Digital Music Synthesizer
E155 Final Project Report
December 8, 2017
Aaron Lutzker and Gabe Rubin

Abstract:
The goal of this project is to create a digital music synthesizer using the following system components: the Raspberry Pi, MuddPi Mark IV, DAC, speaker with built-in amplifier and a MIDI keyboard. The synthesizer is capable of playing a sine, triangle, sawtooth, or square wave at a frequency between 20Hz and 1KHz. Each key on the MIDI keyboard differs in frequency by a factor of approximately $2^{\frac{1}{12}}$. The note duration is specified by the duration of the user keypress on the USB MIDI keyboard, which is connected to the Raspberry Pi. Waveform selection is performed via a repurposed switch on the USB MIDI keyboard.

Introduction:

Musical synthesizers are often expensive and clunky equipment, which rely on fine tuned analog components, making synthesizers less available to general consumers and harder to produce for manufacturers. Digital synthesizers can solve both of these problems. Digital waveform synthesis is cheaper than ever with the low cost and accessibility of digital circuits. Furthermore, digital synthesizers will not decay over time as much as their more expensive analog counterparts. This project's goal was to produce a basic digital synthesizer with different waveform capabilities and the capacity to easily add more synthesizer features beyond the scope of this project in the future.

Breakdown:

The project can be broken into two main categories: hardware and software. The software side will handle user inputs and interface with the hardware, while the hardware side will do the waveform generation and output the waveform through a speaker. Below is an overall block diagram of the system.

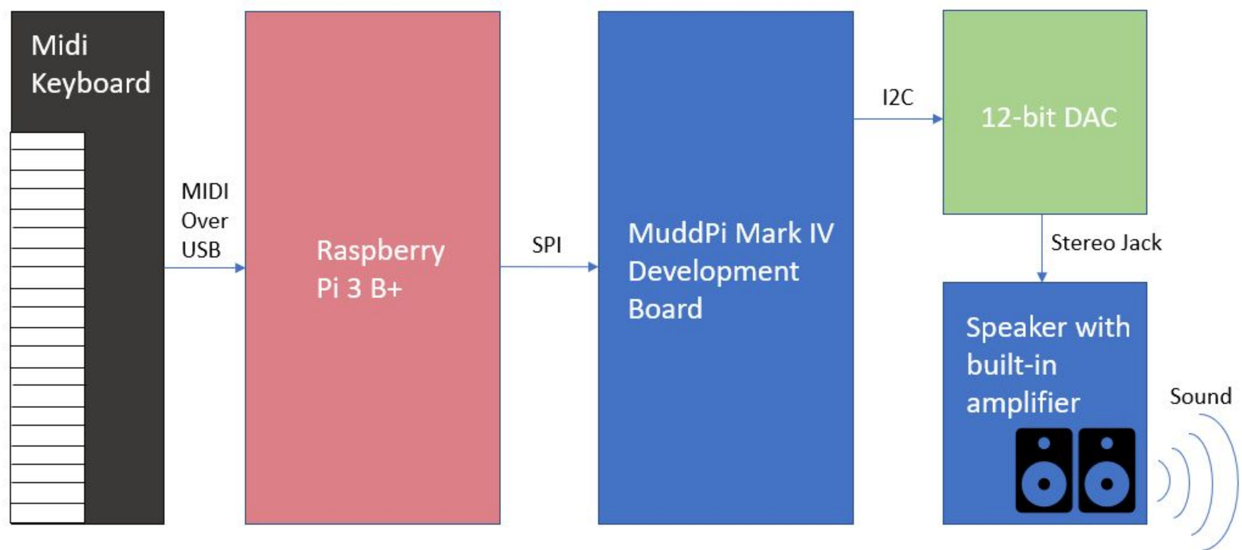

Figure 1

New Hardware:

The new hardware was the MCP4725, 12-bit digital to analog converter, which will henceforth be referred to as the DAC, on a breakout board sold by Adafruit. The DAC is a MCP4725 DAC surface mounted on a PCB with pull up resistors for an I2C bus and decoupling capacitors sold pre-assembled by Adafruit. The DAC uses an I2C bus for communication, which normally works at up to 400 KHz. The MCP4725 DAC has high speed capabilities allowing it to be updated at a bitrate of 3.4Mbps. In order to run the DAC in "high-speed mode," a special message must be sent on startup to tell the DAC to run in high speed mode. This is done on reset and startup of the FPGA. The DAC also has a pin which serves the function of changing the I2C address of the DAC from 0x62 to 0x63 when the pin is driven high. This pin was grounded in order to ensure the address stays at 0x62.

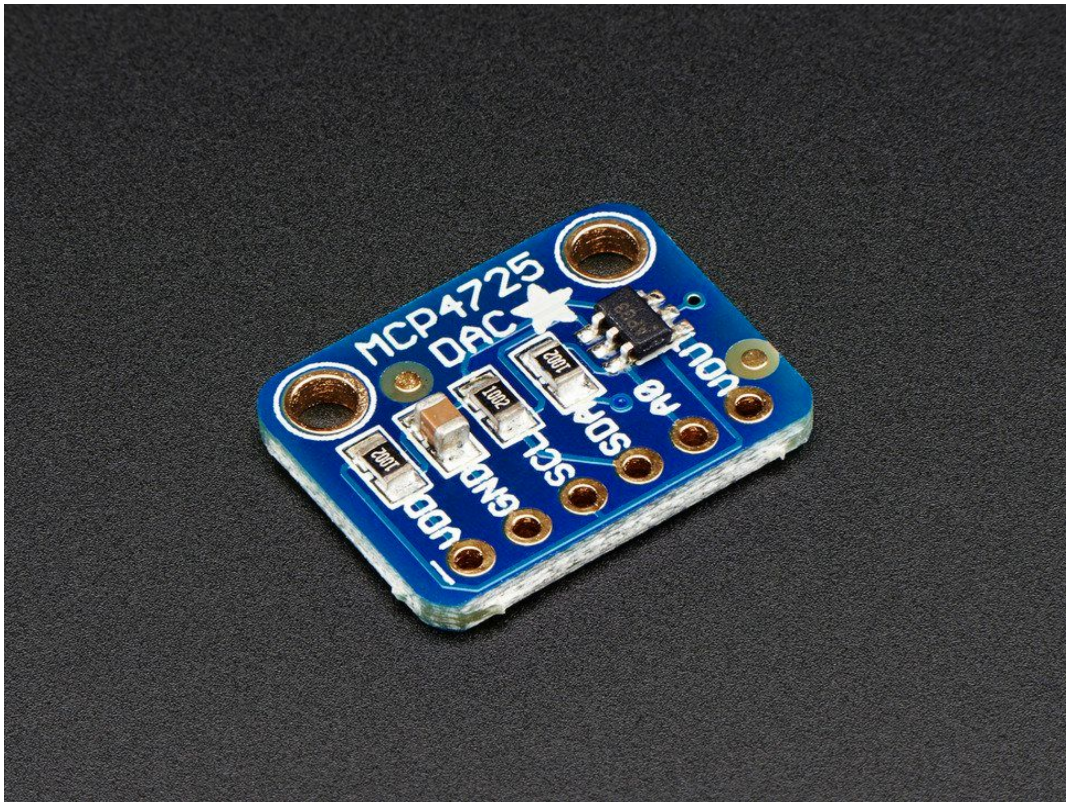

Figure 2

Schematic:

Below is the schematic of all of the circuitry, which was implemented on a standard breadboard.

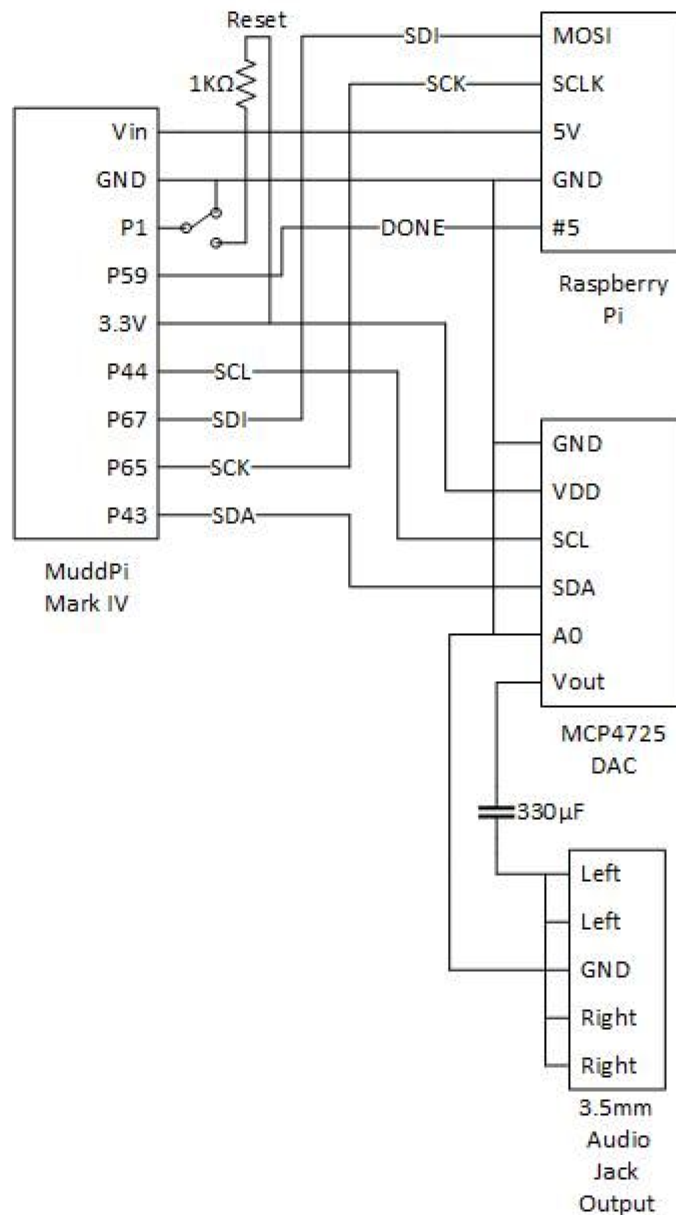

Figure 3

FPGA Design:

Below is a diagram depicting the modules used in the FPGA design, which was done using SystemVerilog. The naming convention in the diagram matches the naming convention used in the SystemVerilog which can be found in the appendix under the “code” section.

Figure 4

The hardware design is broken into two major parts: the FPGA and the DAC. The FPGA does all of the signal processing and creates a waveform, which is then sent to the DAC over an I2C bus. The FPGA logic is broken into four main modules: SPI slave, Waveform Generation, Wave Adder, and I2C master, as can be seen in Figures 5, 6, 7 and 13. The SPI slave is a slightly modified SPI protocol that receives data from the Raspberry Pi, which will also be referred to as the Pi, parses the data and then sends it to the Waveform Generation module. The modifications to the SPI protocol removes unnecessary features for this design and adds a convenient done signal. The done signal is pulled high when data is being transmitted then pulled low when the entire message has been sent. The modified SPI protocol can be seen in Figure 10, in the appendix. These modifications to SPI reduces the amount of logic needed on the FPGA for the SPI module, opening up room for more logic. The Raspberry Pi, and FPGA SPI module use a predetermined message format that is documented in Figure 11, in the appendix. The Pi sends the channel number, FPGA parameters for the sampling clock divider and a waveform select code, leaving extra padding in the message for any future features.

The Waveform Generation module uses a clock divider based on parameters passed from the Pi, to sample the wavetable at the desired sampling frequency, generating the proper note. The module also selects which pre-saved wavetable to sample from in order to output the user selected waveform as dictated by a portion of the SPI data message. The wavetable contains a period of each waveform where each point is 12-bits wide in order to be compatible with the 12-bit DAC. The Waveform Generation module is synthesized 16 times on the FPGA which allows for the 16 most recently pressed notes to be played concurrently. These are referred to as

the 16 waveform channels.

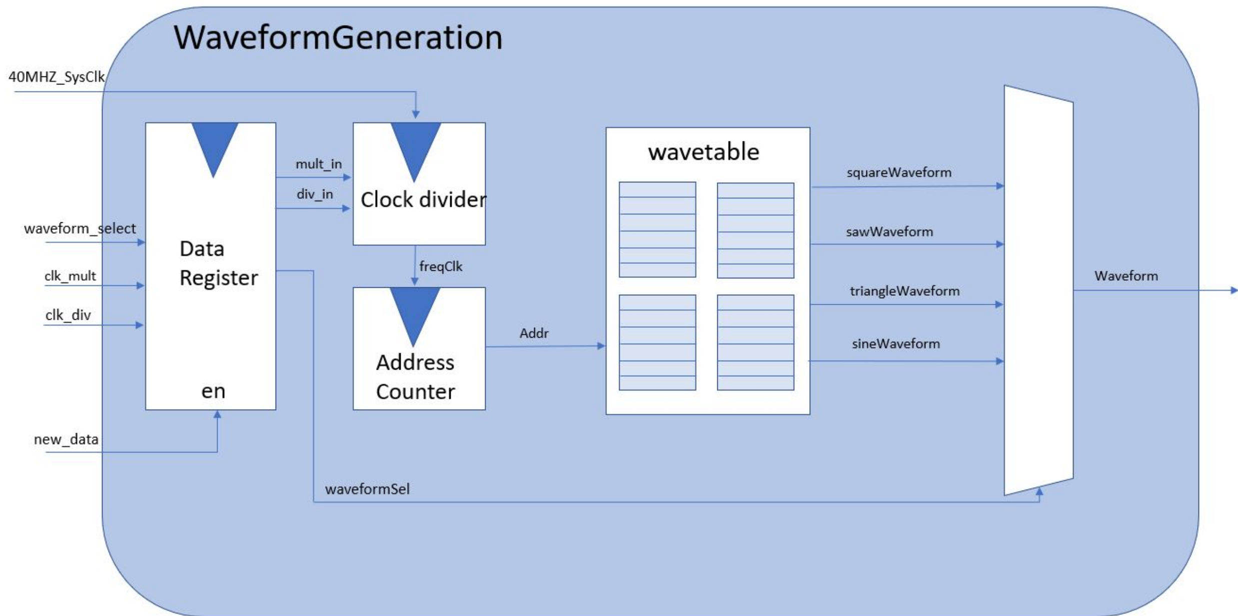

Figure 5

The Wave Adder module on the FPGA adds the output of the 16 waveform channels and then normalizes the amplitude based on how many channels are currently active. The wave adder then samples the output at 10KHz and sends the data over to the I2C module to the DAC.

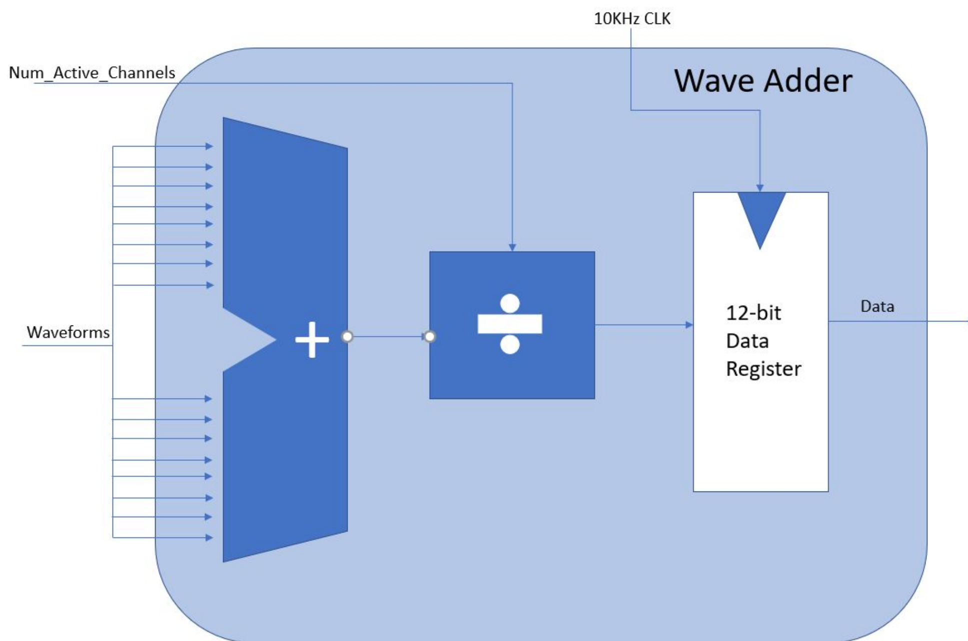

Figure 6

The I2C Master module on the FPGA is a I2C master protocol, with limited functionality to reduce logic needed on the FPGA by eliminating unneeded features. The I2C module has a transmission protocol documented in Figures 9 and 12, in the appendix. The I2C module lacks a receive data function and has a fixed address because it was unnecessary to read data from the DAC and the DAC has a static I2C address. The I2C master is also built “acknowledgement blind,” which means that it waits for but does not check if the slave device responds to the message transfer after each byte as specified by the I2C protocol. Block diagrams for all three FPGA modules are included in Figures 5, 6, 7, and 13.

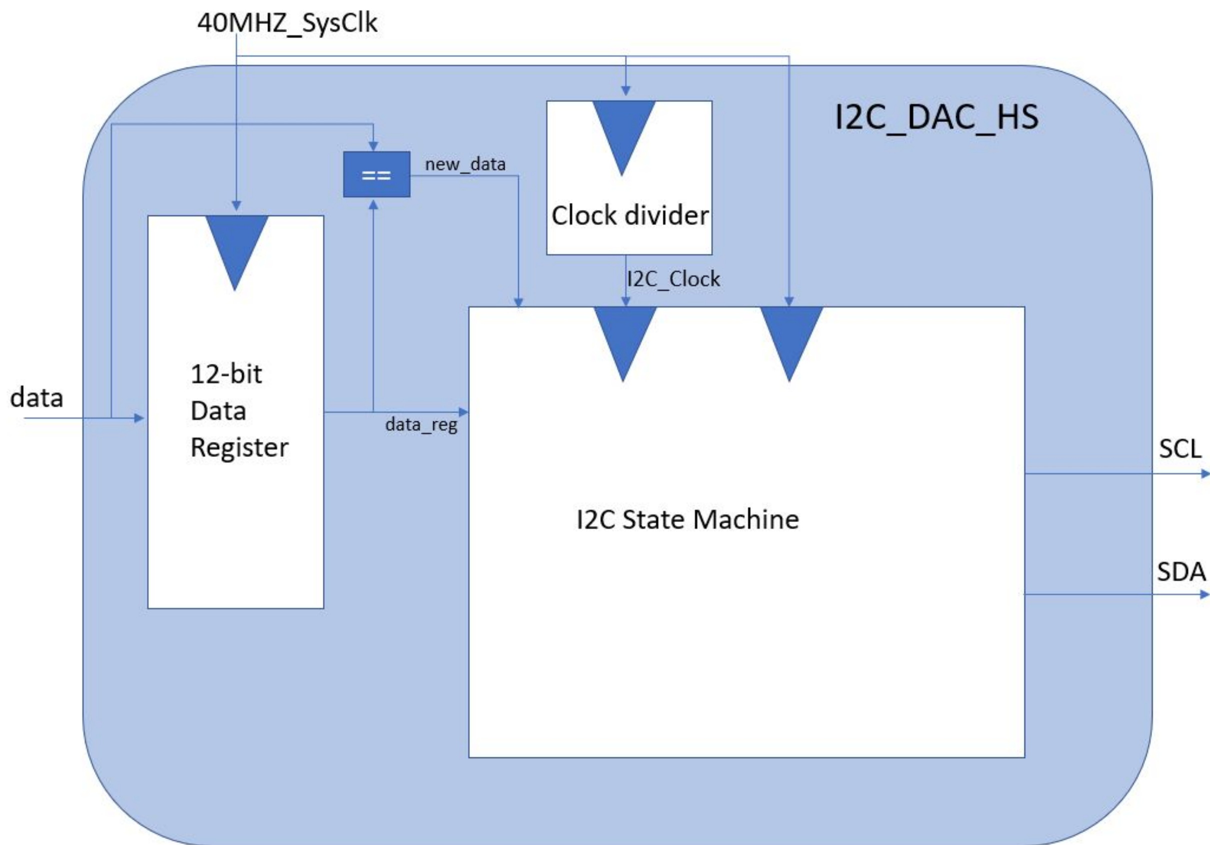

Figure 7

<u>Software Design:</u>

The software side of this project, used the Raspberry Pi 3 B+ running Raspbian. On top of the operating system, libraries and programs written in C were used, which can be found in the references and the appendix. The software running on the Raspberry Pi utilizes a native library on the Pi, aMIDI, to parse MIDI commands as well as EASYPIO, which was provided on the E155 class page, for peripheral use.

The aMIDI library gives the system support for MIDI commands passed through the USB port on the device. The source library was found online which was written in C. The library was able to be downloaded, built and adapted to fit the needs of the project. The native version of aMIDI on the Pi is system exclusive, meaning it could not be modified or used effectively inside a C program. In the downloaded aMIDI library, the program could be modified such that instead of just printing out the MIDI messages sent to the USB port, the program could additionally processes the messages with respect to the waveform generation, formatting the parameters to fit into a SPI message, which then are sent over SPI to the FPGA.

The standard MIDI command is 3 bytes long, with the first byte representing the command type as well as channel and the other bytes representing parameters of the command. This project only utilized the on and off command, for waveform generation, and a special command corresponding to a switch on the keyboard, which dictated waveform selection. If the command is a note on command, the MIDI key number is extracted and converted to the clock division parameters for the clock divider in the Waveform Generation unit on the FPGA. The program then calls a function to pack the word into a data message and send it over SPI to the FPGA.

To send data to the FPGA, the spiSendReceive function in EASYPIO was utilized eight times in a row to transmit the 64 bit data word. A block diagram of the software routines is pictured below in Figure 8.

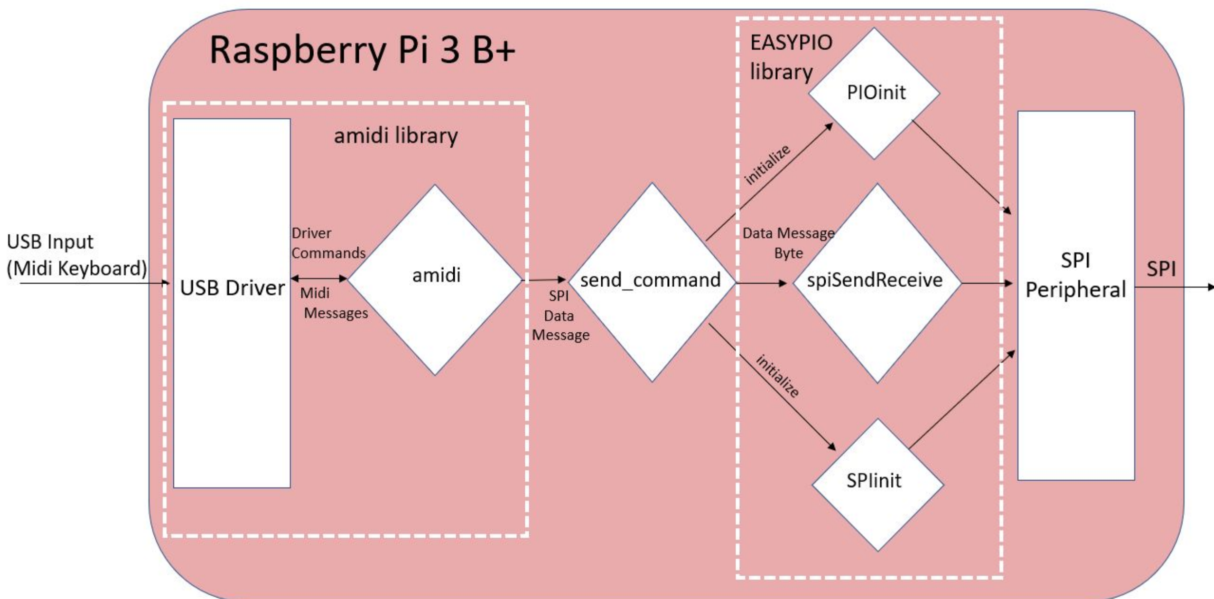

Figure 8

Results:

The digital synthesizer accomplished all set objectives as well as some stretch goals, such as the ability to play multiple notes simultaneously. The arbitrary playback of different frequencies and wave types was successfully accomplished. Having met the set forth objectives, polyphonic capabilities were added so the user could play up to 16 notes at the same time. The only prevailing issue, which was a problem with either the inexpensive MIDI keyboard or the aMIDI library, was that if many keys were pressed or released almost simultaneously, the on or off message would sometimes not be logged. This resulted in either a note not being played or the note not turning off. This was patched by added a 5 second override timer to the wave synthesis modules, such that a wave that was playing longer than 5 seconds would be overridden and turned off. This worked quite well and as in music, the longest note, a whole note, is approximately four seconds. Furthermore, during the demo no one attempted to play a note for a time that exceeded the override. Overall, this project was a success, the synthesizer was able to play some beautiful music as was demonstrated by some musically inclined students of E155.

Appendix:

References:

[1] 12-bit DAC photo: www.adafruit.com/product/935
[2] E155 Class Page: http://pages.hmc.edu/harris/class/e155/
[3] aMIDI Library: https://github.com/bear24rw/alsa-utils

Parts:

Part	Price	Quantity	Supplier
MIDI keyboard	37.93	1	Amazon
12-bit DAC	4.99	2	Adafruit
3.5mm Stereo Jack	0.99	2	Adafruit
Adafruit Shipping	2.99	1	Adafruit
Amazon Shipping	0.00	1	Amazon
Total:	52.88		

Part Links:
MIDI Keyboard:
https://amazon.com/MIDIplus-AKM320-MIDI-Keyboard-Controller/dp/B00VHKMK64/ref=sr_1_20?s=musical-instruments&rps=1&ie=UTF8&qid=1509327114&sr=1-20&refinements=p_85%3A2470955011

12-bit DAC:
https://www.adafruit.com/product/935

3.5 mm Stereo Jack:
https://www.adafruit.com/product/1699

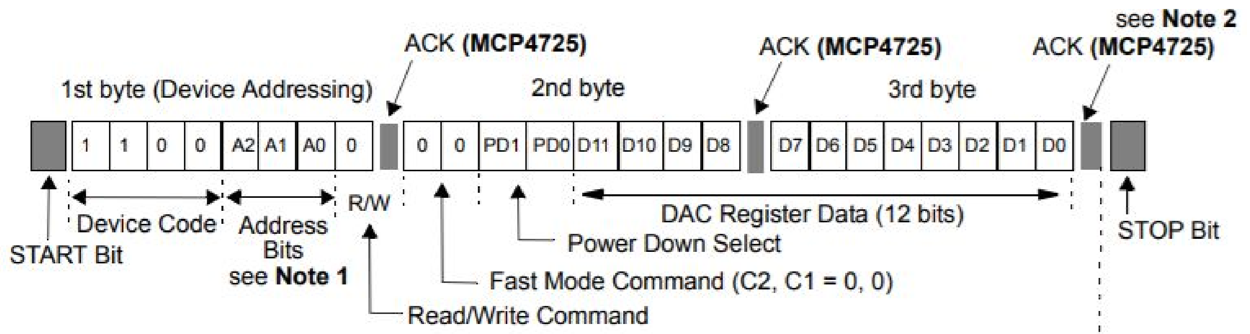

Figure 9 I2C Message for DAC

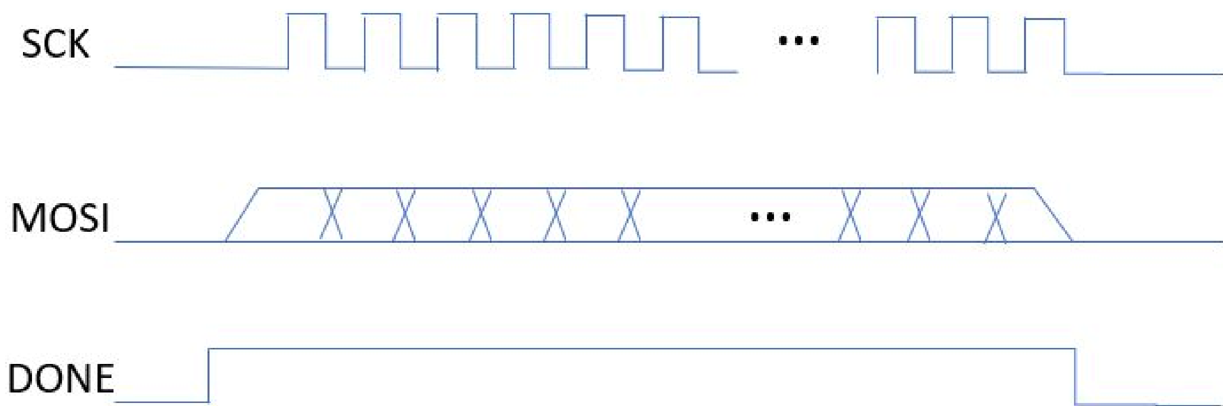

Figure 10

Figure 11

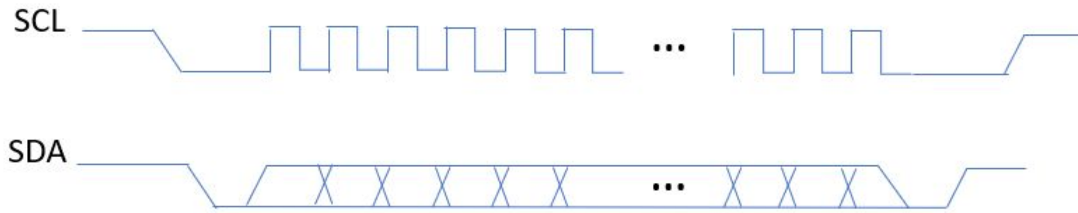

Figure 12

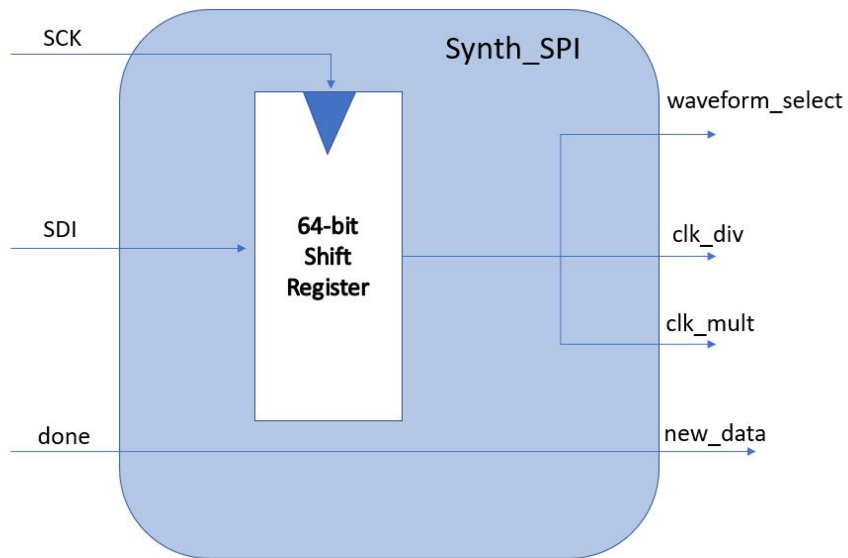

Figure 13

.

Code:

```
//top module for putting everything together
module Multi_Channel_Synthesizer(input logic clk,reset,sck, sdi, CS,
                                         output logic scl, sda);


//signals
logic [5:0] clk_div;
logic [13:0] clk_mult;
logic [15:0] new_data;
logic
new_data_flag,channel_active0,channel_active1,channel_active2,channel_active3,channel_activ
e4,channel_active5,channel_active6,channel_active7,channel_active8,channel_active9,channel_a
ctive10,channel_active11,channel_active12,channel_active13,channel_active14,channel_active1
5;
logic [1:0] waveform_sel;
logic [11:0]
data0,data1,data2,data3,data4,data5,data6,data7,data8,data9,data10,data11,data12,data13,data14,
data15, new_waveform;
logic [3:0] channel;
logic [4:0] channel_active_count;

assign channel_active_count = channel_active0 +  channel_active1 + channel_active2 +
channel_active3 + channel_active4 + channel_active5 + channel_active6 + channel_active7 +
channel_active8 + channel_active9 + channel_active10 + channel_active11 + channel_active12
+ channel_active13 + channel_active14 + channel_active15;

//spi module
mc_synth_spi SPI(clk, sck, sdi, reset, CS, new_data_flag, waveform_sel,clk_div, clk_mult,
channel);

//waveform gen          channels: 16
//all channels get data from message
```

```
//but only specified channel gets new data flag to use the data
waveformGenerationHighRes wgen0(clk, reset, new_data[0],
clk_mult,clk_div,waveform_sel,data0,channel_active0);
waveformGenerationHighRes wgen1(clk, reset, new_data[1],
clk_mult,clk_div,waveform_sel,data1,channel_active1);
waveformGenerationHighRes wgen2(clk, reset, new_data[2],
clk_mult,clk_div,waveform_sel,data2,channel_active2);
waveformGenerationHighRes wgen3(clk, reset, new_data[3],
clk_mult,clk_div,waveform_sel,data3,channel_active3);
waveformGenerationHighRes wgen4(clk, reset, new_data[4],
clk_mult,clk_div,waveform_sel,data4,channel_active4);
waveformGenerationHighRes wgen5(clk, reset, new_data[5],
clk_mult,clk_div,waveform_sel,data5,channel_active5);
waveformGenerationHighRes wgen6(clk, reset, new_data[6],
clk_mult,clk_div,waveform_sel,data6,channel_active6);
waveformGenerationHighRes wgen7(clk, reset, new_data[7],
clk_mult,clk_div,waveform_sel,data7,channel_active7);
waveformGenerationHighRes wgen8(clk, reset, new_data[8],
clk_mult,clk_div,waveform_sel,data8,channel_active8);
waveformGenerationHighRes wgen9(clk, reset, new_data[9],
clk_mult,clk_div,waveform_sel,data9,channel_active9);
waveformGenerationHighRes wgen10(clk, reset, new_data[10],
clk_mult,clk_div,waveform_sel,data10,channel_active10);
waveformGenerationHighRes wgen11(clk, reset, new_data[11],
clk_mult,clk_div,waveform_sel,data11,channel_active11);
waveformGenerationHighRes wgen12(clk, reset, new_data[12],
clk_mult,clk_div,waveform_sel,data12,channel_active12);
waveformGenerationHighRes wgen13(clk, reset, new_data[13],
clk_mult,clk_div,waveform_sel,data13,channel_active13);
waveformGenerationHighRes wgen14(clk, reset, new_data[14],
clk_mult,clk_div,waveform_sel,data14,channel_active14);
waveformGenerationHighRes wgen15(clk, reset, new_data[15],
clk_mult,clk_div,waveform_sel,data15,channel_active15);

//Wave adder 8 channels
wave_adder waveadd(clk, reset, data0, data1, data2, data3, data4, data5, data6, data7, data8,
data9, data10, data11, data12, data13, data14, data15,channel_active_count, new_waveform);

//I2c to send to DAC
```

```
DAC_I2C_HS DAC(clk, reset, new_waveform,scl,sda);



always_comb
	begin
		case(channel) //distribtuing new_data flag for specificed channel in message
			4'd0: new_data = {15'b0, new_data_flag};
			4'd1: new_data = {14'b0, new_data_flag, 1'b0};
			4'd2: new_data = {13'b0, new_data_flag, 2'b0};
			4'd3: new_data = {12'b0, new_data_flag, 3'b0};
			4'd4: new_data = {11'b0, new_data_flag, 4'b0};
			4'd5: new_data = {10'b0, new_data_flag, 5'b0};
			4'd6: new_data = {9'b0, new_data_flag, 6'b0};
			4'd7: new_data = {8'b0, new_data_flag, 7'b0};
			4'd8: new_data = {7'b0, new_data_flag, 8'b0};
			4'd9: new_data = {6'b0, new_data_flag, 9'b0};
			4'd10: new_data = {5'b0, new_data_flag, 10'b0};
			4'd11: new_data = {4'b0, new_data_flag, 11'b0};
			4'd12: new_data = {3'b0, new_data_flag, 12'b0};
			4'd13: new_data = {2'b0, new_data_flag, 13'b0};
			4'd14: new_data = {1'b0, new_data_flag, 14'b0};
			4'd15: new_data = {new_data_flag, 15'b0};
		endcase
	end



endmodule

//test bench
module Multi_Channel_Synthesizer_tb();

	logic clk, reset, sck, sdi, CS, scl, sda;
	logic [63:0] cntrl_message;

	Multi_Channel_Synthesizer dut(clk, reset, sck, sdi, CS, scl, sda);

	//generate clk
```

```
	always
		begin
			clk <= 1'b1; #12.5ns;
			clk <= 1'b0; #12.5ns;
		end

	initial
		begin
			cntrl_message <= {39'b0,23'b00101000000000000110001}; //trying to get
about 400 HZ channel 1
			reset <= 1'b1;
			#25ns;
			reset <= 1'b0;
			#25ns;
			CS <= 1'b1;
			for(int x = 0;  x < 64; x++)
				begin
				sck <= 1'b0;
				sdi <= cntrl_message[63 - x]; //push over message MSB first
				#100ns; //wait
				sck <= 1'b1;
				#100ns;
				end
			CS <= 1'b0;
			#1000ns;
			cntrl_message <= {39'b0,23'b11101000000000000100001}; //trying to get
about 200 HZ channel 7
			CS <= 1'b1;
			for(int x = 0;  x < 64; x++)
				begin
				sck <= 1'b0;
				sdi <= cntrl_message[63 - x]; //push over message MSB first
				#100ns; //wait
				sck <= 1'b1;
				#100ns;
				end
			CS <= 1'b0;
			#1000000ns; //wait indefinetly
```

```
		end
endmodule

//waveform table look up for 50 point high res waveforms
module waveformGenerationHighRes(input logic clk, reset, en,
							input logic [13:0] mult_in,
							input logic [5:0] div_in,
							input logic [1:0] waveformSel,
							output logic [11:0] waveformval,
							output logic channel_active);



	logic freqClk;
	logic [11:0] waveformval_table;
	logic [13:0] mult_out;
	assign counter_reset = en; //new data flag resets note length counter
	createFreqClk freqClkGen(clk, reset, en, mult_in, div_in, mult_out, freqClk); //sampling
clk gen
	note_counter(clk, counter_reset, note_off); //note counter asserts note_off signal once
time up
	assign channel_active = ~(|mult_out); //using convenient fact that if channel is all 0
	//it is no longer active and we can turn it off

	logic [7:0] maxaddr, addr;
	always_comb
		begin
			case(waveformSel)
				2'b00: maxaddr=8'd49;//squarewave
				2'b01: maxaddr=8'd49;//saw
				2'b10: maxaddr=8'd47;//tri
				2'b11: maxaddr=8'd48;//sine
				default: maxaddr=8'd49;//default squarewave
			endcase
			//zeroes output if note not being used or note length flag has been thrown
			if((mult_out == 14'b0) | note_off)
				waveformval = 12'b0;
			else//otherwise just output what the wavetable gives us
				waveformval = waveformval_table;
```

```
		end

	always_ff @(posedge freqClk, posedge reset)
	begin
		if(reset)
			begin
				addr <= 0;
			end
		else
			begin
				if( addr >= maxaddr)
					begin
						addr <= 0;
					end
				else
					begin
						addr <= addr + 1;
					end
			end
	end

	waveformGen waveGen(waveformSel, addr, waveformval_table);

endmodule

//counter that throws flag once note has been on for 5 seconds
module note_counter(input logic clk, reset,
					output logic note_off);
		logic [27:0] count; //need >= 28 bits for 200 Million
		assign note_off = (count >= 28'd200000000); //5 seconds for 40 MHZ clock
			always_ff @(posedge clk)
				begin
					if(reset)
					count <= 28'b0;
					else if(count < 28'd200000000)
					count <= count + 1;
					else
					count <= count;
				end
```

```
endmodule


module waveformGen(input logic [1:0]waveform,
                                                input logic [7:0] addr,//this will come from the
createFreqClk
                                                output logic [11:0] waveformval);

        logic [11:0] sawWaveformVal, triangleWaveformVal, squareWaveformVal,
sineWaveformVal;
        loadSquareWaveform square(addr, squareWaveformVal);
        loadSawtoothWaveform sawtooth(addr, sawWaveformVal);
        loadTriangleWaveform triangle(addr, triangleWaveformVal);
        loadSineWaveform sine(addr, sineWaveformVal);

        always_comb
                begin
                        case(waveform)
                                2'b00: waveformval=squareWaveformVal;
                                2'b01: waveformval=sawWaveformVal;
                                2'b10: waveformval=triangleWaveformVal;
                                2'b11: waveformval=sineWaveformVal;
                                default: waveformval=squareWaveformVal;
                        endcase
                end
endmodule

module createFreqClk(input logic clk, reset, en,
                                                        input logic [13:0] mult_in,
                                                        input logic [5:0] div_in,
                                                        output logic [13:0] mult_out,
                                                        output logic newclk);

        logic [31:0]counter;
        logic [13:0]multiplier;
        logic [5:0]divisor;
```

```
	assign mult_out = multiplier; //used for giving higher level
	//entities access to the current multiplication factor

	assign newclk=counter[divisor];
	always_ff @(posedge clk, posedge reset)
		begin
			if(reset)
				begin
					counter <= 32'b0;
					multiplier <= 14'b0;
					divisor <= 6'b0;
				end
			else if(en)
				begin
					counter <= 0;
					multiplier <= mult_in;
					divisor <= div_in;
				end
			else
				begin
					counter <= counter + multiplier;
					multiplier <= multiplier;
					divisor <= divisor;
				end
		end
endmodule

module loadSawtoothWaveform(input logic [7:0]a,
										output logic [11:0]y);

	logic [11:0] sawtoothWaveformValues[0:49];

	initial $readmemh("sawtoothWaveformValues.txt", sawtoothWaveformValues);
	assign y = sawtoothWaveformValues[a];
endmodule

module loadSineWaveform(input logic [7:0]a,
									output logic [11:0]y);
```

```
	logic [11:0] sineWaveformValues[0:48];

	initial $readmemh("sineWaveformValues.txt", sineWaveformValues);
	assign y = sineWaveformValues[a];
endmodule

module loadSquareWaveform(input logic [7:0]a,
						output logic [11:0]y);

	logic [11:0] squareWaveformValues[0:49];

	initial $readmemh("squareWaveformValues.txt", squareWaveformValues);
	assign y = squareWaveformValues[a];
endmodule

module loadTriangleWaveform(input logic [7:0]a,
						output logic [11:0]y);

	logic [11:0] triangleWaveformValues[0:47];

	initial $readmemh("triangleWaveformValues.txt", triangleWaveformValues);
	assign y = triangleWaveformValues[a];
endmodule



//I2C Master High Speed Version
module DAC_I2C_HS(input logic clk_40MHZ, reset,
			input logic [11:0] data,
			output logic scl,sda);

		logic [6:0] counter, half_period;
		logic [36:0] data_out; //reg holds data being outputted  (37 bits)
		logic [5:0] bit_counter;
		logic [11:0] data_reg;
		logic [7:0] byte_1,byte_2,byte_3,byte_4; //data bytes for sending data
		logic clk_I2C, got_data, new_data, transmit_active, transmit_soon, HS_mode_set;

		assign byte_1 =  8'b11000100; //Addr byte will be cosntant <= got this from
```

tutorial on adafruit

```
		assign byte_2 = 8'b01000000; //settings : write dac reg, reg operating mode
		assign byte_3 = data_reg[11:4]; //upper 8 bits of data reg
		assign byte_4 = {data_reg[3:0],4'b0000}; //lower 4 bits of data reg plus zeroes

		assign scl = transmit_active? clk_I2C: 1'b1; //output I2C clock is transmission is
active
		assign half_period = HS_mode_set? 7'd9: 7'd49;    ///changes depending on if in
HS mode or not


		//40MHZ clk sequential logic
		always_ff @(posedge clk_40MHZ)
			begin
			if(reset)
				begin
				clk_I2C <= 1'b0;
				counter <= 7'd0;
				data_reg <= 11'd0;
				new_data <= 1'b0;
				end
			else if(counter == half_period)
				begin
				counter <= 7'd0;
				clk_I2C <= ~clk_I2C;
				end
			else if(data_reg != data)
				begin
				new_data <= 1'b1; //set new data flag
				counter <= counter + 1; //increment counter
				data_reg <= data; //put data into data_reg
				end
			else if(got_data)
				begin
				new_data <= 1'b0; //clear new data flag
				counter <= counter + 1;
				end
			else
				counter <= counter + 1;
```

```
                    end

            //I2C clk sequential logic (used for sending data)
            //might need to change this to negedge to accomodate for typical i2c protocol
            //new data rate must be kept slightly below 400KHZ/38 =aprox. 10KHZ , thus we
can do max of 10KHZ sampling rate
            always_ff @(negedge clk_I2C)
                    begin
                            if(reset)
                                    begin
                                            bit_counter <= 6'd0;
                                            sda <= 1'b1;
                                            transmit_active <= 1'b0;
                                            data_out <= 37'd0;
                                            got_data = 1'b1;
                                            transmit_soon <= 1'b0;
                                            HS_mode_set <= 1'b0;
                                    end
                            else if(transmit_soon & bit_counter != 6'd37)
                                    begin
                                            transmit_active <= 1'b1;
                                            transmit_soon <= 1'b0;
                                            sda <= data_out[36-bit_counter]; //shifitng out
nescessary bit (skipping first bit cuz will be zero and is already shifted out at start)
                                            bit_counter <= bit_counter + 1; //incrementing bit
counter
                                    end
                            else if(transmit_active & bit_counter != 6'd37)
                                    begin
                                            sda <= data_out[36-bit_counter]; //shifitng out
nescessary bit (skipping first bit cuz will be zero and is already shifted out at start)
                                            bit_counter <= bit_counter + 1; //incrementing bit
counter
                                    end
                            else if(new_data)
                                    begin
                                            got_data <= 1'b1; //reset new data flag
                                            sda <= 1'b0;  //start new data transmission
                                            data_out <=
```

```
{byte_1,1'b1,byte_2,1'b1,byte_3,1'b1,byte_4,2'b10};  //load whole message into register , 1'b1
are to take space of ACK bit
                                                transmit_soon <= 1'b1; //set flag that we are
transmitting data on next cycle
                                        end
                                else if(bit_counter == 6'd37)
                                        begin
                                                got_data = 1'b0; //reset got data flag
                                                bit_counter <= 6'd0;
                                                transmit_active <= 1'b0;
                                                sda <= 1'b1; //asserting first part of stop condition
                                                HS_mode_set <= 1'b1; //if got to here, then HS
mode has had to been set
                                        end
                                else if(!HS_mode_set)
                                        begin
                                                got_data <= 1'b1; //reset new data flag
                                                sda <= 1'b0;  //start new data transmission
                                                data_out <= {8'b00001001,29'b0}; //send HS mode
message
                                                transmit_soon <= 1'b1; //set flag that we are
transmitting data on next cycle
                                        end

                                else
                                        sda <= 1'b1; // do nothing / second part of stop condition
                        end

endmodule


/*
module send_I2C_testbench();


        logic clk_40MHZ,reset,data,scl,sda;
        logic [11:0] data_1;

        just_DAC_I2C dut(clk_40MHZ, reset,data_1,scl,sda);
```

```
	always
		begin
			clk_40MHZ <= 1'b0; #12.5ns;
			clk_40MHZ <= 1'b1; #12.5ns;
		end

	initial
		begin
		reset <= 1'b1;
		data_1 <= 12'd0;
		#50ns;
		reset <= 1'b0;
		#1000ns; //no data comes in for 1000 ns
		data_1 <= 12'hFFF; //full scale voltage data comes in
		#10000000ns; //wait indefinetly
		end


endmodule
*/




//64 bit spi message
//many bits not used (leaving for future expansion of features)

module mc_synth_spi(input  logic sys_clk_40MHZ, sck, sdi, reset, CS,
	     output  logic new_data,
	     output logic [1:0] waveform_sel,
			output logic [5:0] clk_div,
			output logic [13:0] clk_mult,
			output logic [3:0] channel);

	logic [63:0] data_message;
	logic transfer_active;

	//breaking up message into components
```

```
	assign channel = data_message[25:22];
	assign clk_div = data_message[19:14];
	assign clk_mult = data_message[13:0];
	assign waveform_sel = data_message[21:20];


	//System clock logic
	always_ff @(posedge sys_clk_40MHZ)
		begin
			if(reset)
				begin
					new_data <= 1'b0;
					transfer_active <= 1'b0;
				end
			else if(CS)
				begin
					transfer_active <= 1'b1;
				end
			else if(transfer_active) //when CS pulled low, transfer has ended so we
need to throw new_data flag
				begin
					transfer_active <= 1'b0; //transfer is no longer active
					new_data <= 1'b1;
				end
			else
				new_data <= 1'b0; //no new data
		end


	//Spi clk logic
  always_ff @(posedge sck, posedge reset) //if sck is clocked or reset pulled high
		begin
			if(reset)
				data_message <= 64'b0;
			else if(CS)
				data_message = {data_message[62:0], sdi};
		end
endmodule
```

```
//Support for multiple channel audio
//samples addition and mormalization of waveforms at max sampling frequency
module wave_adder(input logic clk_40MHZ, reset,
                                        input logic [11:0] data0, data1, data2, data3, data4, data5, data6,
data7, data8, data9, data10, data11, data12, data13, data14, data15,
                                                  input logic [4:0] channel_active_count,
                                                  output logic [11:0] new_waveform);

                                        logic clk_sampling;
                                        logic [15:0] counter, half_period;
                                        logic [15:0] waveadded; //additional bits for growth of number
                                        logic [11:0] normalized_wave; //added and divided
                                        logic [4:0] channels_on; //divide by factor

                                        assign half_period = 16'd2000;
                                        assign waveadded = data1 + data2 + data3 + data4 + data5 + data6
+ data7 +data8 + data9 + data10 + data11 + data12 + data13 + data14 + data15 + data0; //max 15
bits wide
                                        assign normalized_wave = waveadded/channel_active_count;

                              //Clock dividor for 10 KHZ sampling of waveform added (10 x
fundamental of highest freq waveform)
                    always_ff @(posedge clk_40MHZ)
                              begin
                              if(reset)
                                        begin
                                        clk_sampling <= 1'b0;
                                        counter <= 16'd0;
                                        end
                              else if(counter == half_period)
                                        begin
                                        counter <= 16'd0;
```

```
                clk_sampling <= ~clk_sampling;
                end
            else
                counter <= counter + 1;
            end

        //samples generated to output
        always_ff @(posedge clk_sampling)
            begin
                new_waveform <= normalized_wave;
            end

endmodule
```

Pin Assignments:

```
# -------------------------------------------------------------------------- #
#
# Copyright (C) 1991-2013 Altera Corporation
# Your use of Altera Corporation's design tools, logic functions
# and other software and tools, and its AMPP partner logic
# functions, and any output files from any of the foregoing
# (including device programming or simulation files), and any
# associated documentation or information are expressly subject
# to the terms and conditions of the Altera Program License
# Subscription Agreement, Altera MegaCore Function License
# Agreement, or other applicable license agreement, including,
# without limitation, that your use is for the sole purpose of
# programming logic devices manufactured by Altera and sold by
# Altera or its authorized distributors.  Please refer to the
# applicable agreement for further details.
#
# -------------------------------------------------------------------------- #
#
# Quartus II 64-Bit
# Version 13.0.0 Build 156 04/24/2013 SJ Web Edition
# Date created = 13:58:09  November 24, 2017
#
# -------------------------------------------------------------------------- #
```

```
#
# Notes:
#
# 1) The default values for assignments are stored in the file:
#		Multi_Channel_Synthesizer_assignment_defaults.qdf
#    If this file doesn't exist, see file:
#		assignment_defaults.qdf
#
# 2) Altera recommends that you do not modify this file. This
#    file is updated automatically by the Quartus II software
#    and any changes you make may be lost or overwritten.
#
# -------------------------------------------------------------------------- #


set_global_assignment -name FAMILY "Cyclone IV E"
set_global_assignment -name DEVICE EP4CE6E22C8
set_global_assignment -name TOP_LEVEL_ENTITY Multi_Channel_Synthesizer
set_global_assignment -name ORIGINAL_QUARTUS_VERSION 13.0
set_global_assignment -name PROJECT_CREATION_TIME_DATE "13:58:09  NOVEMBER
24, 2017"
set_global_assignment -name LAST_QUARTUS_VERSION "13.0 SP1"
set_global_assignment -name SYSTEMVERILOG_FILE waveformGenerationHighRes.sv
set_global_assignment -name SYSTEMVERILOG_FILE wave_adder.sv
set_global_assignment -name SYSTEMVERILOG_FILE Multi_Channel_Synthesizer.sv
set_global_assignment -name SYSTEMVERILOG_FILE mc_synth_spi.sv
set_global_assignment -name SYSTEMVERILOG_FILE DAC_I2C_HS.sv
set_global_assignment -name PROJECT_OUTPUT_DIRECTORY output_files
set_global_assignment -name MIN_CORE_JUNCTION_TEMP 0
set_global_assignment -name MAX_CORE_JUNCTION_TEMP 85
set_global_assignment -name ERROR_CHECK_FREQUENCY_DIVISOR 1
set_global_assignment -name NOMINAL_CORE_SUPPLY_VOLTAGE 1.2V
set_global_assignment -name EDA_SIMULATION_TOOL "ModelSim-Altera
(SystemVerilog)"
set_global_assignment -name EDA_OUTPUT_DATA_FORMAT "SYSTEMVERILOG HDL"
-section_id eda_simulation
set_global_assignment -name PARTITION_NETLIST_TYPE SOURCE -section_id Top
set_global_assignment -name PARTITION_FITTER_PRESERVATION_LEVEL
PLACEMENT_AND_ROUTING -section_id Top
```

```
set_global_assignment -name PARTITION_COLOR 16764057 -section_id Top
set_global_assignment -name STRATIX_DEVICE_IO_STANDARD "2.5 V"
set_location_assignment PIN_59 -to CS
set_location_assignment PIN_88 -to clk
set_location_assignment PIN_1 -to reset
set_location_assignment PIN_65 -to sck
set_location_assignment PIN_44 -to scl
set_location_assignment PIN_43 -to sda
set_location_assignment PIN_67 -to sdi
set_instance_assignment -name PARTITION_HIERARCHY root_partition -to | -section_id Top
```

Modified AMIDI Library Code:

```
/*
 *  aMIDI.c - read from/write to RawMIDI ports
 *
 *  Copyright (c) Clemens Ladisch <clemens@ladisch.de>
 *
 *
 *   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., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
```

```
#include <string.h>
#include <ctype.h>
#include <getopt.h>
#include <errno.h>
#include <signal.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 "EASYPIO.h"
#include <math.h>

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 int timeout;
static int stop;
static snd_rawMIDI_t *input, **inputp;
static snd_rawMIDI_t *output, **outputp;


#define LOAD_PIN 5

//////////////////////////
//global variables for channels
/////////////////////////

//data structure for channel mult-factors active
int channels[] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}; //initializing to 0
```

```
int ws = 0; //waveselect has inital value of 0
/////////////////////////////////////
// Functions
/////////////////////////////////////////////

//function retrieves channel that multfactor is in
int get_channel(int multfactor) //array is global
{
	int i = 0;
	while( i < 16)
	{
		if(channels[i] == multfactor)
			return i;
		else
			i++;
	}
	return 0; //if function passed non-existing multfactor, assume channel 0

}
int find_empty_channel(void) //array is global
{
	int i = 0;
	while( i < 16)
	{
		if(channels[i])
			i++;
		else
			return i;
	}
	return 0; //if no open channel, just use channel 0;
}

void send_Command(char onOff, char keynum, char volume) {

  long word =360448;//22, left shifted 14 for div field,  because this is the divde we will always
use
  //use and shifts and ors to send message  as a long
  //for on case
```

```
if( onOff == 0x90){//if note is below 20 Hz or above 1Khz, dont play anything
	if((keynum < 21) | (keynum > 84))
	{
		return;
	}
	double freq = (27.5*pow(2,((((double) (keynum)) - 21)/12)));
	//printf("Freq: %f\n", freq);
	int multfactor = ((int) (2*freq/(0.095))); //calculate mult factor from desired frequency
	word = (word | (multfactor & 0x3FFF));//and with 3FFF to bound multfactor
	int channel_num = find_empty_channel(); //finding an empty channel
	word = (word | (channel_num << 22)); //orring in channel number
	channels[channel_num] = multfactor; //recording multfactor active in channel
	word = (word | (ws << 20));
}
else if(onOff == 0x80){ //if note is below 20 Hz or above 1Khz, dont play anything
	if((keynum < 21) | (keynum > 84))
	{
		return;
	}
	int multfactor = 0;
	word = (word | (multfactor & 0x3FFF));//off case
	double freq = (27.5*pow(2,((((double) (keynum)) - 21)/12)));
	int old_mult = ((int) (2*freq/(0.095)));
	int channel_num = get_channel(old_mult);
	word = (word | (channel_num << 22)); //orring in channel number
	channels[channel_num] = 0; //returning state of channel to open
	//int ws = digitalRead(WAVE_SELECT0);
	//ws = ws | (digitalRead(WAVE_SELECT1) << 1);
	word = (word | (ws << 20));
}
else if(onOff == 0XE0) { //wave select command
	 if((keynum == 0x7f) & (volume == 0x7f)){ //rocker all forward
		 ws = (ws + 1) % 4; //cycles through 4 wave modes
	 }
	 word = (word | (ws << 20));  //put wave select into command
}
```

```
  else{
        return;
  }

  //Sending command over SPI
  int i=0;
  digitalWrite(LOAD_PIN,1);
   while(i < 8) {
    spiSendReceive((char) (word>>((7-i)*8) & 0xff)); //getting correct 8 bits of data
    i = i + 1;
  }
   digitalWrite(LOAD_PIN,0);
}

//AMIDI functions begin:


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   don't ignore active sensing bytes\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 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;
}

int main(int argc, char *argv[])
{
	static const char short_options[] = "hVlLp:s:r:S::dt:a";
	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'},
		{ }
	};
	int c, err, ok = 0;
	int ignore_active_sensing = 1;
	int do_send_hex = 0;
```

```
	//Our code,
	//initiating peripherals single time
pioInit(); //initiate peripherals
spiInit(1000000, 0);  //initiate spi connection to 1 Mbps
pinMode(LOAD_PIN,OUTPUT);
//pinMode(WAVE_SELECT0,INPUT);
//pinMode(WAVE_SELECT1,INPUT);


	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':
			timeout = atoi(optarg);
			break;
		case 'a':
			ignore_active_sensing = 0;
			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 ((err = snd_rawMIDI_write(output, send_data, send_data_length)) < 0) {
			error("cannot send data: %s", snd_strerror(err));
			goto _exit;
		}
	}

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

		timeout *= 1000;
		npfds = snd_rawMIDI_poll_descriptors_count(input);
		pfds = alloca(npfds * sizeof(struct pollfd));
		snd_rawMIDI_poll_descriptors(input, pfds, npfds);
		signal(SIGINT, sig_handler);
		for (;;) {
			unsigned char buf[1024];
			int i, length;
			unsigned short revents;

			err = poll(pfds, npfds, 200);
			if (stop || (err < 0 && errno == EINTR))
				break;
			if (err < 0) {
				error("poll failed: %s", strerror(errno));
				break;
			}
			if (err == 0) {
				time += 200;
				if (timeout && time >= timeout)
					break;
```

```
				continue;
			}
			if ((err = snd_rawMIDI_poll_descriptors_revents(input, pfds, npfds,
&revents)) < 0) {
				error("cannot get poll events: %s", snd_strerror(errno));
				break;
			}
			if (revents & (POLLERR | POLLHUP))
				break;
			if (!(revents & POLLIN))
				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 (!ignore_active_sensing || buf[i] != 0xfe)
					buf[length++] = buf[i];
			if (length == 0)
				continue;
			read += length;
			time = 0;
			if (receive_file != -1)
				write(receive_file, buf, length);
			if (dump) {
				for (i = 0; i < length; ++i)
					print_byte(buf[i]);
				//printf("\n buf[0]=%X\n", buf[0]);
				//printf("buf[1]= %X\n", buf[1]);
				send_Command(buf[0], buf[1], buf[2]);
				fflush(stdout);
			}
		}
		if (isatty(fileno(stdout)))
```

```
				printf("\n%d bytes read\n", read);
	}

	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;
}
```