (data > send_data) {
            snd_rawmidi_status(output, st);
            do {
                /* 320 µs per byte as noted in Page 1 of MIDI spec */
                usleep((buffer_size - snd_rawmidi_status_get_avail(st)) * 320);
                snd_rawmidi_status(output, st);
            } while(snd_rawmidi_status_get_avail(st) < buffer_size);
            usleep(sysex_interval * 1000);
        }

        /* find end of SysEx */
        if ((temp = memchr(data, 0xf7, len)) != NULL)
            len = temp - data + 1;

        if ((err = snd_rawmidi_write(output, data, len)) < 0)
            return err;

        data += len;
    }

    return 0;
}

static void load_file(void)
{
    int fd;
    off_t length;

    fd = open(send_file_name, O_RDONLY);
    if (fd == -1) {
        error("cannot open %s - %s", send_file_name, strerror(errno));
        return;
```

```
    }
    length = lseek(fd, 0, SEEK_END);
    if (length == (off_t)-1) {
        error("cannot determine length of %s: %s", send_file_name, strerror(errno));
        goto _error;
    }
    send_data = my_malloc(length);
    lseek(fd, 0, SEEK_SET);
    if (read(fd, send_data, length) != length) {
        error("cannot read from %s: %s", send_file_name, strerror(errno));
        goto _error;
    }
    if (length >= 4 && !memcmp(send_data, "MThd", 4)) {
        error("%s is a Standard MIDI File; use aplaymidi to send it", send_file_name);
        goto _error;
    }
    send_data_length = length;
    goto _exit;
_error:
    free(send_data);
    send_data = NULL;
_exit:
    close(fd);
}

static int hex_value(char c)
{
    if ('0' <= c && c <= '9')
        return c - '0';
    if ('A' <= c && c <= 'F')
        return c - 'A' + 10;
    if ('a' <= c && c <= 'f')
        return c - 'a' + 10;
    error("invalid character %c", c);
    return -1;
}

static void parse_data(void)
{
    const char *p;
    int i, value;

    send_data = my_malloc(strlen(send_hex)); /* guesstimate */
```

```
    i = 0;
    value = -1; /* value is >= 0 when the first hex digit of a byte has been read */
    for (p = send_hex; *p; ++p) {
        int digit;
        if (isspace((unsigned char)*p)) {
            if (value >= 0) {
                send_data[i++] = value;
                value = -1;
            }
            continue;
        }
        digit = hex_value(*p);
        if (digit < 0) {
            send_data = NULL;
            return;
        }
        if (value < 0) {
            value = digit;
        } else {
            send_data[i++] = (value << 4) | digit;
            value = -1;
        }
    }
    if (value >= 0)
        send_data[i++] = value;
    send_data_length = i;
}

/*
 * prints MIDI commands, formatting them nicely
 */
static void print_byte(unsigned char byte)
{
    static enum {
        STATE_UNKNOWN,
        STATE_1PARAM,
        STATE_1PARAM_CONTINUE,
        STATE_2PARAM_1,
        STATE_2PARAM_2,
        STATE_2PARAM_1_CONTINUE,
        STATE_SYSEX
    } state = STATE_UNKNOWN;
    int newline = 0;
```

```
if (byte >= 0xf8)
    newline = 1;
else if (byte >= 0xf0) {
    newline = 1;
    switch (byte) {
    case 0xf0:
        state = STATE_SYSEX;
        break;
    case 0xf1:
    case 0xf3:
        state = STATE_1PARAM;
        break;
    case 0xf2:
        state = STATE_2PARAM_1;
        break;
    case 0xf4:
    case 0xf5:
    case 0xf6:
        state = STATE_UNKNOWN;
        break;
    case 0xf7:
        newline = state != STATE_SYSEX;
        state = STATE_UNKNOWN;
        break;
    }
} else if (byte >= 0x80) {
    newline = 1;
    if (byte >= 0xc0 && byte <= 0xdf)
        state = STATE_1PARAM;
    else
        state = STATE_2PARAM_1;
} else /* b < 0x80 */ {
    int running_status = 0;
    newline = state == STATE_UNKNOWN;
    switch (state) {
    case STATE_1PARAM:
        state = STATE_1PARAM_CONTINUE;
        break;
    case STATE_1PARAM_CONTINUE:
        running_status = 1;
        break;
    case STATE_2PARAM_1:
```

```
            state = STATE_2PARAM_2;
            break;
        case STATE_2PARAM_2:
            state = STATE_2PARAM_1_CONTINUE;
            break;
        case STATE_2PARAM_1_CONTINUE:
            running_status = 1;
            state = STATE_2PARAM_2;
            break;
        default:
            break;
        }
        if (running_status)
            fputs("\n  ", stdout);
    }
    printf("%c%02X", newline ? '\n' : ' ', byte);
}

static void sig_handler(int dummy)
{
    stop = 1;
}

static void add_send_hex_data(const char *str)
{
    int length;
    char *s;

    length = (send_hex ? strlen(send_hex) + 1 : 0) + strlen(str) + 1;
    s = my_malloc(length);
    if (send_hex) {
        strcpy(s, send_hex);
        strcat(s, " ");
    } else {
        s[0] = '\0';
    }
    strcat(s, str);
    free(send_hex);
    send_hex = s;
}
```

```
// This function takes in a hexadecimal value for a key and
// outputs the corresponding note in an octave (C=1, B=12, etc)
int keytonote(unsigned char key) {
   if (key == 0x00) return 0;
   else return (key%12)+1;
}


// This function takes in a hexadecimal value for a key and
// outputs the corresponding frequency of the note
int keytofreq(unsigned char key) {
          if (key == 0x00) return 0;
   //the lowest key possible is F-2, which has a frequency of 5.45675 Hz
          //increasing one note corresponds to increasing 2^(1/12) times the freq
          else return (int) (10.9135*pow(2,(double)(key-0x05)/12));
}


// This function takes in a frequency and generates a square wave of that frequency
void square(int frequency) {
   if (frequency!=0) {
      int delaytimemicros = 500000/frequency;
      digitalWrite(GENOUT,1);
      delayMicros(delaytimemicros);
      digitalWrite(GENOUT,0);
      delayMicros(delaytimemicros);
   }
}

// This function generates a square wave of a certain frequency for a certain duration
int timedwave(int frequency, int durationmicros) {
   // A frequency of 0 corresponds to a rest
   if (frequency==0) {
      if (delayMicros(durationmicros)) {
         // delayMicros terminates and returns 1 if the pushbutton is pressed
         return 1;
      }
   } else {
      int periodmicros = 1000000/frequency;
      // The number of periods of the square wave is the total duration
```

```
        // divided by the period of one square wave
        int numperiods = durationmicros/periodmicros;
        // The square wave stays high/low for half the period
        //int delaytimemicros = periodmicros/2;
        for (int i = 0; i < numperiods; i++) {
            square(frequency);
            if (digitalRead(PUSH)) {
                return 1;
            }
        }
    }
    return 0;
}

// This function loops over the stored array of keys and durations to play the recorded pattern
void playback(int* key_array, int* dur_array, int noteslength) {
    int stop;
    int i = 1; // skip over the first pause
    while (i < noteslength) {
        spiSendReceive(keytonote(key_array[i]));
        stop = timedwave(keytofreq(key_array[i]), dur_array[i]);
        if (stop) {
            break;
        }
        i++;
        if (i == noteslength) {
            i = 1;
        }
    }
    spiSendReceive(0);
}



int main(int argc, char *argv[])
{
    pioInit();
    spiInit(244000,0);
    pinMode(GENOUT, OUTPUT);
    pinMode(PUSH, INPUT);
    pinMode(RECORDING, OUTPUT);
    pinMode(PLAYBACK, OUTPUT);
```

```
static const char short_options[] = "hVlLp:s:r:S::dt:aci:";
static const struct option long_options[] = {
    {"help", 0, NULL, 'h'},
    {"version", 0, NULL, 'V'},
    {"list-devices", 0, NULL, 'l'},
    {"list-rawmidis", 0, NULL, 'L'},
    {"port", 1, NULL, 'p'},
    {"send", 1, NULL, 's'},
    {"receive", 1, NULL, 'r'},
    {"send-hex", 2, NULL, 'S'},
    {"dump", 0, NULL, 'd'},
    {"timeout", 1, NULL, 't'},
    {"active-sensing", 0, NULL, 'a'},
    {"clock", 0, NULL, 'c'},
    {"sysex-interval", 1, NULL, 'i'},
    { }
};
int c, err, ok = 0;
int ignore_active_sensing = 1;
int ignore_clock = 1;
int do_send_hex = 0;
struct itimerspec itimerspec = { .it_interval = { 0, 0 } };

while ((c = getopt_long(argc, argv, short_options,
                long_options, NULL)) != -1) {
    switch (c) {
    case 'h':
        usage();
        return 0;
    case 'V':
        version();
        return 0;
    case 'l':
        do_device_list = 1;
        break;
    case 'L':
        do_rawmidi_list = 1;
        break;
    case 'p':
        port_name = optarg;
        break;
    case 's':
        send_file_name = optarg;
```

```
            break;
        case 'r':
            receive_file_name = optarg;
            break;
        case 'S':
            do_send_hex = 1;
            if (optarg)
                add_send_hex_data(optarg);
            break;
        case 'd':
            dump = 1;
            break;
        case 't':
            if (optarg)
                timeout = atof(optarg);
            break;
        case 'a':
            ignore_active_sensing = 0;
            break;
        case 'c':
            ignore_clock = 0;
            break;
        case 'i':
            sysex_interval = atoi(optarg);
            break;
        default:
            error("Try `amidi --help' for more information.");
            return 1;
        }
    }
    if (do_send_hex) {
        /* data for -S can be specified as multiple arguments */
        if (!send_hex && !argv[optind]) {
            error("Please specify some data for --send-hex.");
            return 1;
        }
        for (; argv[optind]; ++optind)
            add_send_hex_data(argv[optind]);
    } else {
        if (argv[optind]) {
            error("%s is not an option.", argv[optind]);
            return 1;
        }
```

```
}

if (do_rawmidi_list)
    rawmidi_list();
if (do_device_list)
    device_list();
if (do_rawmidi_list || do_device_list)
    return 0;

if (!send_file_name && !receive_file_name && !send_hex && !dump) {
    error("Please specify at least one of --send, --receive, --send-hex, or --dump.");
    return 1;
}
if (send_file_name && send_hex) {
    error("--send and --send-hex cannot be specified at the same time.");
    return 1;
}

if (send_file_name)
    load_file();
else if (send_hex)
    parse_data();
if ((send_file_name || send_hex) && !send_data)
    return 1;

if (receive_file_name) {
    receive_file = creat(receive_file_name, 0666);
    if (receive_file == -1) {
        error("cannot create %s: %s", receive_file_name, strerror(errno));
        return -1;
    }
} else {
    receive_file = -1;
}

if (receive_file_name || dump)
    inputp = &input;
else
    inputp = NULL;
if (send_data)
    outputp = &output;
else
    outputp = NULL;
```

```
if ((err = snd_rawmidi_open(inputp, outputp, port_name, SND_RAWMIDI_NONBLOCK)) < 0) {
    error("cannot open port \"%s\": %s", port_name, snd_strerror(err));
    goto _exit2;
}

if (inputp)
    snd_rawmidi_read(input, NULL, 0); /* trigger reading */

if (send_data) {
    if ((err = snd_rawmidi_nonblock(output, 0)) < 0) {
        error("cannot set blocking mode: %s", snd_strerror(err));
        goto _exit;
    }
    if (!sysex_interval) {
        if ((err = snd_rawmidi_write(output, send_data, send_data_length)) < 0) {
            error("cannot send data: %s", snd_strerror(err));
            return err;
        }
    } else {
        if ((err = send_midi_interleaved()) < 0) {
            error("cannot send data: %s", snd_strerror(err));
            return err;
        }
    }
}

if (inputp) {
    int read = 0;
    int npfds;
    struct pollfd *pfds;

    npfds = 1 + snd_rawmidi_poll_descriptors_count(input);
    pfds = alloca(npfds * sizeof(struct pollfd));

    if (timeout > 0) {
        pfds[0].fd = timerfd_create(CLOCK_MONOTONIC, 0);
        if (pfds[0].fd == -1) {
            error("cannot create timer: %s", strerror(errno));
            goto _exit;
        }
        pfds[0].events = POLLIN;
    } else {
```

```
    pfds[0].fd = -1;
}

snd_rawmidi_poll_descriptors(input, &pfds[1], npfds - 1);

signal(SIGINT, sig_handler);

if (timeout > 0) {
    float timeout_int;

    itimerspec.it_value.tv_nsec = modff(timeout, &timeout_int) * NSEC_PER_SEC;
    itimerspec.it_value.tv_sec = timeout_int;
    err = timerfd_settime(pfds[0].fd, 0, &itimerspec, NULL);
    if (err < 0) {
        error("cannot set timer: %s", strerror(errno));
        goto _exit;
    }
}
int currentPressed = 0;
int currentKey = 0;
int nextPressed = 0;
int nextKey = 0;

int *key_array;
int *dur_array;
key_array = (int *)malloc(sizeof(int)*100);
dur_array = (int *)malloc(sizeof(int)*100);
if (key_array == NULL) {
    printf("malloc of key failed");
    exit(1);
}
if (dur_array == NULL) {
    printf("malloc of dur failed");
    exit(1);
}
int j = 0;
int noteslength = 0;
int starttime = *(sys_timer+1);
int endtime;

int livemode = 1;
int recordingmode = 0;
int playbackmode = 0;
```

```
digitalWrite(RECORDING, 0);
digitalWrite(PLAYBACK, 0);
for (;;) {
  unsigned char buf[256];
  int i, length;

  unsigned short revents;

  err = poll(pfds, npfds, -1);
  if (stop || (err < 0 && errno == EINTR))
    break;
  if (err < 0) {
    error("poll failed: %s", strerror(errno));
    break;
  }

  err = snd_rawmidi_poll_descriptors_revents(input, &pfds[1], npfds - 1, &revents);
  if (err < 0) {
    error("cannot get poll events: %s", snd_strerror(errno));
    break;
  }
  if (revents & (POLLERR | POLLHUP))
    break;
  if (!(revents & POLLIN)) {
    if (pfds[0].revents & POLLIN)
      break;
    continue;
  }

  err = snd_rawmidi_read(input, buf, sizeof(buf));
  if (err == -EAGAIN)
    continue;
  if (err < 0) {
    error("cannot read from port \"%s\": %s", port_name, snd_strerror(err));
    break;
  }
  length = 0;
  for (i = 0; i < err; ++i)
    if ((buf[i] != MIDI_CMD_COMMON_CLOCK &&
         buf[i] != MIDI_CMD_COMMON_SENSING) ||
         (buf[i] == MIDI_CMD_COMMON_CLOCK   && !ignore_clock) ||
         (buf[i] == MIDI_CMD_COMMON_SENSING && !ignore_active_sensing)) {
      buf[length++] = buf[i];
```

```
    }
if (length == 0)
    continue;
read += length;

if (receive_file != -1)
    write(receive_file, buf, length);

if (dump) {
    for (i = 0; i < length; ++i) {
        //print_byte(buf[i]);
    }
    nextPressed = (buf[0]==0x90);
    nextKey = buf[1];

    // if the stop key is pressed, toggle modes
    if (nextKey == STOPKEY && nextPressed) {
        if (livemode) {
            printf("\nSwitch from livemode to recording mode\n");
            digitalWrite(RECORDING, 1);
            starttime = *(sys_timer+1);
            livemode = 0;
            recordingmode = 1;
        } else if (recordingmode) {
            printf("\nSwitch from recording mode to playback mode\n");
            digitalWrite(RECORDING, 0);
            digitalWrite(PLAYBACK, 1);
            endtime = *(sys_timer+1);
            key_array[j] = 0;
            dur_array[j] = endtime-starttime;
            printf("\nStoring {%x, %d}\n", 0, endtime-starttime);
            j++;
            noteslength++;
            recordingmode = 0;
            playbackmode = 1;
        }
    }

    // a new key is pressed
    if (!currentPressed && nextPressed && nextKey!=STOPKEY) {
        currentKey = nextKey;
        currentPressed = nextPressed;
        spiSendReceive(keytonote(currentKey));
```

```
    if (recordingmode && currentKey!=STOPKEY) {
        endtime = *(sys_timer+1);
        key_array[j] = 0;
        dur_array[j] = endtime-starttime;
        printf("\nStoring {%x, %d}\n", 0, endtime-starttime);
        starttime = *(sys_timer+1);
        j++;
        noteslength++;
    }
}

starttime = *(sys_timer+1);
// play a note until the key is released
while(currentPressed && !playbackmode) {
    if (currentKey != STOPKEY) {
        square(keytofreq(currentKey));
    }

    // read in new input
    err = snd_rawmidi_read(input, buf, sizeof(buf));
    if (err == -EAGAIN)
        continue;
    if (err < 0) {
        error("cannot read from port \"%s\": %s", port_name, snd_strerror(err));
        break;
    }
    length = 0;
    for (i = 0; i < err; ++i)
        if ((buf[i] != MIDI_CMD_COMMON_CLOCK &&
          buf[i] != MIDI_CMD_COMMON_SENSING) ||
          (buf[i] == MIDI_CMD_COMMON_CLOCK   && !ignore_clock) ||
          (buf[i] == MIDI_CMD_COMMON_SENSING && !ignore_active_sensing)) {
            buf[length++] = buf[i];
        }
    if (length == 0)
        continue;
    read += length;

    nextPressed = (buf[0]==0x90);
    nextKey = buf[1];

    // key is unpressed
    if (currentKey==nextKey && nextPressed==0) {
```

```
            if (recordingmode && currentKey!=STOPKEY) {
                endtime = *(sys_timer+1);
                key_array[j] = currentKey;
                dur_array[j] = endtime-starttime;
                printf("\nSTORING {%x, %d}\n", currentKey, endtime-starttime);
                j++;
                noteslength++;
            }
            currentKey = 0;
            currentPressed = nextPressed;
            spiSendReceive(keytonote(currentKey));
            starttime = *(sys_timer+1);
        }
    }
    fflush(stdout);
}

if (timeout > 0) {
    err = timerfd_settime(pfds[0].fd, 0, &itimerspec, NULL);
    if (err < 0) {
        error("cannot set timer: %s", strerror(errno));
        break;
    }
}

// loop through stored array if in playback mode
if (playbackmode) {
    printf("\nnoteslength: %d\n", noteslength);
    for (int i = 0; i < noteslength; i++) {
        printf("{%x, %d}\n", key_array[i], dur_array[i]);
    }
    playback(key_array, dur_array, noteslength);
    playbackmode = 0;
    livemode = 1;
    printf("\nSwitch from playbackmode to livemode\n");
    digitalWrite(PLAYBACK, 0);
    j = 0;
    noteslength = 0;
}
}
if (isatty(fileno(stdout))) {
    printf("\nDone\n");
    //digitalWrite(RECORDING, 0);
```

```
            //digitalWrite(PLAYBACK, 0);
        }
    }

    ok = 1;
_exit:
    if (inputp)
        snd_rawmidi_close(input);
    if (outputp)
        snd_rawmidi_close(output);
_exit2:
    if (receive_file != -1)
        close(receive_file);
    return !ok;
}
```