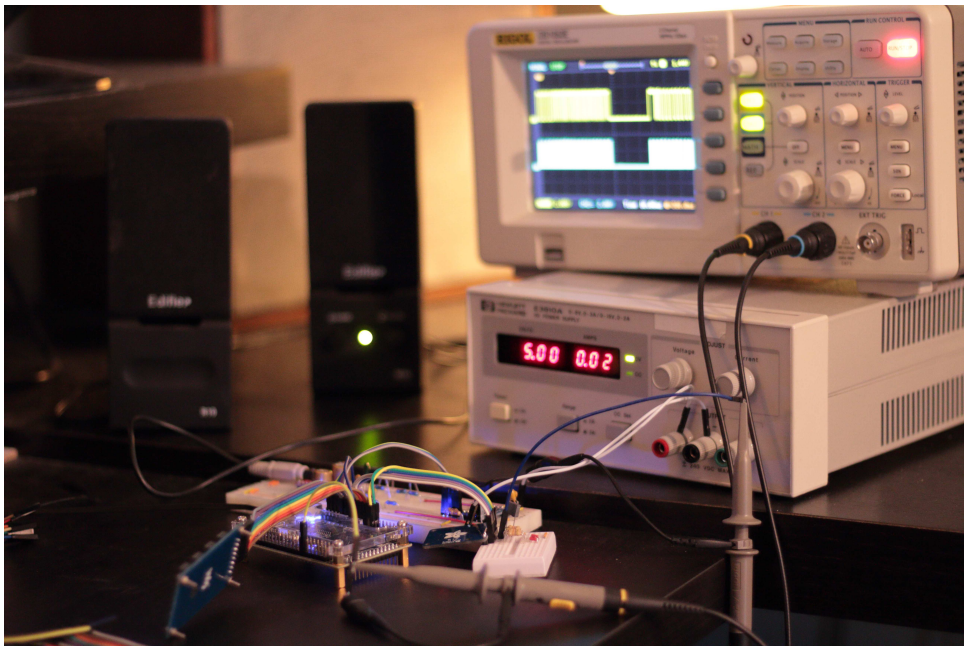
Digital Music Player
ELEX7660 Digital System Design

Nathaniel Rohrick
Preston Thompson



Contents

List of Figures

Listings

1 Introduction

For our Digital System Design course at BCIT we decided to build a digital music player. The system reads digital audio off of an SD card and interfaces to a digital-to-analog converter. While this certainly has been done before (every cell phone sold today is capable of this!) we wanted to design the digital hardware from scratch and learn about SD cards and digital audio.

In addition, we started work on a class-D audio amplifier to go along with the digital music player. The inspiration originally was to have the FPGA drive the H-bridge of the audio amplifier directly however we found out that the speed requirements to do this digitally were not possible. In light of this, the amplifier became a side project of the music player. We have included it in this report for posterity.

Work was also started on a FAT32 firmware driver and but due to time constraints it was not integrated with the rest of the system.

2 Background

2.1 Digital Audio

The audio format used in this project follows the Compact Disc Digital Audio (CDDA) standard for audio stored on CDs. The audio encoding used is 2-channel signed 16-bit PCM with a sample rate of 44.1 kHz. For simplicity, we opted to use little endian byte ordering to store the samples as the Nios II processor is little endian.

The music used to demonstrate this project came originally from MPEG-3 encoded audio files (`.mp3`). The MPEG-3 audio first needed to be decoded for use with the music player. To do this, a Linux utility (`mpg123`) was used to decode the audio and save it in the Waveform Audio File Format (`.wav`). Afterwards, another Linux utility (`sox`) was used to convert the `.wav` files to the raw audio format described above.

3 System Architecture

The system consisted of a Nios II soft-core processor and three custom memory-mapped peripherals for it: the `dac`, the `spi`, the `keypad` and the `ledc` modules.

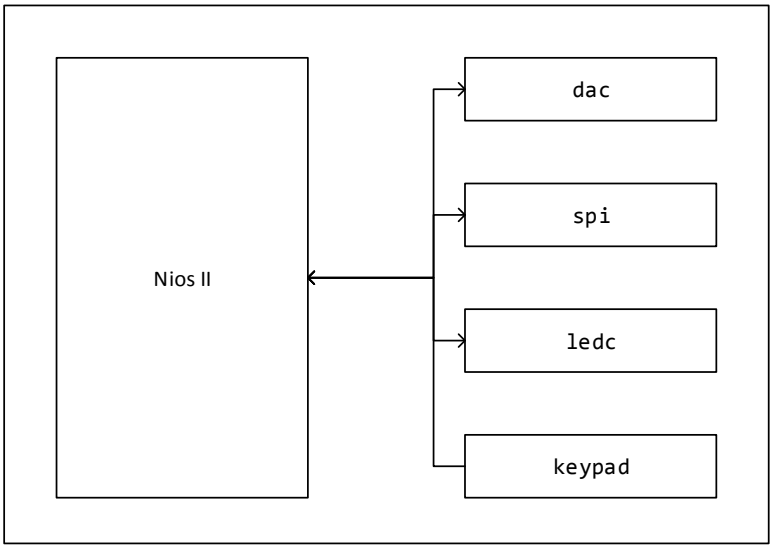


Figure 1: System block diagram

3.1 Digital-to-Analog Conversion

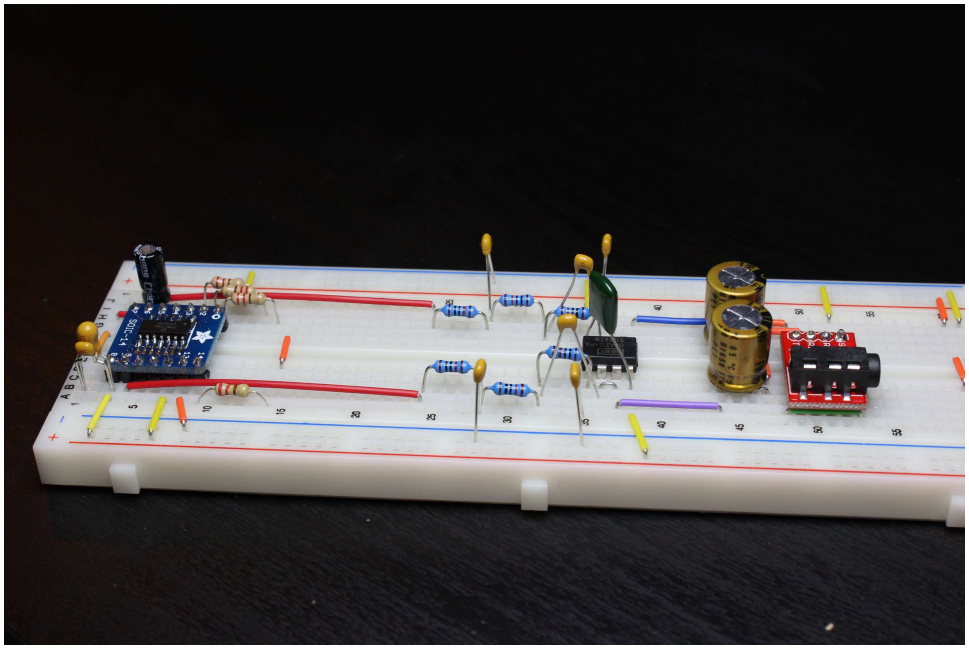


Figure 2: Breadboarded DAC circuit

The DAC used is a Burr-Brown PCM1725 Stereo Audio 16 bit 96 kHz DAC. The DAC only came from Digikey in a SOIC-14 package, so a SOIC-14 breakout board from Adafruit was used to connect it to the breadboard.

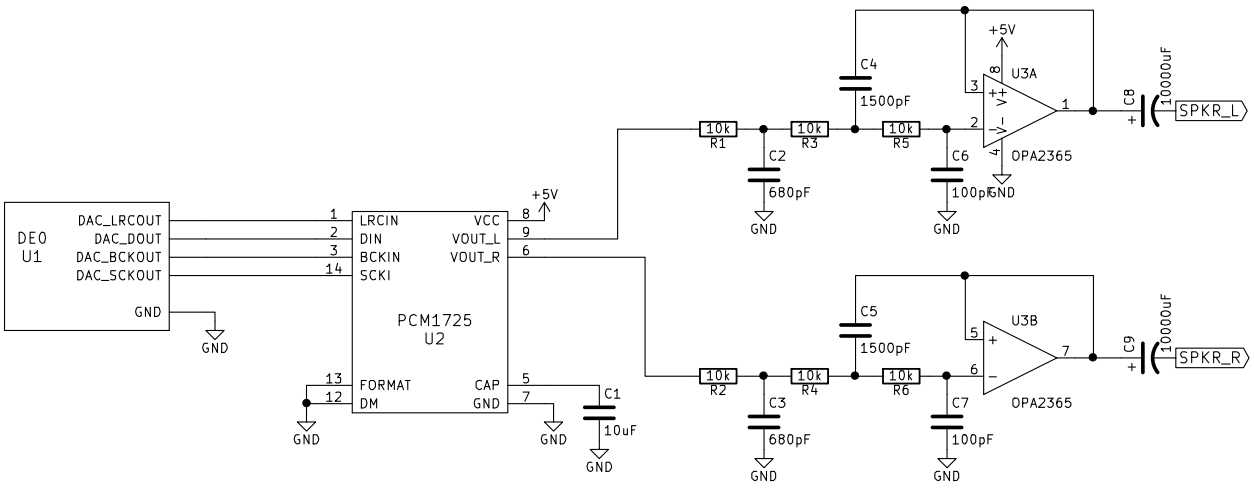


Figure 3: DAC circuit schematic

3.1.1 `dac` CPU Peripheral

The PCM1725 DAC supports I^2S as well as having its own digital interface described in its datasheet. Fortunately this digital interface is simple enough to be able to interface to using nothing but the timing diagram. The only detail that was not mentioned in the datasheet is that the DAC uses signed integers.

A complication we faced was that the sample rate of the music was not an integer divisor of the FPGA's system clock. Even using one of the available PLLs on the FPGA would not get an accurate enough clock because of the large mulitiplier and divisor. To solve this problem, we devised a method where the DAC chip's clock signal's period was constantly modified to average out to make exactly 44.1 kHz playback rate.

The `dac` peripheral has a FIFO in the form of a ring buffer for storing the samples. From the CPU's perspective there is only one register to write to, `dac_sample`. When the CPU writes to `dac_sample`, the `dac` peripheral puts the sample onto the ring buffer. The format of the data written to `dac_sample` is the 16-bit left channel sample in bits `[31:16]` and the 16-bit right channel sample in bits `[15:0]`.

3.1.2 Analog Front-End

The output of the DAC needed to be filtered, amplified, and AC-coupled before going out to a pair of headphones or speakers. To do this we used an audio-grade op-amp, the OPA2134PA. It is a dual op-amp circuit in a DIP-8 package. The op-amp was configured to be a 3rd order low-pass filter. The output of the op-amp is AC-coupled before connecting to a 3.5mm TRRS stereo connector.

3.1.3 Class-D Amplifier

The class-D amplifier is a power amplifier topology that boasts the highest power efficiency of all the amplifier classes. It accomplishes this by modulating the input signal to a high frequency pulse-width modulated waveform (PWM). Since PWM is digital in nature, they do not require FETs to be in the linear I-V region. FETs dissipate little power in the saturation conduction region and efficiency is mostly limited by switching losses [1].

To modulate the incoming signal, a simple comparison is made with a triangle wave which acts as the carrier. Below gives a practical implementation of this modulation scheme [2].

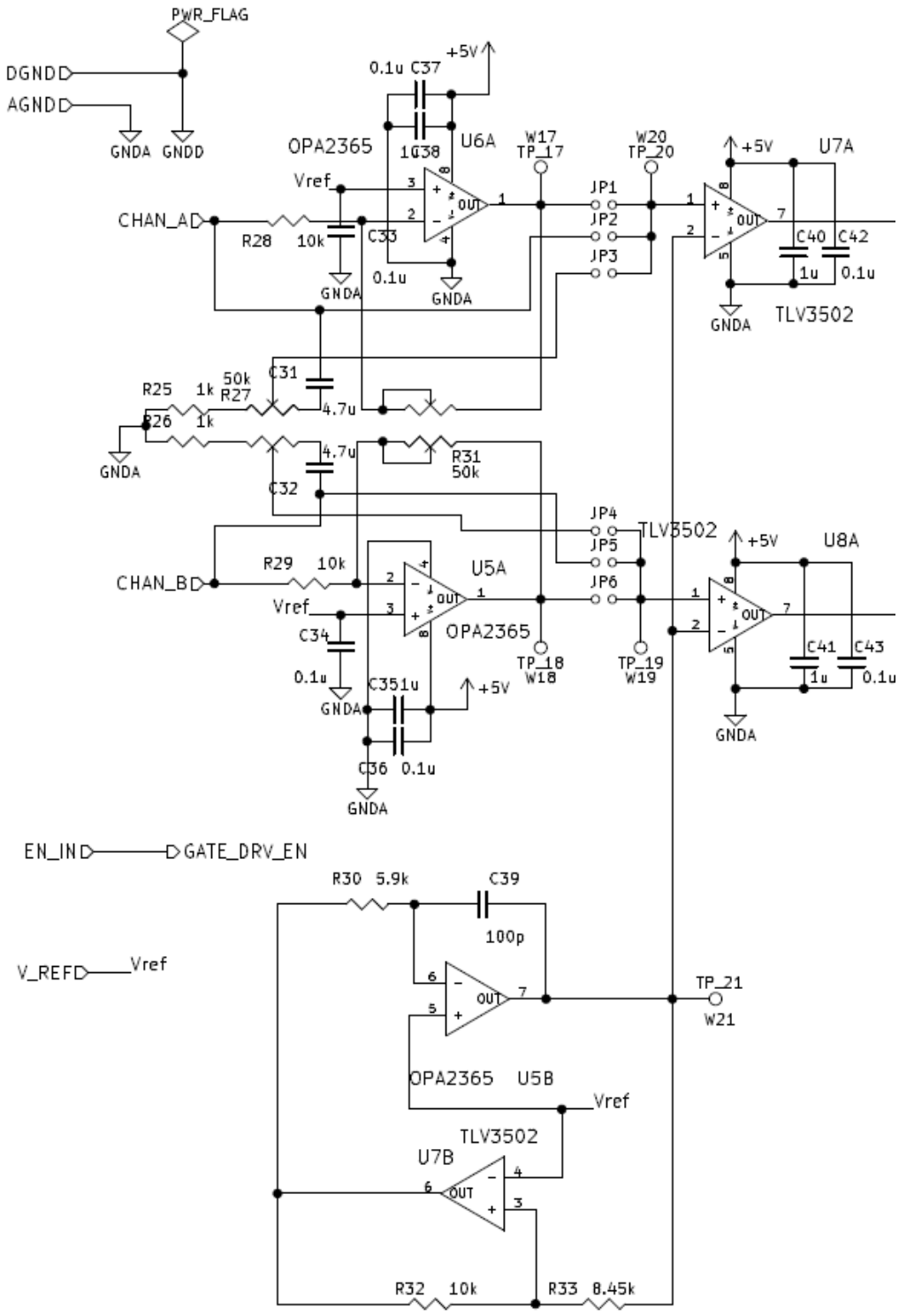


Figure 4: Stereo PWM schematic

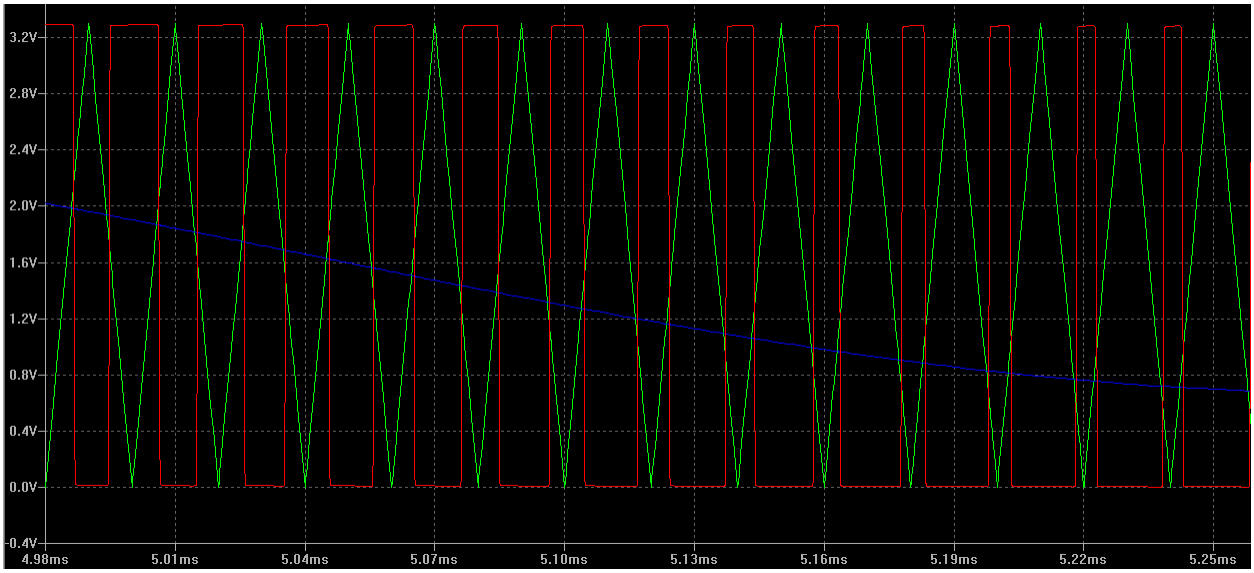


Figure 5: Single channel PWM output

Now we have a digital signal that represents the baseband signal's amplitude in its duty cycle. From this signal, we can drive an inverter topology of our choosing. In this case we've selected the H-bridge topology for higher power amplification for a given supply voltage. Before we can do this, we need complementary PWM signals with deadtime to drive the diagonal legs of the H-bridge. This is accomplished with a simple inverter and a FET based asymmetric RCD delay network fed into a comparator with hysteresis. Below shows such a circuit, its complement has an inverter at the input.

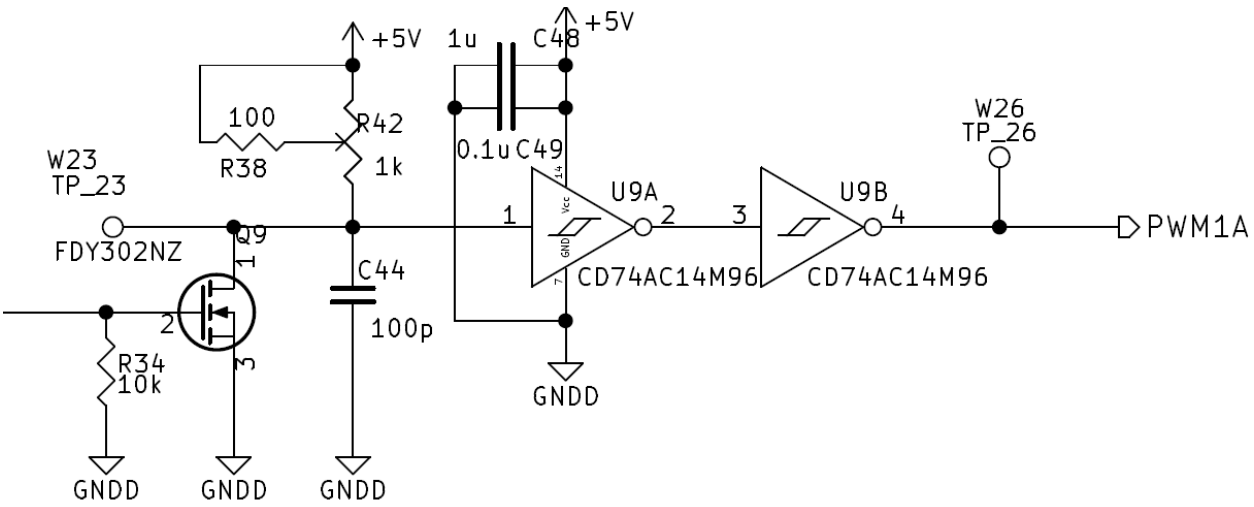


Figure 6: Schematic of complementary PWM generator with deadtime

Lastly, the power stage of the amplifier. The H-bridge is constructed of 4 FETs controlled by PWM. For stereo audio we would require a total of 8 FETs! Some unique challenges are faced by the designer; particularly, driving the relatively large gate capacitances of the FETs

at high frequencies and driving the floating gate of the high side FET. Information can be found at [2]. This is left to the reader to analyze and is out of the scope of this report. A single channel of our design can be seen on the next page.

Demodulation is accomplished by simply filtering off the carrier frequency. In our case it was recommended by [2] to use a second order Butterworth filter with only reactive components to minimize losses.

Additional electronics was needed for the operation of this all-in-one class-D amplifier, particularly the power supply designs, input protection and signal conditioning, jack detect, and DC blocking. Appended to the end of this report is the complete schematic. MATLAB scripts developed for the analysis of the design are also included.

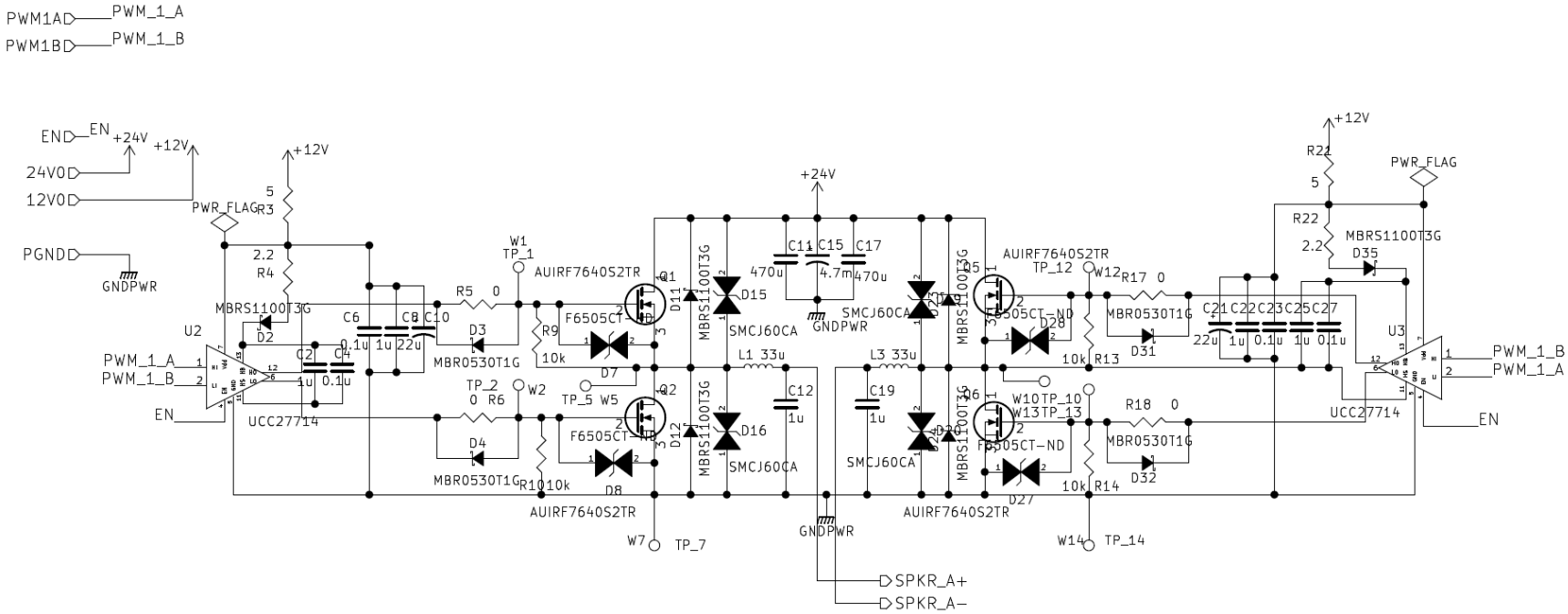


Figure 7: Single channel H-Bridge schematic

3.2 SD Card Data Management

For the implementation of the music player we require a music source to play from. A significant amount of data storage is necessary for this project because we won't have the means to decompress MPEG-3 encoded audio. Therefore we will be using raw 16-bit signed audio with a sample rate of 44.1 kHz. This will also put considerable strain on the communications interface and CPU when managing the immense amount of data.

The SD card interface will be broken down as follows:

- Hardware communications interface `spi.sv`
- Firmware driver for communications interface `sd_card.c`
- The main loop in `main.c`

The SD card used is the SDHC variant card which allows for capacities up to 32GB. We will be utilizing a micro SD card breakout board by Adafruit which will be powered and controlled by the DE0 Nano.

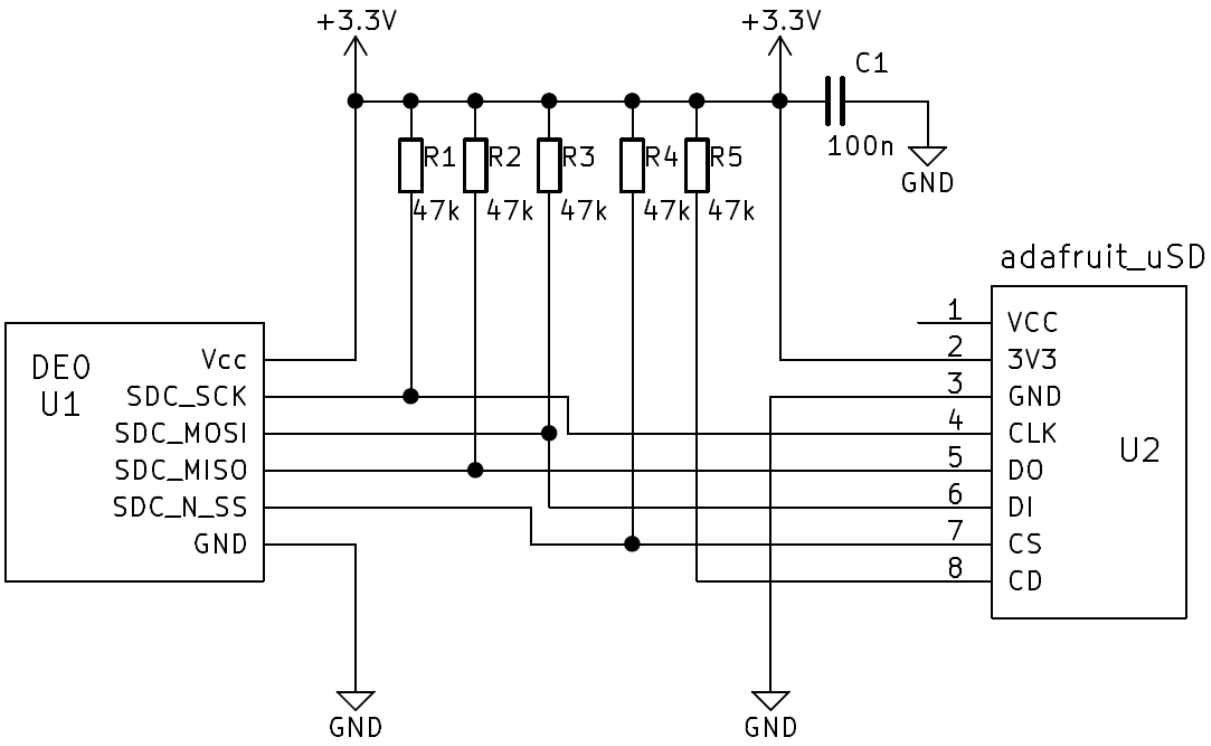


Figure 8: SD card circuit schematic

Below we will detail the requirements and design steps needed to realize the above components of the SD card interface.

3.2.1 `spi` CPU Peripheral

SD cards can be communicated with using two protocols: a serial peripheral interface (SPI) and a proprietary 5 wire synchronous communications interface. We will be using SPI since the latter requires licensing from the SD Association. From [3], the SD card requires the following hardware parameters:

- Variable SCK control
- 8-bit data packets
- Type 0 SPI mode
- Send and receive capability
- Busy status flag
- CPU integratable

Because we have incorporated the Nios II softcore processor in our design, it allows us to have direct control over our `spi` peripheral by connecting control registers to the Avalon MM bus. Given the CPU's ability to read/write to data registers, we can set status flags for the CPU to check and data/control registers to operate the `spi` peripheral.

The entire `spi` peripheral can be broken down into 3 states represented below as a state diagram:

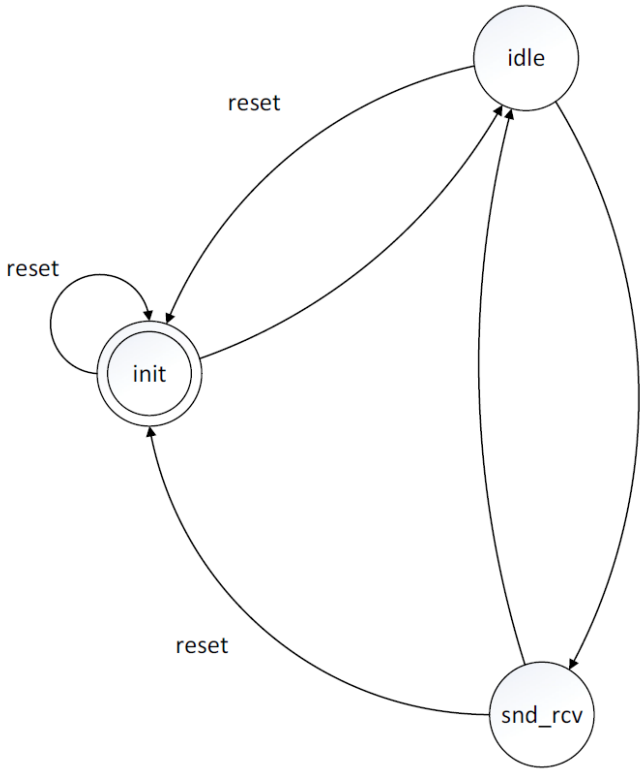


Figure 9: State-machine for `spi` peripheral

At start-up the module enters the `init` state which initializes all the design registers and settings to a default value. This involves setting a `clk` divider to 2 (max `sck` speed) and pre-loading the corresponding counters to drive `sck`.

Additionally, any state (including `init`) will transition to init when the `reset` signal is asserted. After a single `clk` cycle the state machine transitions to `idle` where the peripheral is continuously loading valid count and I/O pin states, as well as handling CPU requests to read/write data to its registers.

When the CPU requests to write data to the `rcv_buf` the next state becomes `snd_rcv`. This state pulls the busy flag high and sends/receives data through the `mosi` and `miso` signals. This is accomplished using a shift register designed using pointer arithmetic in hardware. When exactly 8 falling edges of `sck` have been sent over `mosi`, the next state becomes `idle` and the busy flag is deasserted.

3.3 `sdc` Driver for SD Card Using the `spi` Peripheral

To simplify the configuration of the `spi` peripheral and initialization process of the SD card, we opted for a firmware solution for this task. We've developed an API for the SD card constituting the following functions:

- `bool sdc_init();`
- `bool baud_set(uint8_t div);`
- `bool sdc_read(uint32_t addr, uint8_t len, uint8_t* const buf);`
- `bool sdc_send_cmd(const uint8_t cmd, const uint32_t payload, const uint8_t resp, uint8_t* const resp_buf);`

The `baud_set()` function takes an argument `div` and simply checks whether the `spi` interface is busy by parsing the busy flag, if not busy the CPU writes `div` to the divider register in the `spi` peripheral.

The `sdc_send_cmd()` follows a command and response sequence specified by the SD card specification found in [3]. It starts by pulling `n_ss` line low and sends a command byte, a 4 bytes payload, and a CRC byte over `mosi`. The CRC byte is optional in SPI mode according to [3]. This function then commands the `spi` peripheral to send dummy bytes (0xFF) to shift data in from the SD card over `miso`. The beginning of a response is indicated by a start bit. Once a response is received, it checks if the command was accepted and if the SD card is functioning properly.

The SD card always responds to a command with a defined response which can vary between different commands. The exact expected response has to be specified by the caller but could conceivably be decoded with a simple lookup table.

The initialization function `sdc_init()` initializes the sd card following the procedure outlined by [3]. It utilizes the `sdc_send_cmd()` and `baud_set()` functions to operate.

Finally the `sdc_read()` function is used to read a set amount of blocks from the SD card. It takes an address, expected length of response in blocks and a pointer to a buffer to put the data into. This function utilizes `sdc_send_cmd()` to initialize the data read then iterates a loop that loads the response into `buf`. We will be transferring the contents of `buf` into the `dac` ring buffer.

3.4 `keypad` CPU Peripheral

We required a means for the user to control audio playback. We designed a `keypad` module that detects a keypress and decodes the value and stores it into a register that can be read by the CPU. The internal circuitry of the keypad is a 4x4 matrix of normally open switches with each row and column connected to a pin header. We set up the DE0 to output to the rows and have the columns connected as inputs with pull-up resistors enabled. To decode a keypress we simply pull a row low and check the states of the columns. If any of the column are low, continuity to ground has been made and the exact button pressed can be determined from the low row/column combination.

To ensure reliability a hardware debouncer was setup to store the decoded value on the positive edge of clk immediately after a keypress is detected then wait 100 ms before checking again. The state machine is illustrated below:

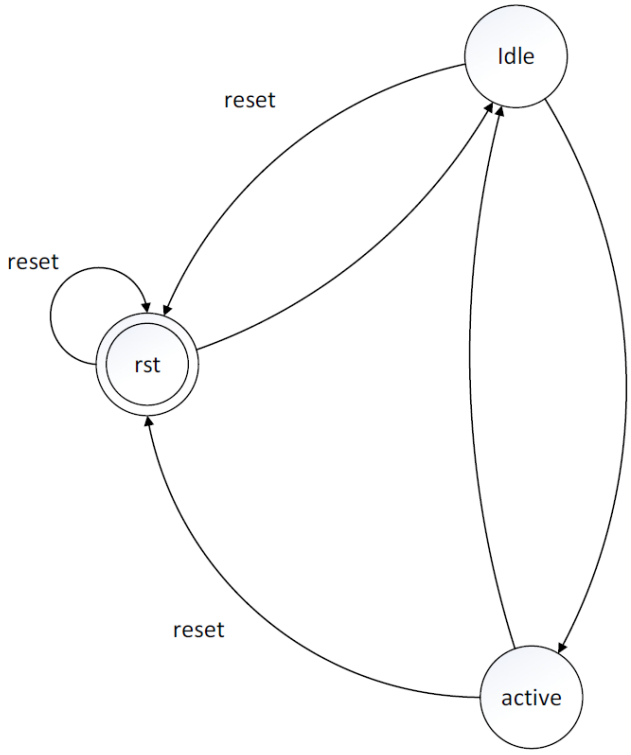


Figure 10: State-machine for the `keypad` peripheral

3.5 `keypad` Driver

A simple API was created for the `keypad` module, the `keypad_get_button()` function, which parses the register that stores the latest keypress and returns an integer corresponding to the key pressed. An enumeration is defined in a header file that details the return value. This module is designed such that the main program should constantly be polling with this function to detect any changes in keypress states. The physical hardware configuration is as simple as connecting the 8 pin headers to the keypad board.

3.6 `ledc` CPU Peripheral

The `ledc` is the LED controller peripheral. Initially it was designed to control the LEDs on the DE0 Nano board for debugging use, however later the `ledc` was extended to include GPIO functionality. The GPIO functionality made debugging much easier by allowing us to trigger an oscilloscope on events originating in firmware, and allowing us to determine the execution time of firmware subroutines.

3.7 Firmware

The firmware for the music player consisted of an initialization routine and a main program loop that checks the keypad for events, transfers audio data to the `dac` if there is room and handles switching to the next track if the currently playing track has reached its end.

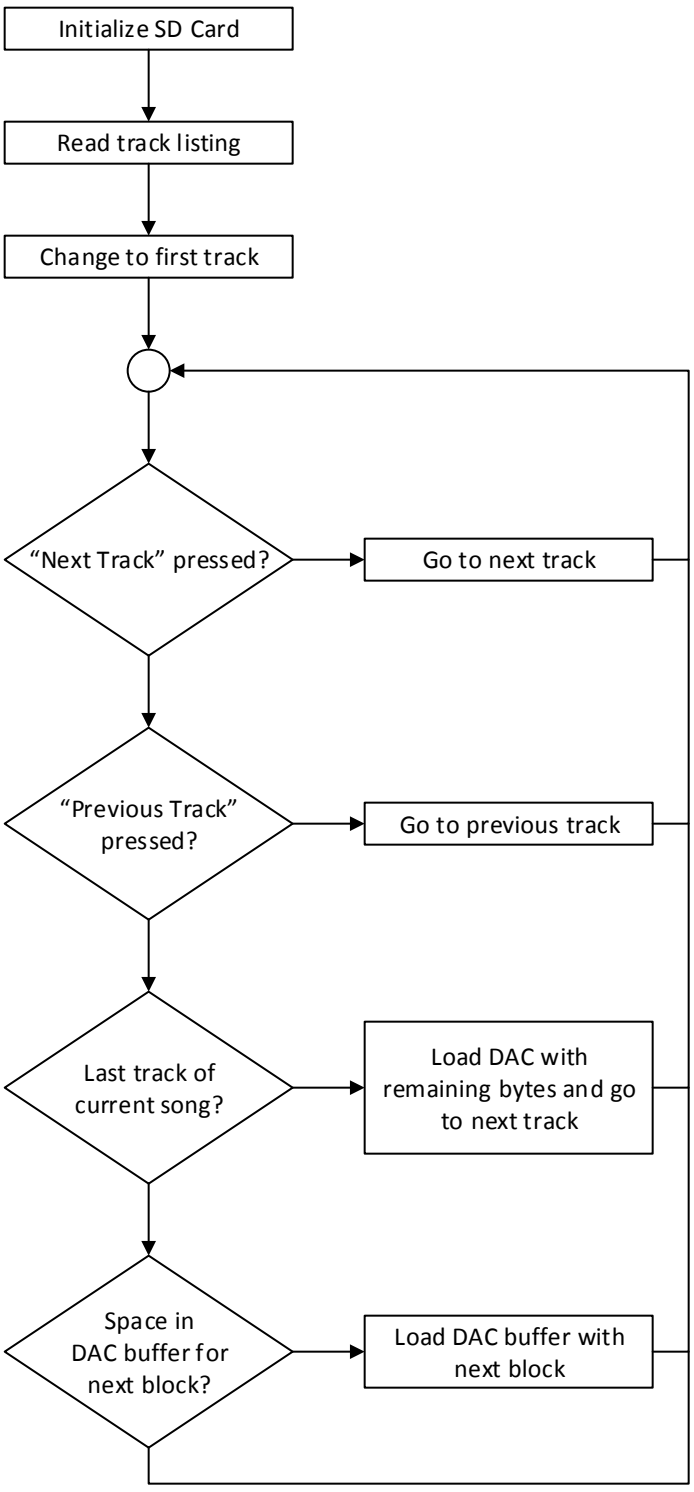


Figure 11: Flowchart of firmware's main loop

3.7.1 SD Card Format

After the music has been prepared as raw audio files, a Python script running on a Linux PC is used to prepare the SD card image. At the begining of the image, in the first block (512

bytes), a table exists that contains information for each of the music tracks. For each track two pieces of information is recorded in the table: the file size in bytes, and which 512-byte block the track starts in. Finally, the UNIX utility `dd` is used to write the image to the SD card.

Each track starts on its own block. The space in between tracks is padded with zeros until the next track starts on a 512-byte block boundary. Below is an example of the SD card layout for two tracks.

Byte address	Description
0x0000	Track 0 start block number
0x0004	Track 0 size
0x0008	Track 1 start block number
0x000c	Track 1 size
0x0010	Zeros
...	
0x0200	Beginning of track 1
...	
0x29f4	Last byte of track 1
0x29f8	Zeros
...	
0x2a00	Beginning of track 2

3.7.2 FAT32 Driver

A basic read-only FAT32 implementation was created but due to time constraints it was not integrated into the project. It is capable of getting a directory listing of the root directory and reading files in the root directory. It is a primitive driver and is not optimized for performance. The source code listing is attached in the appendice section.

An important performance improvement that could be made for the FAT32 driver is to store the FAT in RAM instead of constantly re-reading the FAT to find out where the next sector of a file is.

4 Challenges

4.1 `dac` Ring Buffer

During initial debugging, we experimented with changing the `dac`'s FIFO buffer size. To do this, two things must be changed: the size of the `dac` buffer, and the limits of the read and write pointers for managing the ring buffer. In increasing the size of the `dac` buffer, we had to increase the limits of the read and write pointers. Once the initial issues had been resolved, we reduced the `dac` buffer size but forgot to change the limits of the read and write pointers.

The symptoms were that the audio would repeatedly click and stutter. This is because the write and read pointers that are supposed to index samples in the buffer were pointing beyond the limit of the buffer. Eventually the pointers would overflow and point within the buffer again and audio would resume properly.

To debug this issue, we created a GPIO module that allowed the CPU to toggle pins from firmware. By correlating the toggling of the pins with the audio glitches we were able to determine where the CPU was spending most of its time. This allowed us to rule out the SD card as the source of the glitches and revealed that the bug was in the `dac` module.

4.2 Keypad

During the initial bring-up phase of the `keypad` module we had much difficulty in getting all the rows to work on the keypad. After exhausting all other options over the course of several hours we finally decided to do a continuity test on the ribbon cable that connects the keypad to the FPGA. It turned out that one of the wires was broken. Continuity testing is important when using cheap ribbon cables.

4.3 SD Card

After all of our modules were designed and debugged, we encountered performance issues when reading information from the SD card. The maximum `sck` rate that we could reliably operate the SD card at was 12.5 MHz. Without a dedicated DMA module we optimized the firmware to improve efficiency on the firmware side. The minimum required baud rate needed to run 16-bit stereo audio is 1,411,200 bits per second. While this is significantly slower than SCK, the SD card latency coupled with ferrying uncompressed audio data with less than optimal code was introducing too much overhead. This was giving us a baud rate of 1,120,000 bits per second.

The solution was to simplify low level loops and pre-calculate constants used for the duration of the loop. Additionally, we increased the compiler optimization settings to maximum. With these changes the data rate exceeded 3,840,000 bits per second!

5 Conclusion

Overall, we saw many fine examples of digital system design. We learned more than expected. The `sdc` implementation proved to be the most complex module of the project. Overall, the project was a success and concluded with a working demonstration to the rest of the class.

Initially we were concerned about signal integrity issues surrounding the connection between the FPGA and the DAC and SD card. While error rates at 25 MHz were unnacceptable, it not an issue at all at 12.5 MHz. Poor signal integrity would have limited our bitrate and potentially caused the whole project to fail if it was not fast enough. In our testing we did not detect any bit errors at this speed, although they could have been possible.

Recommendations for future work include finishing and validating the FAT32 filesystem implementation, implementing a real-time LCD-based frequency spectrum display, implementing an audio equalizer for boosting and attenuating frequency bands, and construct the class-D amplifer.

References

[1] 2017. [Online]. Available: http://www.irf.com/product-info/audio/classdtutorial.pdf

[2] 2017. [Online]. Available: http://www.ti.com/lit/ug/slau508/slau508.pdf

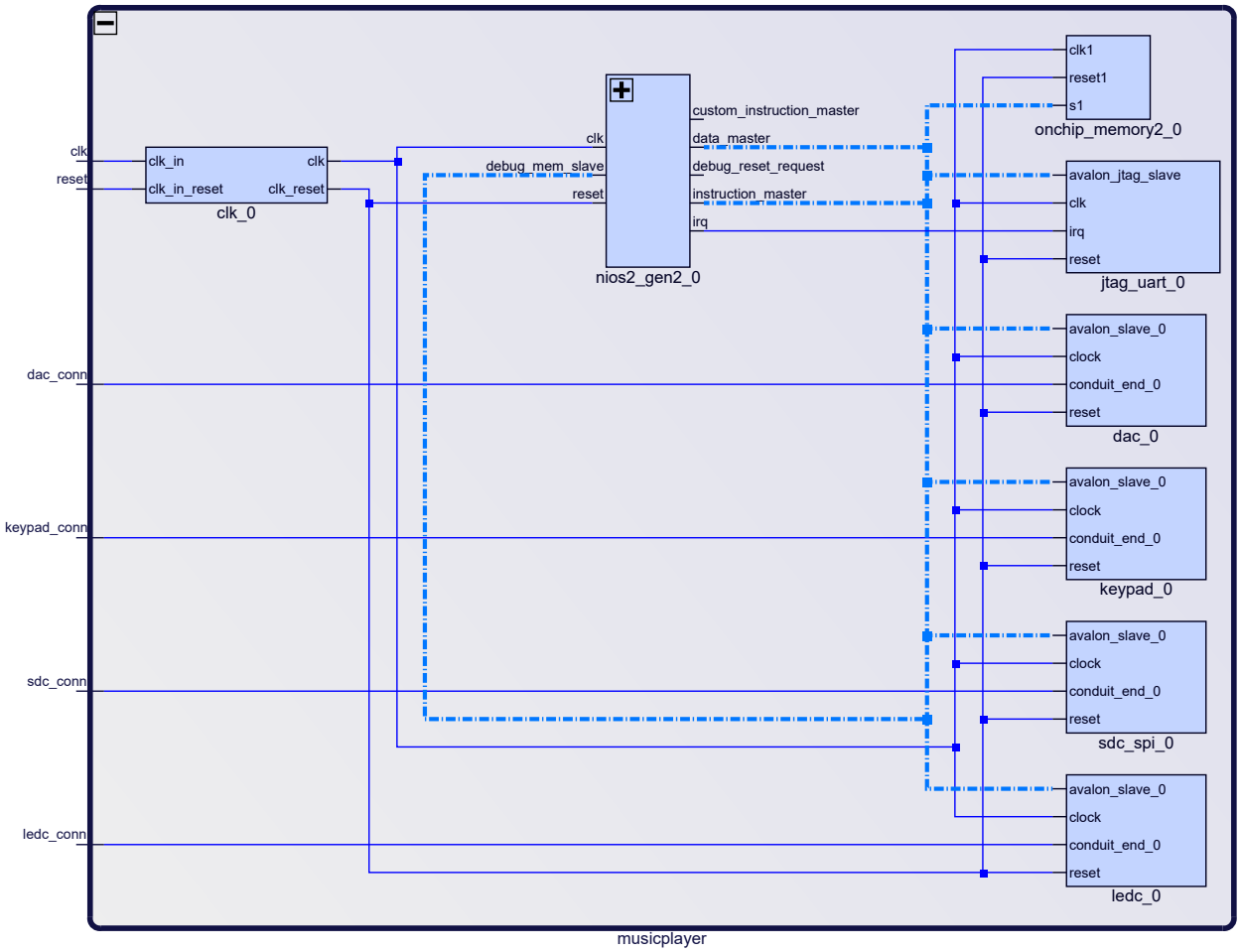
[3] F. Foust, "Secure digital card interface for the msp430," 2017. [Online]. Available: http://alumni.cs.ucr.edu/~amitra/sdcard/Additional/sdcard_appnote_foust.pdf

[4] Terasic Inc, "De0 nano user manual," 2017.

[5] E. Casas, "Digital system design," 2017.

6 Appendice

6.1 QSYS Circuit Schematic



6.2 Class-D Amplifier Circuit Schematic

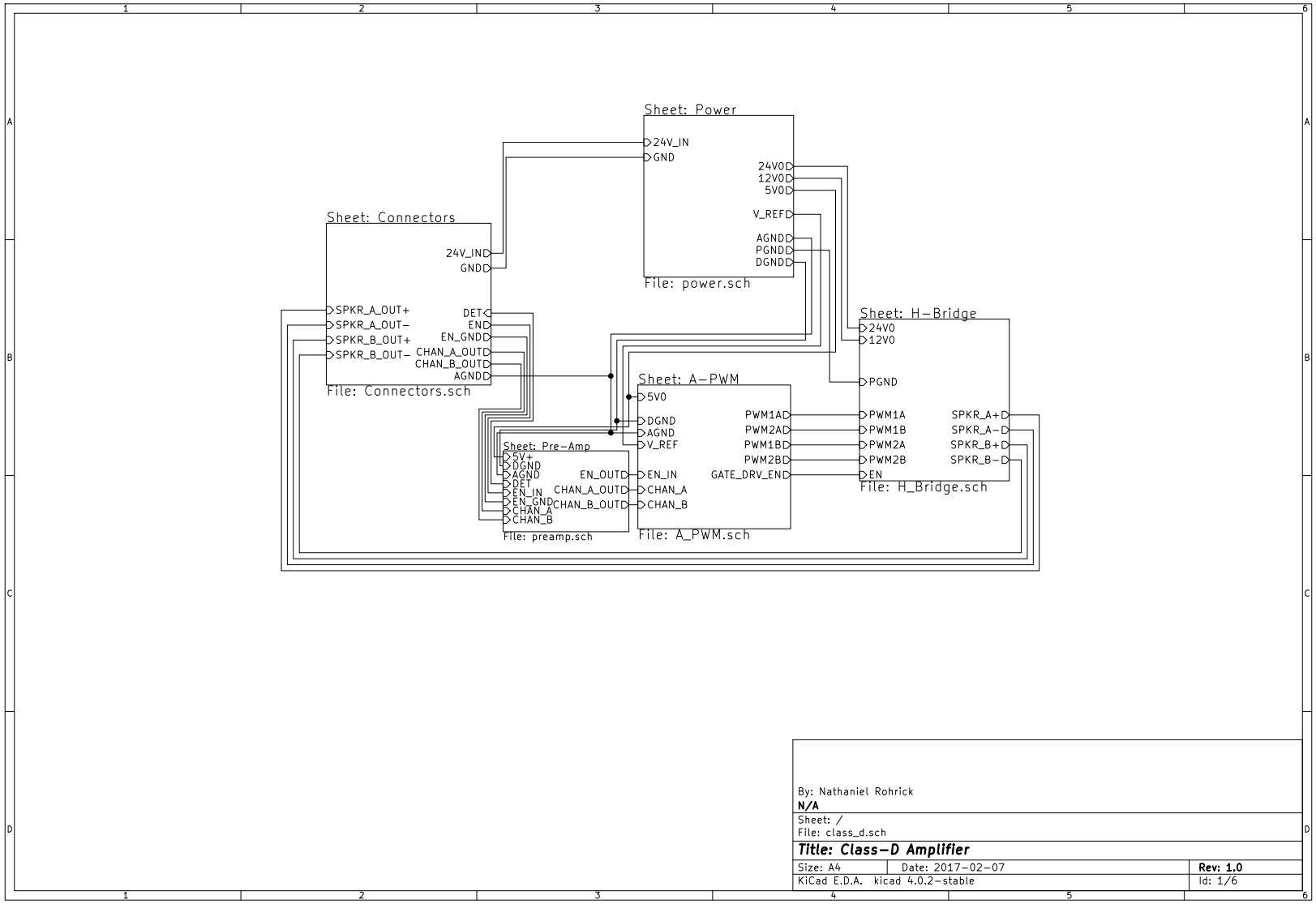


Figure 12: Class-D amplifer schematic pg. 1/6

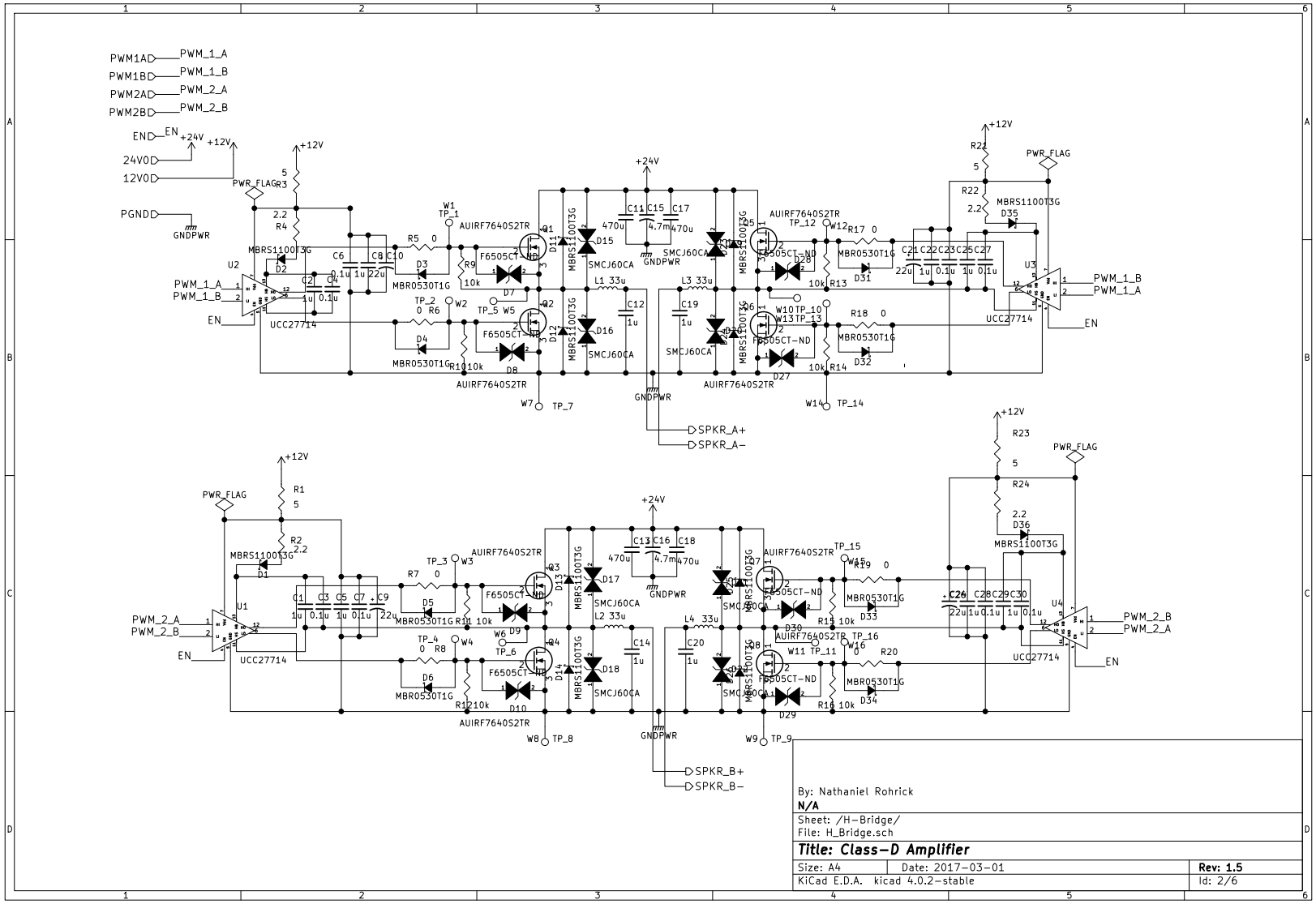


Figure 13: Class-D amplifer schematic pg. 5/6

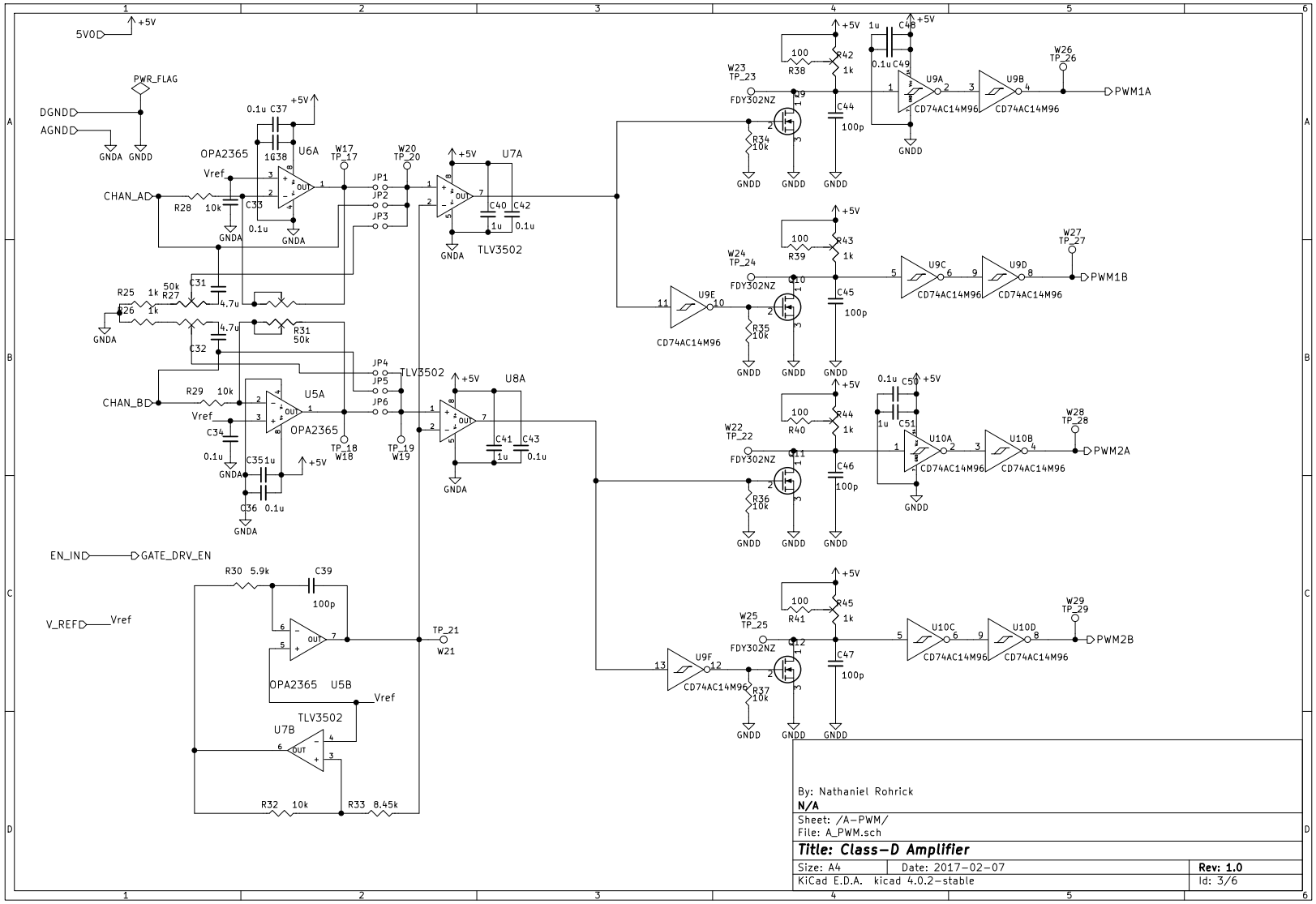


Figure 14: Class-D amplifer schematic pg. 3/6

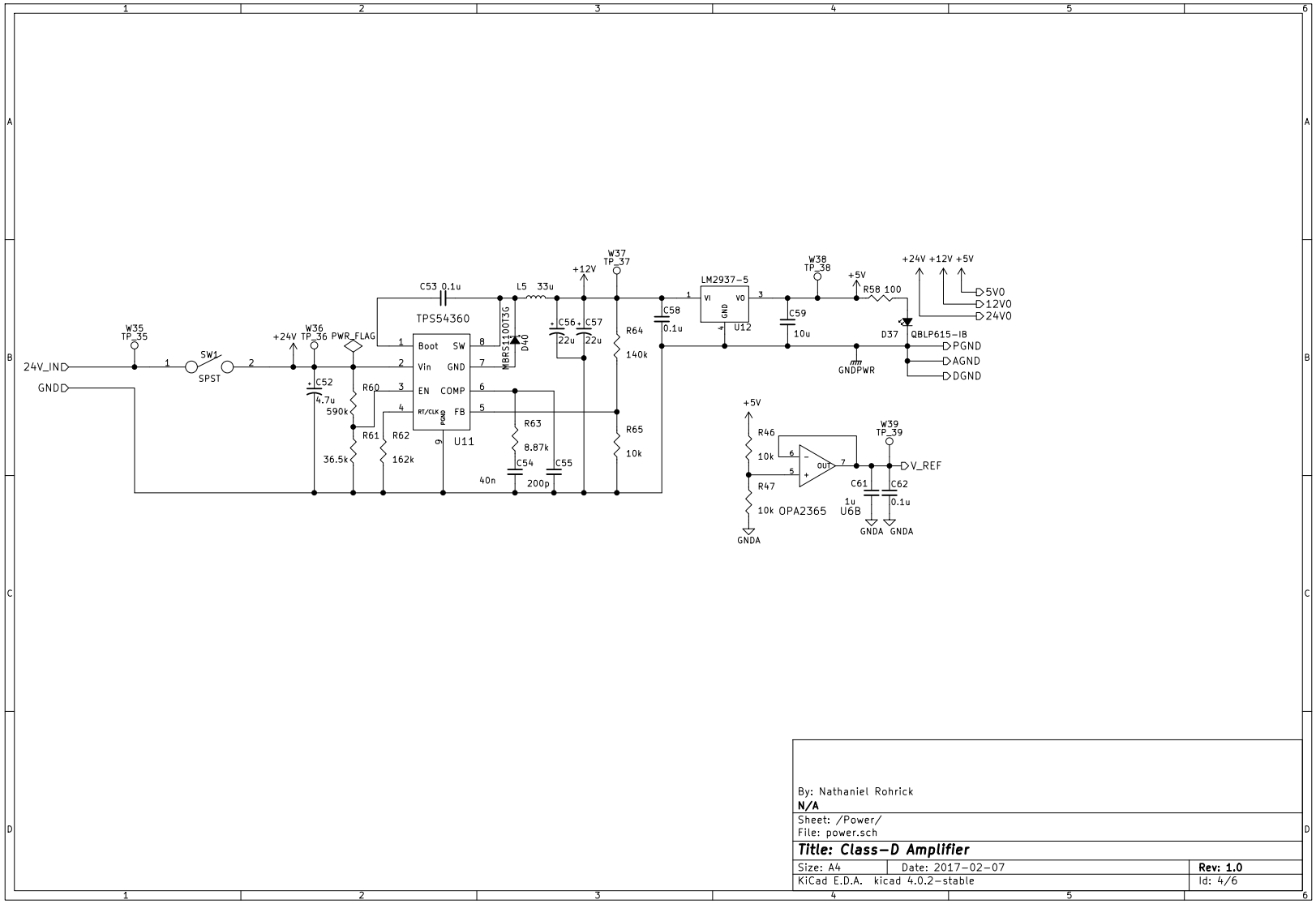


Figure 15: Class-D amplifer schematic pg. 4/6

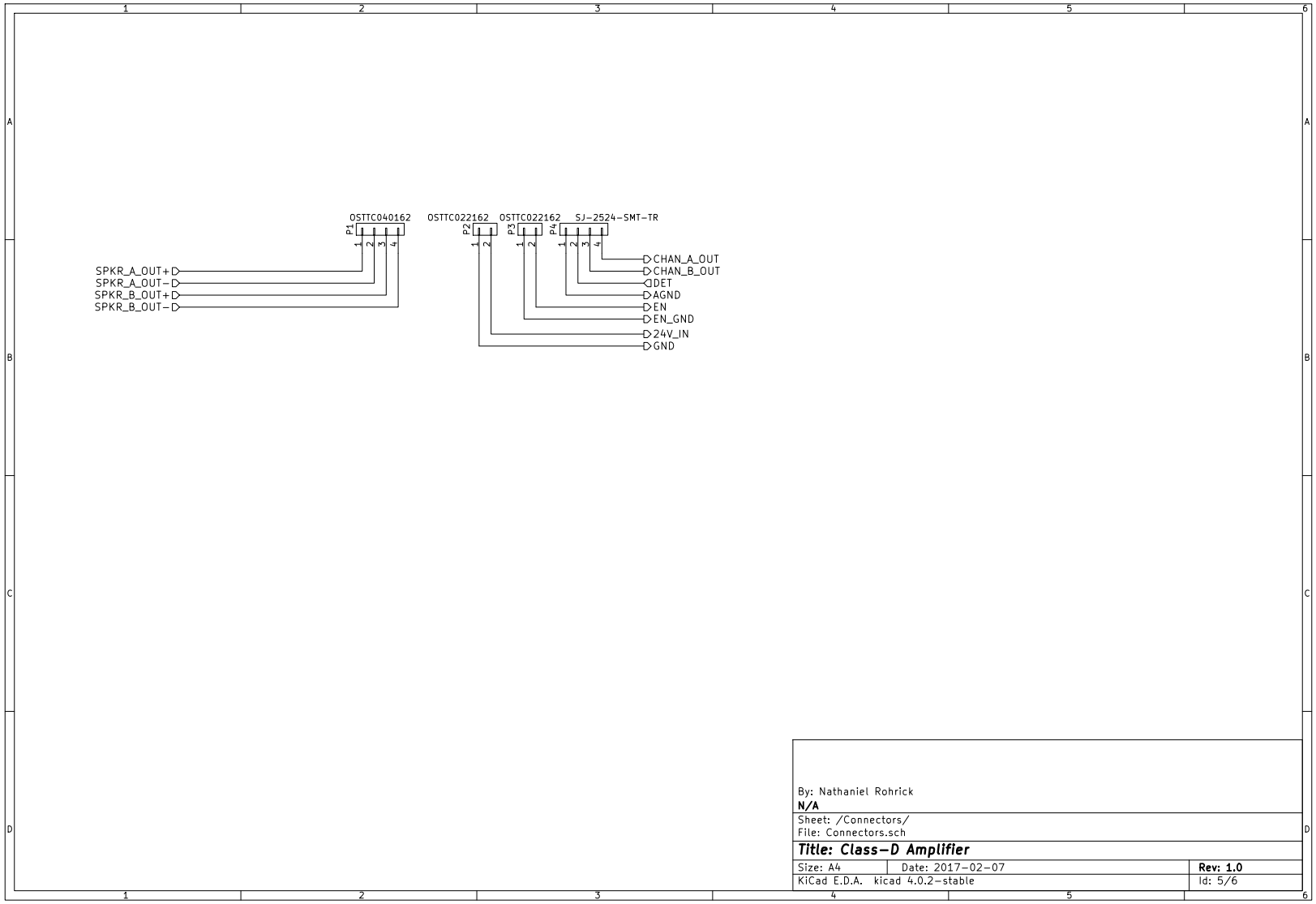


Figure 16: Class-D amplifer schematic pg. 5/6

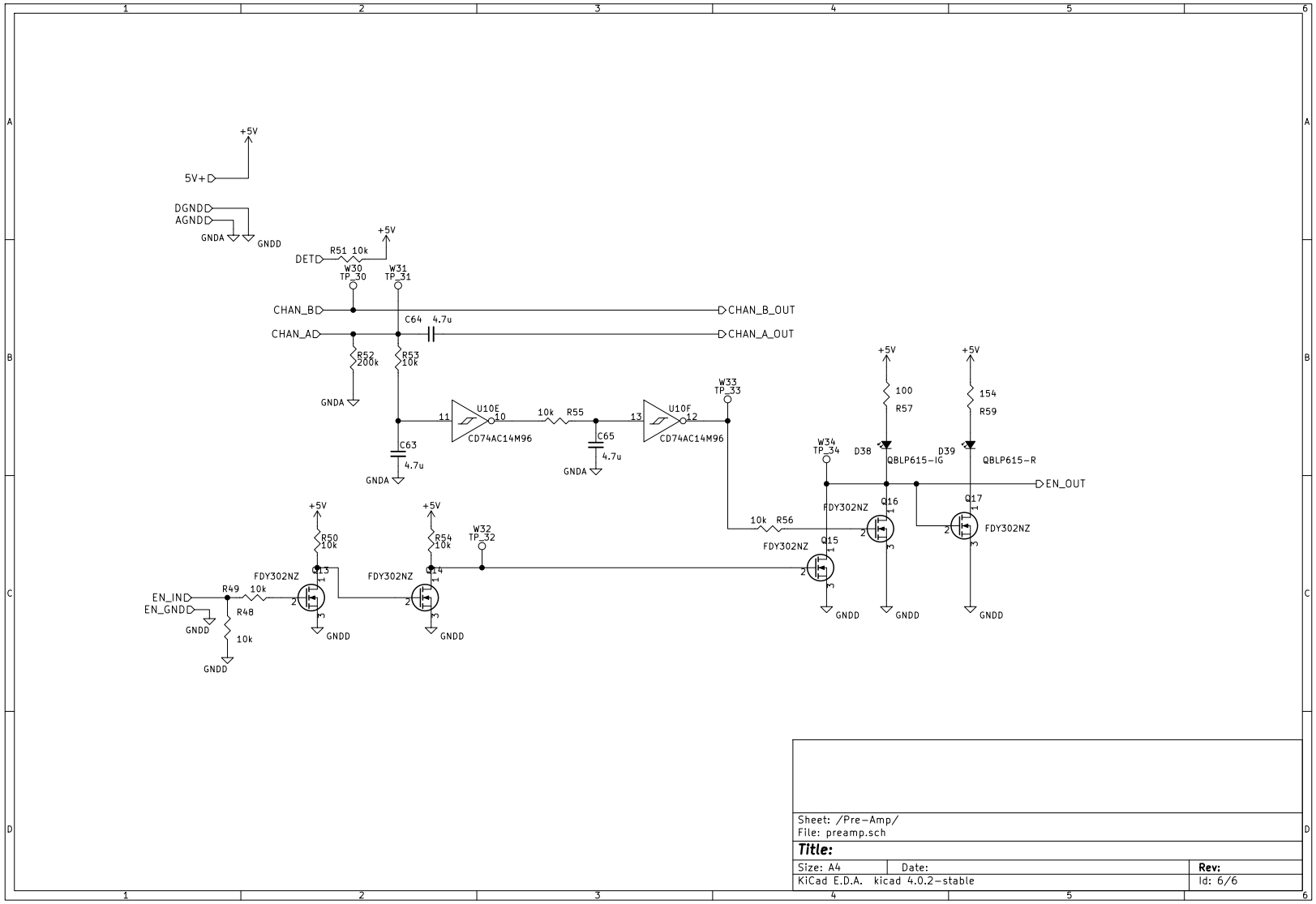


Figure 17: Class-D amplifer schematic pg. 6/6

6.3 Software Listings

Listing 1: musicplayer_top.sv

```
// Author: Preston Thompson & Nathaniel Rohrick
// Date: April 7, 2017

module musicplayer_top (
    input logic CLOCK_50,
    output logic [15:0] LED,
    input logic [1:0] KEY,
  input logic SDC_MISO,
  output logic SDC_SCK,
  output logic SDC_MOSI,
  output logic SDC_N_SS,
    output logic dac_dout,
    output logic dac_lrcout,
    output logic dac_sckout,
    output logic dac_bckout,
    (* altera_attribute = "-name WEAK_PULL_UP_RESISTOR ON" *)
  input logic [3:0] keypad_kpr,
  output logic [3:0] keypad_kpc
);

    musicplayer u0 (
        .clk_clk(CLOCK_50),
        .reset_reset_n(KEY[0]),
        .ledc_conn_leds(LED),
        .sdc_conn_miso(SDC_MISO),
        .sdc_conn_mosi(SDC_MOSI),
        .sdc_conn_sck(SDC_SCK),
        .sdc_conn_n_ss(SDC_N_SS),
        .dac_conn_dout(dac_dout),
        .dac_conn_lrcout(dac_lrcout),
        .dac_conn_sckout(dac_sckout),
        .dac_conn_bckout(dac_bckout),
    .keypad_conn_kpc(keypad_kpc),
    .keypad_conn_kpr(keypad_kpr)
    );

endmodule
```

Listing 2: ledc.sv

```
// Author: Preston Thompson
// Date: April 7, 2017

module ledc (
    input logic reset,
    input logic clk,

    input logic avs_read,
    input logic avs_write,
    input logic [15:0] avs_writedata,
```

```
    output logic [15:0] avs_readdata,

    output logic [15:0] leds
);

    always_ff @(posedge clk) begin
        if (reset) begin
            leds <= '0;
        end else begin
            if (avs_write) begin
                leds <= avs_writedata;
            end else if (avs_read) begin
                avs_readdata <= leds;
            end
        end
    end

endmodule
```

Listing 3: dac.sv

```
// Author: Preston Thompson
// Date: April 7, 2017

module dac #(
    // 50 MHz / (15625/7056) = 22.5792 MHz
    parameter SCK_NUMERATOR = 16'd15625,
    parameter SCK_DENOMINATOR = 16'd7056,
    parameter BUFFER_SIZE = 2048
) (
    input logic reset,
    input logic clk,

    output logic dout,
    output logic lrcout,
    output logic sckout,
    output logic bckout,

    input logic avs_write,
    input logic avs_read,
    input logic [31:0] avs_writedata,
    output logic [31:0] avs_readdata,
    input logic [3:0] avs_address
);

    logic [31:0] sample_buf[BUFFER_SIZE-1:0];
    logic [10:0] sample_read_ptr;
    logic [10:0] sample_write_ptr;

    logic [15:0] sck_count;
    logic [15:0] sck_count_next;
    logic [2:0] bck_count;
    logic [4:0] lrcout_count;
    logic [31:0] sample;
```

```
34      logic [4:0] dout_ptr;
35      enum logic {increment, overflow} op;
36
37      always_ff @(posedge clk) begin
38
39          if (reset) begin
40
41              sck_count <= '0;
42              bck_count <= '0;
43              lrcout_count <= '0;
44              lrcout <= '1;
45              sckout <= '0;
46              bckout <= '0;
47              dout_ptr <= '0;
48              sample_write_ptr <= '0;
49              sample_read_ptr <= '0;
50
51          end else begin
52
53              // if the ring buffer is empty, output zeros and nop
54              if (sample_read_ptr != sample_write_ptr) begin
55                  dout <= sample[5'hf - dout_ptr];
56                  sck_count <= sck_count_next;
57              end else begin
58                  dout <= '0;
59              end
60
61              // handle any reads from the CPU
62              if (avs_read) begin
63                  unique case (avs_address)
64                      4'h0: begin
65                          avs_readdata[31:16] <= '0;
66                          avs_readdata[15:0] <= sample_read_ptr;
67                      end
68                      4'h1: begin
69                          avs_readdata[31:16] <= '0;
70                          avs_readdata[15:0] <= sample_write_ptr;
71                      end
72                      4'h2: begin
73                          avs_readdata <= sample_buf[sample_write_ptr];
74                      end
75                  endcase
76              end
77
78              // handle any writes from the CPU
79              else if (avs_write) begin
80                  unique case (avs_address)
81                      4'h0: begin
82                          sample_read_ptr <= avs_writedata[15:0];
83                      end
84                      4'h1: begin
85                          sample_write_ptr <= avs_writedata[15:0];
86                      end
87                      4'h2: begin
```

```
                        sample_buf[sample_write_ptr] <= avs_writedata;
                        sample_write_ptr <= sample_write_ptr + 16'd1;
                    end
                endcase
            end

            // only interact with the DAC if there's samples in the ring buffer
            if (op == overflow && sample_read_ptr != sample_write_ptr) begin
                sckout <= ~sckout;
                bck_count <= bck_count + 3'b1;
                if (bck_count == '1) begin
                    lrcout_count <= lrcout_count + 5'b1;
                    bckout <= ~bckout;
                    if (bckout) begin
                        dout_ptr <= dout_ptr + 5'b1;
                    end
                    if (lrcout_count == '1) begin
                        lrcout <= ~lrcout;
                    end
                end
            end

            if (dout_ptr == '1 &&
                bck_count == '1 &&
                op == overflow &&
                lrcout_count == '1
            ) begin
                sample_read_ptr <= sample_read_ptr + 16'd1;
                sample <= sample_buf[sample_read_ptr + 16'd1];
            end

        end
    end

    always_comb begin

        if (sck_count >= (SCK_NUMERATOR - SCK_DENOMINATOR)) begin
            sck_count_next = sck_count - (SCK_NUMERATOR - SCK_DENOMINATOR);
            op = overflow;
        end else begin
            sck_count_next = sck_count + SCK_DENOMINATOR;
            op = increment;
        end

    end

endmodule
```

Listing 4: dac_tb.sv

```
// Author: Preston Thompson
// Date: April 7, 2017

module dac_tb;
```

```
5
6       logic reset;
7       logic clk;
8
9       // HALF SCALE SINE
10      logic [15:0] lsamples [16] = '{
11          16'd0, 16'd6122, 16'd11313, 16'd14782,
12          16'd16000, 16'd14782, 16'd11313, 16'd6122,
13          16'd0, -16'd6122, -16'd11313, -16'd14782,
14          -16'd16000, -16'd14782, -16'd11313, -16'd6122
15      };
16      logic [15:0] rsamples [16] = '{
17          16'd0, 16'd6122, 16'd11313, 16'd14782,
18          16'd16000, 16'd14782, 16'd11313, 16'd6122,
19          16'd0, -16'd6122, -16'd11313, -16'd14782,
20          -16'd16000, -16'd14782, -16'd11313, -16'd6122
21      };
22
23      logic avs_write;
24      logic avs_read;
25      logic [31:0] avs_writedata;
26      logic [3:0] avs_address;
27
28      wire dout;
29      wire lrcout;
30      wire sckout;
31      wire bckout;
32      wire [31:0] avs_readdata;
33
34      dac dac_0(.*);
35
36      initial begin
37
38          // initial state
39          reset = 1;
40          clk = 0;
41          avs_write = 0;
42          avs_read = 0;
43
44          repeat (2) begin
45              #10ns; clk = ~clk;
46          end
47
48          reset = 0;
49
50          repeat (10) begin
51              #10ns; clk = ~clk;
52          end
53
54          avs_address = 4'h8;
55          avs_write = 1;
56          for (int i = 0; i < 16; i++) begin
57              avs_writedata = {lsamples[i], rsamples[i]};
58              repeat (2) begin
```

```
59                  #10ns; clk = ~clk;
60              end
61          end
62
63          avs_write = 0;
64
65          repeat (64000) begin
66              #10ns; clk = ~clk;
67          end
68
69          $stop;
70
71      end
72
73  endmodule
```

Listing 5: spi.sv

```
1   //Author: Nathaniel Rohrick
2   //Date: April 7, 2017
3
4   module spi (
5     input logic clk,        //master clk
6     input logic reset,      //master reset
7     input logic miso,       //rx data from sd card
8
9     input logic avs_write,        //write flag from cpu
10    input logic avs_read,         //read flag from cpu
11    input logic [2:0] avs_address,     //cpu address bus
12    input logic [31:0] avs_writedata,  //cpu data output bus
13
14    output logic [31:0] avs_readdata,  //cpu data input bus
15    output logic sck,                  //sd card clock
16    output logic mosi,                 //tx data to sd card
17    output logic n_ss                  //chip select
18  );
19    //Below shows the memory allocation of the registers needed by spi.sv
20    /*
21    ADDRESS BEGIN | ADDRESS END | REG NAME  | DESCRIPTION
22    ------------------------------------------------------------------------
23    0x0000      |  0x0000   | data            | MOSI data buffer to be send out
24    0x0001      |  0x0001   | rcv_buf         | MISO buffer to rec ieve data
25    0x0002      |  0x0002   | div             | sck frequency divider register
26    0x0003      |  0x0003   | busy            | Tells spi caller if peripheral is busy
27    0x0004      |  0x0004   | n_ss            | peripheral enable
28    ------------------------------------------------------------------------
29    */
30    localparam DATA_START = 8'h0;
31    localparam RCV_START = 8'h1;
32    localparam DIV_START =  8'h2;
33    localparam BUSY_START = 8'h3;
34    localparam SS_START = 8'h4;
35
36    //CPU visible registers
```

```
37  logic [7:0] data;       //data to send to slave
38  logic [7:0] rcv_buf;    //rcv buffer that holds data from send
39  logic [7:0] div;        //frequency divider register
40  logic busy;             //busy flag (1 when busy, 0 otherwise)
41  //End memory register description
42
43  localparam BYTE = 16'd8;      //size of a byte
44
45  enum {init, idle, snd_rcv} state, state_next;
46
47  logic [15:0] sck_freq_cnt;  //loaded with number of clk cycles
48  logic [15:0] sck_cnt;       //loaded with number of sck
49  logic [7:0] mosi_ptr;       //pointer to select cmd_buf bit
50  logic [7:0] miso_ptr;       //pointer to select cmd_buf byte
51  logic [7:0] divider;        //variable to make div the actual clock divider rate
52
53  assign divider = div >> 1'd1; //divide by 2 for usability on the firmware side
54
55  //state exit conditions
56  //allow data to latch on falling edge
57  assign SND_RCV_EXIT = ((sck_cnt == 16'd0)&&(sck == 0));
58  assign IDLE_EXIT = (avs_write && (avs_address == DATA_START));
59
60  always_ff @(posedge clk) begin
61    if (avs_read && (avs_address == BUSY_START)) begin
62      avs_readdata[0] <= busy;
63      avs_readdata[31:1] <= '0;
64    end
65
66    state <= state_next;
67
68        if (state_next == init)
69            busy <= 1;
70        else if (state_next == idle) begin
71            busy <= 0;
72            sck_cnt <= BYTE;    //load with number of sck cycles
73            sck_freq_cnt <= divider;//load with number of clk cycles
74            mosi_ptr <= 8'h7;     //get mosi_ptr looking at MSb of data
75            miso_ptr <= 8'h7;     //get miso_ptr looking at MSb of rcv_buf
76            mosi <= 1'd1;
77            sck <= 1'd0;
78        end
79        else if (state_next == snd_rcv)
80            busy <= 1;
81
82    unique case (state)
83      init: begin
84        div <= 8'd1;        //default clk divide is (2/2)*2 = 2
85        sck_cnt <= 16'd8;   //load with number of sck cycles
86        sck_freq_cnt <= divider;//load with number of clk cycles
87        mosi_ptr <= 8'h7;     //get mosi_ptr looking at MSb of data
88        miso_ptr <= 8'h7;     //get miso_ptr looking at MSb of rcv_buf
89        rcv_buf <= 8'hFF;     //default MISO state is high
90        data <= 8'hFF;        //default data  state is 0xFF
```

```
91          mosi <= 1'd1;          //default MOSI state is high
92          sck <= 1'd0;           //default sck state is high
93          n_ss <= 1'd1;          //de-select chip
94        end
95
96        idle: begin
97          //if CPU requests to read/write, allow to do so in idle
98          if (avs_write && (avs_address == DATA_START))
99            data <= avs_writedata[7:0];
100
101         if (avs_read && (avs_address == RCV_START)) begin
102           avs_readdata[31:8] <= 24'd0;
103           avs_readdata[7:0] <= rcv_buf;
104         end else if (avs_read && (avs_address == DIV_START)) begin
105           avs_readdata[31:8] <= 24'd0;
106           avs_readdata[7:0] <= div;
107         end
108
109         if (avs_write && (avs_address == DIV_START))
110           div <= avs_writedata[7:0];
111
112         if (avs_write && (avs_address == SS_START))
113           n_ss <= avs_writedata[0];
114
115       end
116
117       snd_rcv: begin
118
119         if (sck_freq_cnt == 16'd0) begin
120           sck_freq_cnt <= divider;
121           sck <= ~sck;
122         end else begin
123           sck_freq_cnt <= sck_freq_cnt - 16'd1;
124         end
125
126         if ((sck == 1'd0) && (sck_freq_cnt == 16'd0)) begin//latch on rising edge
127           sck_cnt <= sck_cnt - 16'd1;
128           rcv_buf[miso_ptr] <= miso;
129           miso_ptr <= miso_ptr - 8'd1;
130           mosi_ptr <= mosi_ptr - 8'd1;
131         end
132
133         if (sck && (sck_freq_cnt == 16'd0)) begin//shift on falling egde
134           mosi <= data[mosi_ptr];
135         //put valid data on mosi (spi mode 0)
136         end else if ((sck_cnt == 16'd8)&&(sck_freq_cnt == divider)) begin
137           mosi <= data[mosi_ptr];
138         end
139       end
140
141     endcase
142   end
143
144   //bring system to next state
```

```
145   always_comb begin
146     case (state)
147       init: begin
148         if (reset)
149           state_next <= init;
150         else
151           state_next <= idle;
152       end
153
154       idle: begin
155         if (reset)
156           state_next <= init;
157         else if (IDLE_EXIT)
158           state_next <= snd_rcv;
159         else
160           state_next <= idle;
161       end
162
163       snd_rcv: begin
164         if (reset)
165           state_next <= init;
166         else if (SND_RCV_EXIT)
167           state_next = idle;
168         else
169           state_next = snd_rcv;
170
171       end
172     endcase
173   end
174 endmodule
```

Listing 6: spi_tb.sv

```
1  // Author: Preston Thompson
2  // Date: April 7, 2017
3
4  module spi_tb();
5
6      logic clk;
7      logic reset;
8
9      logic avs_write;
10     logic avs_read;
11     logic [2:0] avs_address;
12     logic [31:0] avs_writedata;
13
14     wire [31:0] avs_readdata;
15     wire sck;
16     wire mosi;
17     wire n_ss;
18
19     spi DUT(.*);
20
21     assign miso = mosi;
```

```
22
23      initial begin
24
25          // initial state
26          reset = 1;
27          clk = 0;
28          avs_write = 0;
29          avs_read = 0;
30
31          // perform a reset
32          repeat(2) begin
33              #10ns; clk = ~clk;
34          end
35
36          reset = 0;
37
38          // idle for a bit
39          repeat (10) begin
40              #10ns; clk = ~clk;
41          end
42
43          // write to baud div register
44          avs_address = 3'h2;
45          avs_write = 1;
46          avs_writedata = 32'd4;
47          repeat (2) begin
48              #10ns; clk = ~clk;
49          end
50          avs_write = 0;
51
52          // idle for a bit
53          repeat (10) begin
54              #10ns; clk = ~clk;
55          end
56
57          // deassert ss
58          avs_address = 3'h4;
59          avs_write = 1;
60          avs_writedata = 32'd1;
61          repeat (2) begin
62              #10ns; clk = ~clk;
63          end
64          avs_write = 0;
65
66          // idle for a bit
67          repeat (10) begin
68              #10ns; clk = ~clk;
69          end
70
71          // assert ss
72          avs_address = 3'h4;
73          avs_write = 1;
74          avs_writedata = 32'd0;
75          repeat (2) begin
```

```
            #10ns; clk = ~clk;
        end
        avs_write = 0;

        // idle for a bit
        repeat (10) begin
            #10ns; clk = ~clk;
        end

        // write a byte
        avs_address = 3'h0;
        avs_write = 1;
        avs_writedata = 32'ha5;
        repeat (2) begin
            #10ns; clk = ~clk;
        end
        avs_write = 0;

        // many cycles
        repeat (100) begin
            #10ns; clk = ~clk;
        end

        // read the busy bit
        avs_address = 3'h3;
        avs_read = 1;
        repeat (2) begin
            #10ns; clk = ~clk;
        end
        avs_read = 0;

        // idle for a bit
        repeat (10) begin
            #10ns; clk = ~clk;
        end

    end

endmodule
```

Listing 7: sdc.c

```
//Author: Nathaniel Rohrick
//Date: April 7, 2017

#include "../musicplayer_bsp/system.h"
#include <stdbool.h>
#include <stdint.h>
#include "sdc_cmd.h"
#include "sdc.h"


static volatile const uint8_t* const spi_sdc =
  (volatile uint8_t*)SDC_SPI_0_BASE;
```

```
13 static volatile uint8_t* const spi_data =
14   (volatile uint8_t*)(SDC_SPI_0_BASE + 0x00);
15 static volatile uint8_t* const spi_rcvBuf =
16   (volatile uint8_t*)(SDC_SPI_0_BASE + 0x04);
17 static volatile uint8_t* const spi_div =
18   (volatile uint8_t*)(SDC_SPI_0_BASE + 0x08);
19 static volatile uint8_t* const spi_n_ready =
20   (volatile uint8_t*)(SDC_SPI_0_BASE + 0x0C);
21 static volatile uint8_t* const spi_n_ss =
22   (volatile uint8_t*)(SDC_SPI_0_BASE + 0x10);
23 static volatile uint8_t * const ledc =
24   (volatile uint8_t *)LEDC_0_BASE;
25
26 static const uint32_t WORD_LEN = 32;
27 static const uint8_t STARTb = 0x40;
28
29 static const uint8_t dByte = 0xff;    //dummy byte, all 1's
30 static const uint8_t idleb = 0x01;    //idle bit
31 static const uint8_t respb = 0xFE;  //response good byte
32 static const uint32_t CLK = 50000000;//clk rate of fpga
33
34 //Function: sdInit
35 //Arguments: None
36 //Return: true when error
37 //Description: Initializes an spi communications interface with an external
38 //sd card. Puts sd card into spi mode and sets clock divider to default 2.
39 bool sdc_init ()
40 {
41   uint8_t resp_buf[5];
42
43   for (int i = 0; i < 50000; i++);//delay 1ms
44
45     if (baud_set(126))//set sck to 50M/126~=400kHz
46         return true;
47
48   while (*spi_n_ready);
49   *spi_n_ss = 1;//deassert SS
50   for (int i = 0; i < 50000; i++);//delay 1ms
51   for (int i = 0; i < 200; i++)
52   {
53     while (*spi_n_ready);
54     *spi_data = dByte;
55   }
56
57   for (int i = 0; i < 50000; i++);//delay 1ms
58
59   if (sdc_send_cmd(CMD0, 0x0, R1, resp_buf))//send CMD0
60     return true;
61
62   if (sdc_send_cmd(CMD8, 0x1AA, R7, resp_buf))
63         return true;
64
65   while (*spi_n_ready);
66
```

```
67    if (*resp_buf > 1)
68      return true;
69
70    for (int i = 0; i <= TIMEOUT; i++)
71    {
72      if (sdc_send_cmd(CMD55, 0x0, R1, resp_buf))
73              return true;
74
75      while (*spi_n_ready);
76
77      if (*resp_buf > 1)
78        continue;
79
80      if (sdc_send_cmd(CMD41, 0x40000000, R1, resp_buf))
81              return true;
82
83      if (*resp_buf > 0)
84        continue;
85      else
86        break;
87    }
88    sdc_send_cmd(CMD16, BLOCK_SIZE, R1, resp_buf);
89    while (*spi_n_ready);
90    if (*resp_buf > 0)
91      return true;
92    return(baud_set(2));
93  }
94
95  //Function: baud_set
96  //Arguments: uint8 div divides clk by a value between 2 and 255
97  //Return: True wiith error
98  //Description: Sets clk divider to generate sck, the spi clock
99  bool baud_set (uint8_t div)
100 {
101   while (*spi_n_ready);
102   *spi_div = div;
103   return false;
104 }
105 //Function: sdc_send_cmd
106 //Arguments: takes a command number and a corresponding payload
107 // and an expected response type in resp. It loads the response
108 // into resp_buf.
109 //Return: True when error
110 //Description: Sends a command to SD card and loads a buffer with
111 // the response.
112 bool sdc_send_cmd(
113   const uint8_t cmd,
114   const uint32_t payload,
115   const uint8_t resp,
116   uint8_t* resp_buf
117 )
118 {
119   uint16_t spi_timeout;
120   while (*spi_n_ready);
```

```
121   *spi_n_ss = 0; // assert SS
122   while (*spi_n_ready);
123   *spi_data = dByte;
124   //send cmd index
125   while (*spi_n_ready);
126   *spi_data = cmd | STARTb;
127
128   //send payload  MSB first
129   for (int i = 3; i >= 0; i--)
130   {
131     while (*spi_n_ready);
132     *spi_data = (payload >> (i*8));
133   }
134
135   //send CRC
136   while (*spi_n_ready);
137   if (resp == R1 || resp == R1b)
138     *spi_data = DEF_CRC;
139   else if (resp == R7)
140     *spi_data = CMD8_CRC;
141
142   while (*spi_n_ready);
143   *spi_data = dByte;
144
145   for (spi_timeout = 0; spi_timeout < TIMEOUT; spi_timeout++)
146   {
147     while (*spi_n_ready);
148     if (*spi_rcvBuf & (1 << 7))
149       *spi_data = dByte;//send another byte
150     else if (resp == R1)
151     {
152       *resp_buf = *spi_rcvBuf;
153       break;
154     }
155     else if (resp == R7)
156     {
157       while (*spi_n_ready);
158       *resp_buf = *spi_rcvBuf;
159       for (int i = 1; i < 5; i++)
160       {
161         while (*spi_n_ready);
162         *spi_data = dByte;
163         *(resp_buf + i) = *spi_rcvBuf;
164       }
165       break;
166     }
167     else if (resp == R1b)
168     {
169       do
170       {
171         while (*spi_n_ready);
172         *spi_data = dByte;
173         while (*spi_n_ready);
174       }
```

```
175         while(*spi_rcvBuf != 0xFF);
176         break;
177       }
178     }
179
180     if (((cmd == CMD17 )||(cmd == CMD18))&&(spi_timeout != TIMEOUT))
181       return false;
182
183     while (*spi_n_ready);
184     *spi_n_ss = 1;//deassert SS
185
186
187     //if (spi_timeout == TIMEOUT)
188       //return true;
189
190       while (*spi_n_ready);
191       *spi_data = dByte;
192       while (*spi_n_ready);
193       *spi_data = dByte;
194
195     return false;
196 }
197
198 bool sdc_read(
199     uint32_t addr,
200     uint8_t len,
201     uint8_t* buf
202 )
203 {
204     uint8_t resp_buf[8];
205     if (sdc_send_cmd(CMD18, addr, R1, resp_buf))
206       return true;
207
208       //if (resp_buf[0] != 0)
209           //return true;
210
211     while (*spi_n_ready);
212     *spi_data = dByte;
213
214     for (int iblock = 0; iblock < len; iblock++)
215     {
216       for (int i = 0; i <= TIMEOUT; i++)
217       {
218         while (*spi_n_ready);
219         if (*spi_rcvBuf == 0xFE)
220           break;
221         else
222         {
223           while (*spi_n_ready);
224           *spi_data = dByte;
225         }
226         //if (i == TIMEOUT)
227           //return true;
228       }
```

```
229
230
231     //unsigned block_offs = iblock * BLOCK_SIZE;
232     uint8_t *end = buf + BLOCK_SIZE;
233     for (; buf < end; buf++)//work starts here
234     {
235       while (*spi_n_ready);
236       *spi_data = dByte;
237       while (*spi_n_ready);
238       *buf = *spi_rcvBuf;
239       //buf[i + block_offs] = *spi_rcvBuf;
240             //*buf++ = *spi_rcvBuf;
241     }
242   }
243   while (*spi_n_ready);
244   *spi_data = dByte;
245   while (*spi_n_ready);
246   *spi_data = dByte;
247
248   if (sdc_send_cmd(CMD12, 0x0, R1b, resp_buf)) //stop transmission
249         return true;
250
251     //if (resp_buf[0] != 0)
252         //return true;
253
254   //while (*spi_n_ready);
255   //*spi_n_ss = 1;//deassert SS
256   return false;
257 }
```

Listing 8: sdc.h

```
1  // Author: Nathaniel Rohrick
2  // Date: April 7, 2017
3
4  #ifndef _SDC_H
5  #define _SDC_H
6
7  #include <stdbool.h>
8  #include "system.h"
9
10 bool baud_set (uint8_t div);
11 bool sdc_init ();
12 bool sdc_read(uint32_t addr, uint8_t len, uint8_t* buf);
13 bool sdc_send_cmd(const uint8_t cmd, const uint32_t payload,
14   const uint8_t resp, uint8_t* const resp_buf);
15 #define BLOCK_SIZE 512
16
17 #endif
```

Listing 9: sdc_cmd.h

```
1  // Author: Nathaniel Rohrick
2  // Date: April 7, 2017
```

```

/*
Command breakdown
---------------------------------
B1     | B1 - B5 | B6
---------------------------------
index | payload | CRC
---------------------------------
*/
//name and payload for commands, use 0x95 for CRC (not used)

#define CMD0  0
#define CMD1  1
#define CMD8  8
#define CMD9  9
#define CMD10 10
#define CMD12 12
#define CMD16 16
#define CMD17 17
#define CMD18 18
#define CMD23 23
#define CMD24 24
#define CMD25 25
#define CMD41 41
#define CMD55 55
#define CMD58 58

#define DEF_CRC 0x95
#define CMD8_CRC 0x87

#define R1 1
#define R7 2
#define R1b 3

#define TIMEOUT 100000
```

Listing 10: keypad.sv

```
//Author Nathaniel Rohrick
//Date: April 7, 2017

module keypad(
  input logic [3:0] kpr,          //row vector on keypad
  input logic reset,              //master reset
  input logic clk,                //master clk

  input logic avs_write,          //write flag from cpu
  input logic avs_read,         //read flag from cpu
  input logic [1:0] avs_address,    //cpu address bus
  input logic [31:0] avs_writedata, //cpu data output bus

  output logic [31:0] avs_readdata, //cpu data input bus
  output logic [3:0] kpc          //column vector on keypad
  );
```

```
17
18  //localparam KPHIT_START = 3'd0;  //start address for keypad hit flag
19  localparam BUTTON_START = 1'd0;   //start address for decoded keypress register
20
21  localparam WAIT = 22'h3FFFFF;//~~100ms delay for debouncing
22    localparam DIV = 16'd62500;     //clk divider for polling CLK/(4*DIV)
23
24  logic kphit;          //high when keypad is pressed
25    logic [3:0] kpr_reg;
26  logic [21:0] dbnc_cnt;  //debounce counter
27  logic [15:0] clk_cnt;    //clk frequency counter
28
29  enum {blank, next, v_up, v_down, mute, prev} button;  //button state
30  enum {idle, active, rst} state, state_next;           //state machine variables
31
32  //exit conditions
33  assign IDLE_EXIT = kphit;
34  assign ACTIVE_EXIT = !kphit && (dbnc_cnt == '0);
35
36  //datapath
37  always_ff@(posedge clk) begin
38    state <= state_next;
39        kpr_reg <= kpr;
40
41        if (clk_cnt == '0)
42            clk_cnt <= DIV;
43        else
44            clk_cnt <= clk_cnt - 'd1;
45
46    //always handle requests for keypress info
47    if (avs_read && (avs_address == BUTTON_START)) begin
48      avs_readdata[31:3] <= '0;
49      avs_readdata[2:0] <= button;
50    end
51
52    case (state)
53      rst: begin //load default states
54        kpc <= 4'b0111;
55        dbnc_cnt <= WAIT;
56      end
57
58      idle: begin
59        dbnc_cnt <= WAIT;//prep debounce counter
60        if (state_next == idle) begin//if key is pressed, only 1 to decode
61                    if ((kpr_reg == 4'b1111)&&(clk_cnt == 0)) begin
62                        unique case (kpc)
63                            4'b0111: kpc <= 4'b1011;
64                            4'b1011: kpc <= 4'b1101;
65                            4'b1101: kpc <= 4'b1110;
66                            4'b1110: kpc <= 4'b0111;
67                        endcase
68                    end
69          if ((kpr_reg == 4'b1110) && (kpc == 4'b1110)) begin
70            button <= blank;
```

```
71              kphit <= '1;
72            end else if ((kpr_reg == 4'b1110) && (kpc == 4'b1101)) begin
73              button <= blank;
74              kphit <= '1;
75            end else if ((kpr_reg == 4'b1110) && (kpc == 4'b1011)) begin
76              button <= blank;
77              kphit <= '1;
78            end else if ((kpr_reg == 4'b1110) && (kpc == 4'b0111)) begin
79              button <= blank;
80              kphit <= '1;
81            end else if ((kpr_reg == 4'b1101) && (kpc == 4'b1110)) begin
82              button <= blank;
83              kphit <= '1;
84            end else if ((kpr_reg == 4'b1101) && (kpc == 4'b1101)) begin
85              button <= blank;
86              kphit <= '1;
87            end else if ((kpr_reg == 4'b1101) && (kpc == 4'b1011)) begin
88              button <= v_down;
89              kphit <= '1;
90            end else if ((kpr_reg == 4'b1101) && (kpc == 4'b0111)) begin
91              button <= blank;
92              kphit <= '1;
93            end else if ((kpr_reg == 4'b1011) && (kpc == 4'b1110)) begin
94              button <= blank;
95              kphit <= '1;
96            end else if ((kpr_reg == 4'b1011) && (kpc == 4'b1101)) begin
97              button <= next;
98              kphit <= '1;
99            end else if ((kpr_reg == 4'b1011) && (kpc == 4'b1011)) begin
100             button <= blank;
101             kphit <= '1;
102           end else if ((kpr_reg == 4'b1011) && (kpc == 4'b0111)) begin
103             button <= prev;
104             kphit <= '1;
105           end else if ((kpr_reg == 4'b0111) && (kpc == 4'b1110)) begin
106             button <= blank;
107             kphit <= '1;
108           end else if ((kpr_reg == 4'b0111) && (kpc == 4'b1101)) begin
109             button <= blank;
110             kphit <= '1;
111           end else if ((kpr_reg == 4'b0111) && (kpc == 4'b1011)) begin
112             button <= v_up;
113             kphit <= '1;
114           end else if ((kpr_reg == 4'b0111) && (kpc == 4'b0111)) begin
115             button <= blank;
116             kphit <= '1;
117           end else begin
118             kphit <= 0;
119             button <= blank;
120           end
121         end
122       end
123
124       active: begin
```

```
125         dbnc_cnt <= dbnc_cnt - 'd1;//counts down
126                 if (kpr_reg == 4'b1111)
127                     kphit <= 0;
128       end
129     endcase
130
131   end
132   //control path
133   always_comb begin
134     case(state)
135       rst: begin
136         if (reset)
137           state_next = rst;
138         else
139           state_next = idle;
140       end
141
142       idle: begin
143         if (reset)
144           state_next = rst;
145         else if (IDLE_EXIT)
146           state_next = active;
147         else
148           state_next = idle;
149       end
150
151       active: begin
152         if (reset)
153           state_next = rst;
154         else if (ACTIVE_EXIT)
155           state_next = idle;
156         else
157           state_next = active;
158       end
159     endcase
160   end
161 endmodule
```

Listing 11: keypad.c

```
1  // Author: Nathaniel Rohrick
2  // Date: April 7, 2017
3
4  #include <stdint.h>
5
6  #include "../musicplayer_bsp/system.h"    //defines mapping memory of project
7  #include "keypad.h"
8
9  static volatile uint32_t * const keypad_button =
10   (volatile uint32_t *)(KEYPAD_0_BASE);
11
12 int keypad_get_button(void)
13 {
14   return *keypad_button;
```

```
}
```

Listing 12: keypad.h

```
// Author: Nathaniel Rohrick
// Date: April 7, 2017

#ifndef _KEYPAD_H
#define _KEYPAD_H

enum Keypad_Buttons {
  key_blank,
  key_next,
  key_v_up,
  key_v_down,
  key_mute,
  key_prev
};

int keypad_get_button(void);

#endif
```

Listing 13: main.c

```
// Author: Preston Thompson
// Date: April 7, 2017

#include <stdint.h>

#include "system.h"
#include "sdc.h"
#include "sdc_cmd.h"
#include "keypad.h"

#define DAC_BUFFER_SIZE 2048

static volatile uint16_t * const ledc =
    (volatile uint16_t *)LEDC_0_BASE;
static volatile uint32_t * const dac_sample_read_ptr =
  (volatile uint32_t *)DAC_0_BASE;
static volatile uint32_t * const dac_sample_write_ptr =
  (volatile uint32_t *)(DAC_0_BASE + 0x4);
static volatile uint32_t * const dac_sample =
  (volatile uint32_t *)(DAC_0_BASE + 0x8);

struct track_info {
    uint32_t start;
    uint32_t size;
};

static struct track_info tracks[64];
static uint8_t buf[512];
static int current_track;
```

```
30 static uint32_t current_sector;
31 static uint32_t last_sector;
32 static int num_tracks;
33
34 int get_num_samples(void) {
35     volatile uint32_t write_ptr = *dac_sample_write_ptr;
36     volatile uint32_t read_ptr = *dac_sample_read_ptr;
37   if (write_ptr >= read_ptr)
38     return write_ptr - read_ptr;
39   else
40     return (DAC_BUFFER_SIZE - read_ptr) + write_ptr;
41 }
42
43 void change_track(void) {
44     *ledc = current_track;
45     current_sector = tracks[current_track].start;
46     if (current_track != num_tracks - 1)
47       last_sector = tracks[current_track+1].start - 1;
48     else
49       last_sector = tracks[current_track].start + tracks[current_track].size/512;
50     sdc_read(current_sector, 1, buf);
51     while (keypad_get_button() != key_blank);
52 }
53
54 int main(void) {
55   int i;
56
57   *ledc = 0xff;
58
59   sdc_init();
60
61   sdc_read(0, 1, (uint8_t *)tracks);
62
63     current_track = 0;
64
65     for (i = 0; i < 64; i++) {
66         if (tracks[i].size > 0)
67             num_tracks++;
68     }
69
70     change_track();
71
72     while (1) {
73       int current_key = keypad_get_button();
74         if (current_key == key_next) {
75             current_track++;
76             if (current_track == num_tracks)
77               current_track = 0;
78             change_track();
79             continue;
80         }
81         else if (current_key == key_prev) {
82             current_track--;
83             if (current_track < 0)
```

```
84                current_track = num_tracks - 1;
85            change_track();
86            continue;
87        }
88        else if (current_sector == last_sector) {
89            int dac_room = DAC_BUFFER_SIZE - get_num_samples();
90            if (dac_room >= (tracks[current_track].size % 512)/4) {
91                for (i = 0; i < tracks[current_track].size % 512; i+= 4)
92                    *dac_sample = *(uint32_t *)(buf + i);
93                if (current_track == num_tracks - 1)
94                    current_track = 0;
95                else
96                    current_track++;
97                change_track();
98                sdc_read(current_sector, 1, buf);
99            }
100       }
101       else {
102           int dac_room = DAC_BUFFER_SIZE - get_num_samples();
103           if (dac_room >= 128) {
104               for (i = 0; i < 512; i += 4)
105                   *dac_sample = *(uint32_t *)(buf + i);
106               current_sector++;
107               sdc_read(current_sector, 1, buf);
108           }
109       }
110   }
111
112   return 0;
113 }
```

Listing 14: `fat.c`

```
1  // Author: Preston Thompson
2  // Date: April 7, 2017
3
4  // This is an incomplete FAT32 driver.
5
6  #include <stddef.h>
7  #include <stdint.h>
8
9  #include "fat.h"
10 #include "sdc.h"
11
12 struct Partition_Desc {
13     uint8_t type_code;
14     uint32_t lba_begin;
15 };
16
17 struct Volume_ID {
18     uint16_t BytsPerSec;
19     uint8_t SecPerClus;
20     uint16_t RsvdSecCnt;
21     uint8_t NumFATs;
```

```
22      uint32_t FATSz32;
23      uint32_t RootClus;
24  };
25
26  static uint8_t sector_buf[512];
27  static struct Partition_Desc pd;
28  static struct Volume_ID vid;
29  static uint32_t cluster_begin_lba;
30  static uint32_t fat_begin_lba;
31
32  static void read_word(const uint8_t *src, void *dst, size_t len) {
33      size_t i;
34      for (i = 0; i < len; i++)
35          *(uint8_t*)(dst++) = *src++;
36  }
37
38  void fat_init(void) {
39      sdc_read(0, 1, sector_buf);
40      read_word(sector_buf + 446 + 0x04, &pd.type_code, 1);
41      read_word(sector_buf + 446 + 0x08, &pd.lba_begin, 4);
42      sdc_read(pd.lba_begin, 1, sector_buf);
43      read_word(sector_buf + 0x0b, &vid.BytsPerSec, 2);
44      read_word(sector_buf + 0x0d, &vid.SecPerClus, 1);
45      read_word(sector_buf + 0x0e, &vid.RsvdSecCnt, 2);
46      read_word(sector_buf + 0x10, &vid.NumFATs, 1);
47      read_word(sector_buf + 0x24, &vid.FATSz32, 4);
48      read_word(sector_buf + 0x2c, &vid.RootClus, 4);
49      fat_begin_lba = pd.lba_begin + vid.RsvdSecCnt;
50      cluster_begin_lba = fat_begin_lba + (vid.NumFATs * vid.FATSz32);
51  }
52
53  int fat_get_file_info(uint32_t index, struct File_Info *fi) {
54      uint8_t record[32];
55      uint16_t first_cluster_hi, first_cluster_lo;
56      uint32_t lba_addr;
57
58      lba_addr = cluster_begin_lba + (index / (vid.BytsPerSec / 32));
59
60      sdc_read(lba_addr, 1, sector_buf);
61
62      read_word(sector_buf + index * 32, record, 32);
63
64      if (record[0] == 0)
65          return FAT_END_OF_FILES;
66
67      if ((record[0x0b] & 0xF) == 0xF)
68          return FAT_LONG_FILENAME;
69
70      read_word(record, &fi->name, 11);
71      fi->name[11] = '\0';
72
73      fi->is_dir = (record[11] & (1 << 4)) ? 1 : 0;
74
75      read_word(record + 0x1a, &first_cluster_lo, 2);
```

```
76      read_word(record + 0x14, &first_cluster_hi, 2);
77      fi->first_cluster = first_cluster_lo | (first_cluster_hi << 16);
78
79      read_word(record + 0x1c, &fi->size, 4);
80
81      return FAT_SUCCESS;
82  }
83
84  static uint32_t lba_addr_from_cluster(uint32_t cluster) {
85      return (cluster - vid.RootClus) * vid.SecPerClus + cluster_begin_lba;
86  }
87
88  size_t fat_read_file(uint32_t index, uint32_t sector, uint8_t *dst) {
89      uint32_t lba_addr, cluster_num, cluster, i;
90      struct File_Info fi;
91
92      if (fat_get_file_info(index, &fi) != FAT_SUCCESS)
93          return 0;
94
95      cluster_num = sector / vid.SecPerClus;
96      cluster = fi.first_cluster;
97
98      for (i = 0; i < cluster_num; i++) {
99          sdc_read(fat_begin_lba + (cluster >> 7), 1, sector_buf);
100         read_word(sector_buf + (cluster & 0x7f) * 4, &cluster, 4);
101         if (cluster >= 0xFFFFFFF)
102             return 0;
103     }
104
105     lba_addr = lba_addr_from_cluster(cluster) + (sector % vid.SecPerClus);
106
107     sdc_read(lba_addr, 1, dst);
108
109     read_word(sector_buf + (cluster & 0x7f) * 4, &cluster, 4);
110
111     if (cluster >= 0xFFFFFFF)
112         return fi.size % BLOCK_SIZE;
113     else
114         return BLOCK_SIZE;
115 }
```

Listing 15: fat.h

```
1   // Author: Preston Thompson
2   // Date: April 7, 2017
3
4   #ifndef _FAT_H
5   #define _FAT_H
6
7   #include <stdint.h>
8   #include <stddef.h>
9
10  #define FAT_SUCCESS 0
11  #define FAT_END_OF_FILES 1
```

```
#define FAT_LONG_FILENAME 2

struct File_Info {
    char name[12];
    uint8_t is_dir;
    uint32_t first_cluster;
    uint32_t size;
};

void fat_init(void);
int fat_get_file_info(uint32_t index, struct File_Info *fi);
size_t fat_read_file(uint32_t index, uint32_t sector, uint8_t *dst);

#endif
```

Listing 16: deadtime.m

```
clc;
clear;
fp = fopen('deadtime.txt', 'w');
s = tf('s');

R_low = 100;    %Minimum resistor value (pot = 0)
R_high = 1000;

C = 100e-12;    %Capacitance value

H = (1/(R_low*C))/(s+(1/(R_low*C)));%Filter  transfer function



%step(H);
S_min = stepinfo(H,'RiseTimeLimits',[0.00,0.52]);
S_max = stepinfo(H,'RiseTimeLimits',[0.00,0.68]);
S_typ = stepinfo(H,'RiseTimeLimits',[0.00,0.60]);

fprintf(fp, 'Given:\r\nR_low: %.3f Ohms\r\nR_max = %.3f', R_low, R_high);
fprintf(fp, ' Ohms\r\nC = %.3f pF\r\n \r\n', C*1e12);

fprintf(fp, 'Low end dead time:\r\nTypical:%snS \r\n', S_typ.RiseTime*1e9);
fprintf(fp, 'Min:%snS \r\nMax:%snS', S_min.RiseTime*1e9, S_max.RiseTime*1e9);

H = (1/(R_high*C))/(s+(1/(R_high*C)));%Filter  transfer function

S_min = stepinfo(H,'RiseTimeLimits',[0.00,0.52]);
S_max = stepinfo(H,'RiseTimeLimits',[0.00,0.68]);
S_typ = stepinfo(H,'RiseTimeLimits',[0.00,0.60]);

fprintf(fp, '\r\n \r\nHigh end dead time:\r\nTypical:%snS', S_typ.RiseTime*1e9);
fprintf(fp, ' \r\nMin:%snS \r\nMax:%snS', S_min.RiseTime*1e9, S_max.RiseTime*1e9);
```

Listing 17: V12switcher.m

```
%Author: Nathaniel Rohrick
```

```
%Filename: V12switcher.m
%Date: March 30, 2017
%Description: Design of 12V buck SMPS using TPS54360

clc;clear;
filep = fopen('V12switcher.txt', 'w');

Rdc = 0.0488;         %Inductor resistance, assume 25mOhm
Vout = 12;           %Output voltage
io = 1.5;             %Nominal current value
icl = 3.5;           %Current limit
Vin = 24;            %Input voltage
Vin_max = 48;           %Maximum expected input voltage
Vd = 0.3;            %Forward diode voltage (assumption)
Vf = 0.8;            %Internal forward voltage drop
Cd = 300e-12;        %Diode capacitance estimate
Rds_on = 0.092;       %Maximim internal rdson from datasheet
Ton_min = 135e-9;    %Minimum controllable on time from datasheet
Vref = 0.8;          %internal reference voltage
tr = 4.9e-9;         %estimated rise time of internal gate drive
Qg = 3e-9;           %estimated gate charge of internal fet
Iq = 146e-6;         %queiscient current of chip
f_div = 8;           %Frequency divider of control loop
Vsc = 0.1;           %Short circuit output voltage
Kind = 0.3           %Allowable ripple current as fraction of output current
f_maxskip = (1/Ton_min) * (((io*Rdc)+Vout+Vd)/(Vin_max-(io*Rds_on)+Vd))
f_shift = (f_div/Ton_min) * (((icl*Rdc)+Vsc+Vd)/(Vin_max-(icl*Rds_on)+Vd))
dI = 0.5;            %Allowable change in load current
dV = Vout*0.01;      %Allowable change in load voltage
Ioh = 1.5;             %Output current under heavy load
Iol = 0.9;           %Output current under light load
Ihys = 3.4e-6;       %hysteresis current specified in datasheet
Ienin = 1.2e-6;      %Enable input current
Vstart = 20;         %PSU starts at input voltage of 20V
Vstop = 18;          %PSU stops at input voltage of 16V
Ven = 1.2;           %Enable voltage level without hysteresis
gmps = 12;           %COMP to SW current transconductance
gmea = 250e-6;       %error amplifer transconductance

%Pick 10^log10x below lowest frequency found above.
if (f_maxskip > f_shift)
    fsw = roundn(f_shift - 10^floor(log10(f_shift)), floor(log10(f_shift)));
else
    fsw = roundn(f_maxskip - 10^floor(log10(f_maxskip)), floor(log10(f_maxskip)));
end

RT = rpv((92417 / ((fsw/1000)^0.991))*1000);
RT = RT(1,1);     %R3

%Inductor value calculations
Lo_min = ((Vin-Vout)/(io*Kind))*((Vout)/(Vin*fsw));%L1
Lo_min = 33e-6; %chose 513-1351-1-ND
Irip = ((Vout*(Vin - Vout))/(Vin*Lo_min*fsw));
Il_rms = sqrt((io^2)+((1/12)*(((Vout*(Vin-Vout))/(Vin*Lo_min*fsw))^2)));
```

```
Il_pk = io + (Irip/2);

%Capacitor value calculations
Co1 = ((2*dI)/(fsw*dV));
Co2 = Lo_min*(((Ioh^2)-(Iol^2))/(((Vout+dV)^2)-(Vout^2)));
Co3 = (1/(8*fsw))*(1/(dV/Irip));
Co = 8*max([Co1 Co2 Co3]);%C6/C7
Co = 44e-6;%chose 2 22u in parallel HHXA500ARA220MF61G
Resr = (dV/Irip);%C6/C7 ESR
Resr = 40e-3;%chosen capacitor HHXA500ARA220MF61G in parallel
Icout_rms = (Vout/sqrt(12))*((Vin-Vout)/(Vin*Lo_min*fsw));%C6/C7 irip rms

%Diode value calculations
Pd = (((Vin-Vout)*io*Vd)/Vin)+((Cd*fsw*((Vin+Vd)^2))/2);

%Input capacitor value
Cin = 4.7e-6;%C1/C2
%input rms ripple current %C1/C2 rip rms
I_ic = io*sqrt((Vout/Vin)*((Vin-Vout)/Vin));
dVin = (io*0.25)/(Cin*fsw);%input ripple voltage

%Bootstrap capacitor value
Cboot = 0.1e-6;%specified in datasheet C4

%Under and overvoltage lockout resistor selection
Ruvlo1 = rpv((Vstart - Vstop) / Ihys);
Ruvlo1 = Ruvlo1(1,1);%R1
Ruvlo2 = rpv(Ven / (((Vstart - Ven)/Ruvlo1)+Ienin));
Ruvlo2 = Ruvlo2(1,1);%R2

%Output feedback resistor selector
Rls = 10e3;     %pick 10k for low side resistor R6
Rhs = rpv(Rls * (Vout - Vf) / Vf);
Rhs = Rhs(1,1);%R5

%Compensation values for voltage control loop
fp = io / (2*pi*Vout*Co);
fz = 1 / (2*pi*Resr*Co);
Fco1 = sqrt(fp*fz);
Fco2 = sqrt(fp*(fsw/2));
fco = min(Fco1,Fco1);
R4 = rpv(((2*pi*fco*Co)/gmps)*(Vout/(Vref*gmea)));
R4 = R4(1,1);
C5 = 1/(2*pi*R4*fp);
C8_1 = (Co*Resr) / R4;
C8_2 = 1/(R4*fsw*pi);
C8 = max(C8_2, C8_1);

%Power dissipation estimates for RPS54360
Pcond = (io^2)*Rds_on*(Vout/Vin);
Psw = Vin*fsw*io*tr;
Pgd = Vin*Qg*fsw;
Pq = Vin*Iq;
Pt = Pcond+Pq+Pgd+Psw;
```

```
110
111 fprintf(filep, 'This file computes component values for TPS54360DDA');
112 fprintf(filep, ', refer to\ndatasheet for more information\r\n');
113 fprintf(filep, '\n\nR1 = %.2f\r\nR2 = %.2f\r\n', Ruvlo1, Ruvlo2);
114 fprintf(filep, 'R3 = %.2f\r\nR4 = %.2f\r\n',  RT, R4);
115 fprintf(filep, 'R5 = %.2f\r\nR6 = %.2f\r\n', Rhs, Rls);
116 fprintf(filep, 'C1/C2 = %.2fuF\r\nC4 = %.2fuF\r\n', Cin*1e6, Cboot*1e6);
117 fprintf(filep, 'C5 = %.2fnF\r\n' C5*1e9);
118 fprintf(filep, 'C6/C7 = %.2fuF\r\nC8 = %.2fnF\r\n', Co*1e6, C8*1e9);
119 fprintf(filep, 'L = %.2fuH\r\nP Total = %.2f', Lo_min*1e6, Pt);
```

Listing 18: GATE_DRV_DESIGN.m

```
1  %Author: Nathaniel Rohrick
2  %Date: February 28, 2017
3  %Description:   Creates a list of component values needed for TI's UCC27714
4  %               gate driver IC
5
6  fp = fopen('gate_drive.txt', 'w');
7
8  Qg = 35e-9;              %11nC gate charge AUIRF7640S2CT-ND from digikey
9  Rg = 3.5;                %Intrinsic gate resistance is 3.5Ohm from ds
10
11 fsw = 500e3;             %500 kHz switching frequency
12 Vhv = 24;                %High voltage supply
13 Iqdd = 1050e-6;          %Quiescent current of gate driver Vdd to Vss
14 Iqbs = 300e-6;           %Quiescent current of HB-HS supplies of gate driver
15 Ibl = 20e-6;             %Bootstrap leakage current
16 Vf = 0.3;                %Boot diode Vf
17 Vdd = 12;                %Gate drive voltage supply
18 Cg = Qg / (Vdd - Vf);    %Equivalent gate capacitance
19 Cboot = 10 * Cg;         %Bootstrap cap cant drop 10% in voltage, spec
20                          %cap to be 10x gate capacitance
21
22 Cvdd = 10 * Cboot;       %Vdd capacitor recommended by TI to be 10x Cboot
23
24 Rbias = 5;               %To allow Vdd pins voltage to ramp longer than 50uS
25                          %This is to prevent error logic spikes on O/P
26 Rboot = 2.2;             %Recommended value between 2.2-10, pick minimum
27 Idbootpk = (Vdd - Vf) / Rboot;%Max current boot diode sees on startup
28
29 Pqc = Vdd * (Iqdd + Iqbs);     %Quiescent current power dissipation
30 Pibl = ((Vdd+Vhv) - Vf) * Ibl * 0.5; %Static losses due to HB leakage
31                                      %current at 50% duty cycle PWM
32 Pq1q2 = 2 * Vdd * Qg * fsw;   %Primary power dissipation element turning on
33                               %and off fets.
34 Plvlshft = (Vdd+Vhv) * (0.5e-9) * fsw;%Level shift power dissipation
35                                       %assuming 0.5nC of parasitic charge
36 P_INT = Plvlshft + Pq1q2 + Pibl + Pqc;%Total internal power dissipation
37
38 %Find ripple current of gate driver
39 %Assume maximum allowable current
40 T = 1/fsw;          %Period of switching waveform
41 I = (Vdd - Vf)/Rg;  %Max current from gate driver
```

```
Tc = Qg / I;          %Charge up time

t = 0:(T/10e3):T;
Iwv1 = I*heaviside(t - (T/2)) - I*heaviside(t - (T/2 + Tc));
Iwv2 = -I*heaviside(t - (T-Tc)) + I*heaviside(t - T);
Iwv = Iwv1 + Iwv2;
RMS_Iwv = sqrt((mean(Iwv.^2)));

RMS_TOTAL = 2 * RMS_Iwv;          %2 drivers per chip



fprintf(fp, 'Cboot is: %.4fnF\r\nRboot is: %.4fOhms\r\n', (Cboot*1e9), Rboot);
fprintf(fp, '\r\nInternal Power Dissipation Tabulated Below\r\n');
fprintf(fp, 'Total internal Power dissipation: %.5fW\r\n', P_INT);
fprintf(fp, 'Switching losses: %.5fW\r\n', Pq1q2);
fprintf(fp, 'Level shifting losses: %.5fW\r\n', Plvlshft);
fprintf(fp, 'Quiescient current losses %.5fW\r\n', (Pqc + Pibl));
fprintf(fp, '\r\nRipple Current Rating for bulk decoupler:%.5fA', RMS_TOTAL);

fclose(fp);
```

6.4 Bill of Materials

6.4.1 BOM billed to BCIT

Sheet1

Group No.	Supplier	Part URL	Description	Quantity	Unit Cost
16	Adafruit	https://www.a	MicroSD card breakout board	1	9.95
16	Adafruit	https://www.a	MicroSD card breakout board	1	7.5
16	Digikey	http://www.dig	Stereo audio DAC	1	8.54
16	Adafruit	https://www.a	SMT breakout board for DAC	1	4.95
16	Sparkfun	https://www.sp	Rotary encoder	1	2.95
16	Digikey	http://www.dig	Ceramic bypass cap (0.1uF)	1	0.28
16	Digikey	http://www.dig	Tantalum bypass cap (10uF)	1	0.88
16	Digikey	http://www.dig	Audio op-amp (OPA2134PA)	1	6.16
16	Digikey	http://www.dig	1% 10k resistor	6	0.14
16	Digikey	http://www.dig	Ceramic cap (100pF)	2	0.3
16	Digikey	http://www.dig	Ceramic cap (680pF)	2	0.43
16	Digikey	http://www.dig	Ceramic cap (1500pF)	2	0.51
16	Digikey	http://www.dig	DC blocking cap (1000uF)	2	0.51
16	Sparkfun	https://www.sp	TRRS 3.5mm breakout	1	3.95
			total		

Page 1

Figure 18: Components used in final design

6.4.2 Class-D Amplifier BOM

Name	Manufacturer	Description	Digikey P/N	Type	Value	Package	Unit Cost	Design Quantity	Purchase Quantity	Total Cost	
TL750M12CKVURG3	TI	12V Regulator	296-21601-1-ND	IC		DPAK	1.42	1		$ 1.42	C
UCC27714D	TI	Gate Driver +/- 4A	296-42629-5-ND	IC		14SOIC	4.54	4		$ 18.16	C
LM2937	TI	5V Regulator	LM2937ESX-5.0/NOPBCT-ND	IC		DPAK	1.87	1		$ 1.87	C
TLV3501, TLV3502	TI	Comparator	296-43612-1-ND	IC		8-SOIC	5.57	2		$ 11.14	C
OPA365, 2365	TI	Op-Amp	296-31704-1-ND	IC		8-SOIC	3.92	2		$ 7.84	C
PTD90	Bourns Inc.	2 Ganged Log Pot	PTD902-2015K-A503-ND	R	50k	Through-Hole	2.85	2		$ 5.70	C
Shunt Jumper	3M	Female Jumper	3M9580-ND	JP		Socket	0.14	6		$ 0.84	C
Pin Header	3M	Male 2-Pin Header	3M9447-ND	JP		Through-Hole	0.16	6		$ 0.96	C
10k Ohm Resistor	Yageo	0.1% 10k Resistor	YAG2329CT-ND	R	10k	1206	0.51	5		$ 2.55	C
FDY302NZ	Fairchild	Logic NMOS	FDY302NZCT-ND	NMOS	20V	SC89	0.47	8		$ 3.76	C
CD74AC14	TI	6x Inverting Schmitt Trigger	296-1981-1-ND	IC		SOIC 14	0.89	2		$ 1.78	C
200k Ohm Resistor	Yageo	1% 200k Resistor	311-200KFRCT-ND	R	200k	1206	0.1	1		$ 0.10	C
10k Ohm Resistor	Yageo	1% 10k Resistor	311-10.0KFRCT-ND	R	10k	1206	0.1	21		$ 2.10	C
5.9k Ohm Resistor	Yageo	1% 5.9k Resistor	311-5.90KFRCT-ND	R	5.9k	1206	0.1	1		$ 0.10	C
8.45k Ohm Resistor	Yageo	1% 8.45k Resistor	311-8.45KFRCT-ND	R	8.45k	1206	0.1	1		$ 0.10	C
1uF Capacitor	KEMET	10% 1uF MLCC Capacitor	399-1254-1-ND	C	1uF 15V	1206	0.17	8		$ 1.36	C
0.1uF Capacitor	KEMET	10% 0.1uF MLCC Capacitor	399-9305-1-ND	C	0.1uF 15V	1206	0.11	9		$ 0.99	C
4.7uF Capacitor	KEMET	10% 4.7uF MLCC Capacitor	399-3524-1-ND	C	4.7uF 15V	1206	0.31	5		$ 1.55	C
22uF Capacitor	United Chemi-Con	20% 22uF Electrolytic Capacitor	HHXA500ARA220MF61G	C	22uF 50V	F61 SMT	1.41	4		$ 5.64	C
N-CH MOSFET	Infineon	100V 36nC NMOS	IRF6644TRPBFCT-ND	MOSFET	36nC 100V	DirectFET	2.95	8		$ 23.60	C
1uF Capacitor	KEMET	10% 1uF MLCC Capacitor	399-8147-1-ND	C	1uF 50V	1206	0.23	11		$ 2.53	C
0.1uF Capacitor	KEMET	10% 0.1uF MLCC Capacitor	399-1249-1-ND	C	0.1uF 50V	1206	0.1	8		$ 0.80	C
MBRS1100T3G Diode	ON Semiconductor	1A Shottkey Diode	MBRS1100T3GOSCT-ND	D	1A 100V	SMB 403A-03	0.4	12		$ 4.80	C
MBR0530T1G Diode	ON Semiconductor	0.5A Shottkey Diode	MBR0530T1GOSCT-ND	D	0.5A 30V	SOD-123	0.36	8		$ 2.88	C
F6505CT-ND	Littlefuse	200W TVS	SPHV12-01ETG-C	TVS	200W 25V	SOD-882	0.4	8		$ 3.20	C
SMCJ60CA	Littlefuse	1.5kW TVS	SMCJ60CALFCT-ND	TVS	1.5kW 100V	DO-214AB	0.62	8		$ 4.96	C
EEV-FK1J471M	Panasonic	470uF Electrolytic Capacitor	PCE3485CT-ND	C	470u 63V	SMT	1.88	6		$ 11.28	C
UVR1J472MRD6	Nichicon	4.7mF Electrolytic Capacitor	493-1134-ND	BFC	4.7mF 63V	Through-Hole	4.61	2		$ 9.22	C
16SEV470M10X10.5	Rubycon	470uF Electrolytic Capacitor	1189-2403-1-ND	C	470uF 16V	SMD	0.8	0		$ -	C
25TKV470M10X10.5	Rubycon	470uF Electrolytic Capacitor	1189-2072-1-ND	C	470uF 25V	SMD	1.14	2		$ 2.28	C
OSTTC040162	On Shore Technology	4 Port Screw Terminal	ED2602-ND	CONN		Through Hole	0.79	1		$ 0.79	C
OSTTC022162	On Shore Technology	2 Port Screw Terminal	ED2609-ND	CONN		Through Hole	0.42	2		$ 0.84	C
SJ-2524-SMT-TR	CUI Inc.	4 Terminal Stereo Jack	CP-2524SJCT-ND	CONN		SMD	0.98	1		$ 0.98	C
QBLP615-IB	QT Brightek (QTB)	Blue LED	1516-1072-1-ND	D	Blue D 3.1V	1206	0.5	1		$ 0.50	C
QBLP615-IG	QT Brightek (QTB)	Green LED	1516-1073-1-ND	D	Green D 3.1V	1206	0.5	1		$ 0.50	C
QBLP615-R	QT Brightek (QTB)	Red LED	1516-1076-1-ND	D	Red 2V	1206	0.41	1		$ 0.41	C
ST141D00	E-Switch	SPST Toggle Switch	EG4810-ND	SW	20A 125V	Panel (Through Hole)	4.28	1		$ 4.28	
100 Ohm Resistor	Yageo	1% 100 Resistor	YAG3804CT-ND	R	100	1206	0.1	6		$ 0.60	C
154 Ohm Resistor	Yageo	1% 154 Resistor	YAG3814CT-ND	R	154	1206	0.1	1		$ 0.10	C
100pF Capacitor	Wurth	5% 100pF MLCC	732-7868-1-ND	C	100pF	1206	0.35	5		$ 1.75	C
1k Pot	Copal Electronics Inc.	1k Potentiometer	ST4ETA102CT-ND	R POT	1k	SMD	1.32	4		$ 5.28	
1uF Film Cap	KEMET	5% 1uF Film Capacitor	R82EC4100DQ70J	C	1uF 63V	Through Hole	1.06	4		$ 4.24	
33uH	Eaton	15% 33uH Inductor	513-1351-1-ND	L	33uH 4.42A	SMD	3.36	4		$ 13.44	
2R2	Vishay	5% 2.2Ohm Film Resistor	541-2.2RACT-ND	R	2.2Ohm	2512	0.59	4		$ 2.36	C
5R5	Vishay	5% 5Ohm Film Resistor	541-5.1RBCT-ND	R	5Ohm	2512	0.75	4		$ 3.00	C
						TOTAL		202		$ 172.58	

Figure 19: Components needed for class-D amplifer