Final Project: Digital Synthesizer

ECEN 2020 - Embedded Systems

Fall 2016

Lab Group:

Bader Albader, Shane Kirkley, Christopher Mann

Introduction:

For our final project, we created a basic digital synthesizer using the MSP432. A MIDI device, specifically a Novation ReMote 25-key MIDI controller, is used to send messages to the microcontroller, which ideally would use wavetable synthesis to output data over I2C to a digital to analog converter (DAC), which would then output to a speaker. The Direct Memory Access (DMA) module was used to place generated wavetable data into a communication register, allowing for far less overhead due to communications, and giving us more processing power for the actual sound synthesis calculations. Unfortunately our lab group faced difficulties with I2C, so the digital to analog converter was scrapped in favor of simply producing square wave output from digital I/O, controlled by a timer. Our DMA code is still intact and successfully sends data over UART, which could easily be changed to I2C if we could get it working. This process will be discussed in the paper in addition to our "plan B" implementation, which ultimately allowed us to reach our goal, albeit with more limited functionality that initially planned.

Hardware Specifications:

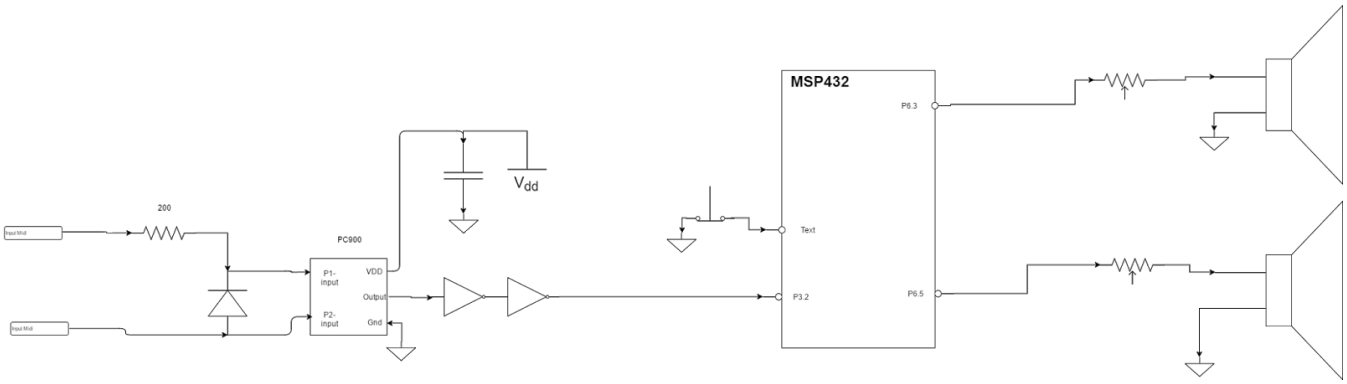


Figure 1 - Digital Synthesizer Circuit Diagram

The hardware starts with an electronic keyboard (midi controller) that sends midi signals to other devices. The raw midi signal is only ~1 V and is not a clean square wave, as shown in Figure 2. When both data lines were connected to an oscilloscope there was no distinct signal only static. This raw midi signal was run from the midi controler into an optocoupler which converted the signal into a semi-square wave. Although this new signal was much better it still was causing some minor miscommunications. To further clean up the signal it was run through two inverters which had a very low turn on threshold making a clean square wave. Ideally we

should have used an op-amp or a buffer for this purpose, but the temporary solution worked and we moved on to other priorities. The newest form of this signal was a nice square wave that transferred 2 to 3 bytes of information to the MSP through a pin on the board. The MSP produced a square wave based on the signal that it received and sent that signal to two pins. The pins were each connected to a potentiometer for volume control and a speaker. Our first configuration had both speakers connected in parallel to each other and received a signal off the same pin. This configuration was much cleaner then the final design but the speakers seemed to ground the signal not producing a sound.

Figure 2 - Raw MIDI signal from controller, Vpp = 1 V

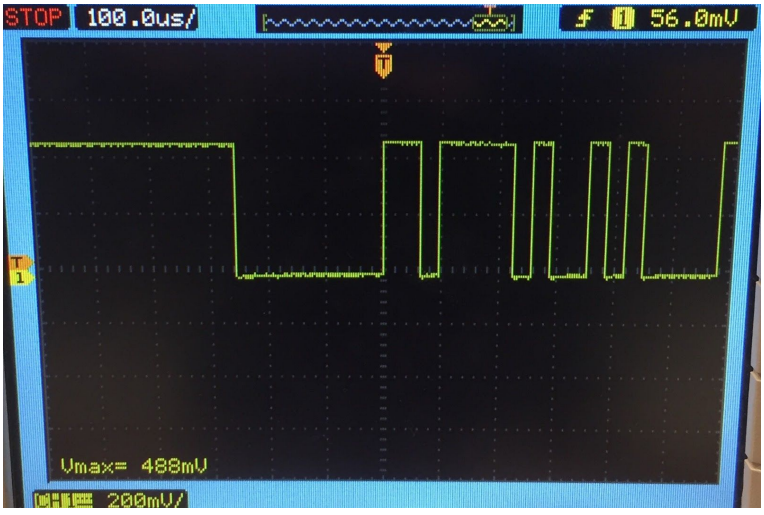


Figure 3 - The clean signal after optocoupler and inverters

Originally we were going to use I2C to send a message to a digital to analog converter which would in turn send a analog signal to the speakers and allow us to make more complex noises as well as play better sounding notes. This could have also eliminated the need for the potentiometers as the amplitude of the signal the DAC produces can be controlled through control messages.

Another important hardware peace was the big red button. This button was needed less on the final design but still came in handy when dealing with small bugs. In early development the button was just one of the buttons on the MSP that was configured as an interrupt to clear the buffer that was receiving the messages from the controller as well as reset all timers that were being used. This process was very useful in debugging where the board would occasionally start playing a high pitch noise and not shut off when the controller sent a note off message. The button would normally fix the issue by clearing the message buffer and disabling all outputs and timer interrupts. As was shown during the demo there was a bug that caused the synthesizer to completely stop working. This was a message buffer memory leak that would completely fill up memory after enough messages were received from the controller. This bug has been fixed in the updated code included in this report. A basic peripheral diagram can be seen in figure 4, showing the EUSCIB0 and 12-bit DAC connected with dotted lines to the speaker, signifying our planned implementation, and the TimerA0 and Digital I/O connection to the speaker as the back-up plan.

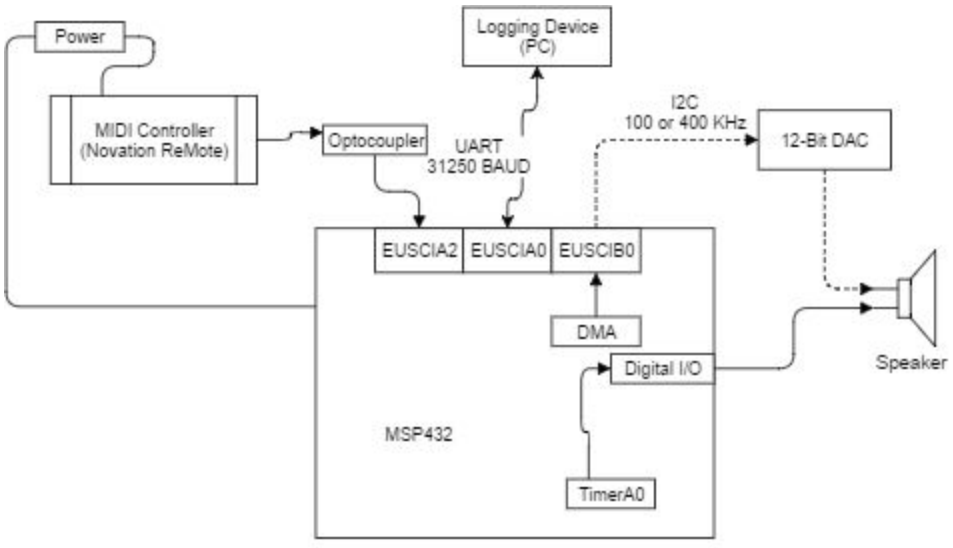


Figure 4 - Basic peripheral diagram

Software Description:

The MIDI standard provides the backbone of the digital synthesizer software, as the messages transmitted from a controller need to be interpreted into the familiar notes of a piano. MIDI messages are received from a MIDI controller over UART at 31250 BAUD. All configuration for this can be found in the configuration header and source files in the appendix. Messages are typically 2 to 3 bytes; one status byte and one or two data bytes. A status byte always has the first bit set, while data bytes always begin with 0. The enumeration type in figure 5 was created to easily decipher status messages and their meaning. This type is defined in the `midi` header file, which includes all of our MIDI communication tools. Also included in the `midi` header file and shown in figure 5 are structure types for complete messages and for a complete midi channel. The channel_message type contains the status of a received message, and all of the relevant data including pitch and velocity. The midi type contains a pointer to the current wavetable in use on the midi channel, and a pointer to the channel's output data. The output data was intended to be the superposition of all notes currently being played, allowing for polyphonic synthesis. As the DAC was never successfully implemented, this functionality was left as a placeholder for future expansion.

```
typedef enum channel_status_t {
      NOTE_OFF, // turn a note off
      NOTE_ON, // turn a note on
      AFTERTOUCH, // hitting note on again (bottoming out)
      CONTROL_CHANGE, // lever/knob value change
      PROGRAM_CHANGE, // wave_table change.
      CHANNEL_PRESSURE, // highest pressure key pressed.
      PITCH_BEND, // bend the note
}channel_status;

typedef struct channel_message_t {
    channel_status status; //note on, note off, etc.
    uint8_t channel; // channel number, 0x0 to 0xF (0-15)
    uint8_t pitch; // note
    uint8_t velocity; // how hard the note is hit
    uint8_t size;  // number of bytes in the message
}channel_message;

typedef struct midi_t {
    uint16_t * program; // wavetable currently in use.
    uint16_t * output; // current output waveform.
}midi;
```

Figure 5 - midi.h data types.

In order to process data received from the midi controller, a message buffer data type was created, following the first-in first-out protocol learned in class. The necessary functions for this buffer include initialization, adding and removing data, the boolean return functions `isFull()` and `isEmpty()`, and a clear/reset function. All of these declarations and definitions can be found in the message_buffer header and source files in the appendix. The EUSCIA2 RX interrupt simply places a byte of data into the global message buffer, and it is up to the main to handle and interpret the buffer data.

As our backup "plan B," voltage is output from a digital I/O pin directly to a speaker to generate sound. This was implemented using `TimerA0` interrupts to flip a pin at calculated capture compare values. The function `generateFakeWave(uint8_t pitch)` finds the corresponding frequency of the midi number `pitch` and sets the `TimerA0` capture compare value to the proper value. In configuration, the `TimerA0` source clock, `SMCLK`, is set to 12 MHz, which is more than fast enough for the highest frequency notes audible to the human ear. At certain lower frequency thresholds, the calculated capture compare value exceeds the 16-bit `TimerA0` capture compare register length. In these cases, a divider is set on the `TimerA0` clock, allowing for larger times between interrupts, thus lower frequencies can be played. Basically, the lower the frequency, the higher the divider. This function can be found in the `generate_fake_wave` header and source files, and the midi pitch to frequency function can be found in the `midi` header and source files, all in the appendix.

The main function of our project is ultimately very simple, due in part to interrupt functionality, the robust coding of included files, and to the simplicity of "plan B" over polyphonic synthesis. The watchdog timer is held, all peripheral configuration functions are called, a global message buffer is initialized for three byte messages, and interrupts are enabled globally. Following this code, the main enters a while-delay, waiting on the message buffer to be full. Once a message is completed, message items are assigned in the order received to the current message data structure. Then the code branches off using if statements based on the status of the current message. The "plan B" main function can be seen in figure 6 on the following page, and the entirety of `main.c`, showing all `#include` statements, macros, globally defined variables/data structures, and interrupt handlers can be seen in the appendix.

```
void main(void) {
  WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer
  configure_clocks();
  configure_pins();
  configure_UART();
  configure_timer();

  initializeBuffer(&message_buf, 3);
        // initialize buffer for 3 byte messages
  __enable_interrupt();
  uint8_t temp_buff; //status temporary buffer

  while(1) {
    while(!isFull(&message_buf));
    getItemFromBuffer(&message_buf, &temp_buff);
current_message.status = (channel_status)((temp_buff & STATUS_BIT_MASK) >> 4);
    getItemFromBuffer(&message_buf, &current_message.pitch);
    getItemFromBuffer(&message_buf, &current_message.velocity);

    if (current_message.status == NOTE_ON) {
        noteOn(&channel1, current_message.pitch, current_message.velocity);
        clearBuffer(&message_buf);
        P1OUT |= BIT0; // turn led on, note should be playing.
    }
    if (current_message.status == NOTE_OFF) {
        noteOff(&channel1, current_message.pitch);
        clearBuffer(&message_buf);
        P1OUT &= ~BIT0; // turn led off, no note should be playing.
    }
  }
}
```

Figure 6 - Main function of "Plan B" digital synthesis

Waveform Generation and Direct Memory Access (DMA):

The initial plan for this project was to use wavetable synthesis in conjunction with the DMA module to output data over I2C to a DAC. Wavetable synthesis would have allowed for a large range of precalculated or custom-made waveforms, and using the DMA would free up the limited MSP432 processor resources, allowing it to perform the calculations necessary for playing multiple notes at once (polyphony). Waveform generator functions, a couple of which can be seen in the `waveform_generator` header and source files, allocate space on the heap for a specified amount of data (also known as sample count). The sample count is generally a power of 2, for ease of computation. A channel's program is a pointer to this wavetable, which would have allowed for an easy and memory efficient way to switch programs at runtime.

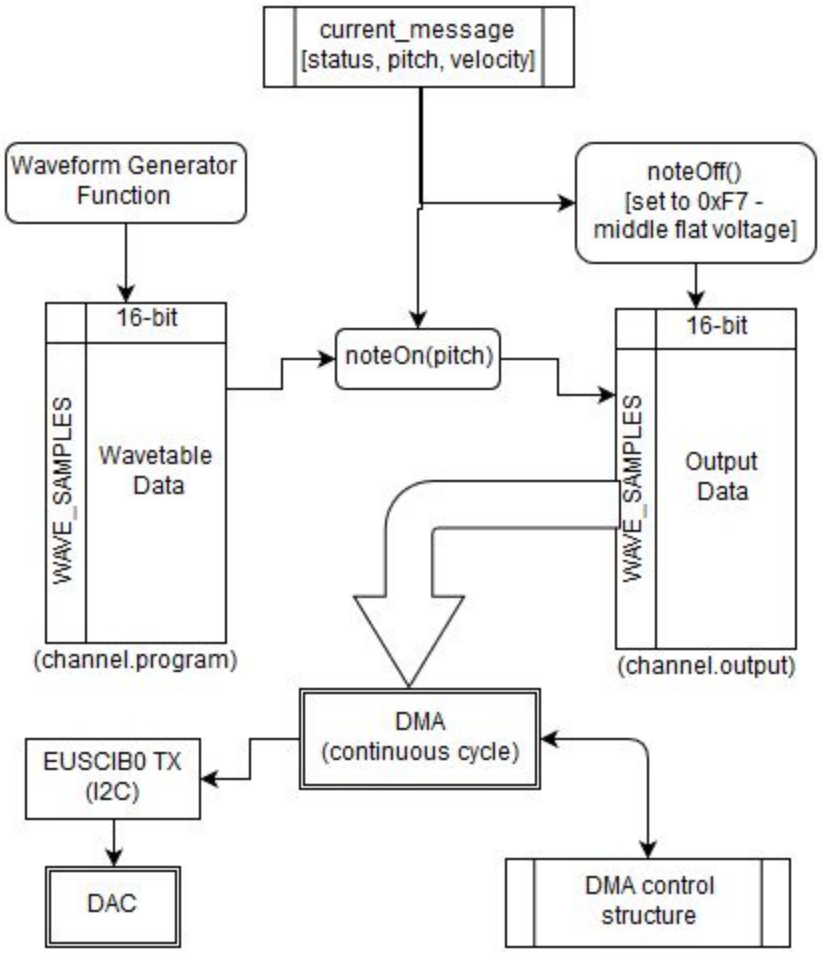


Figure 7 - Wavetable synthesis data flow, showing basic functions and peripheral interaction.

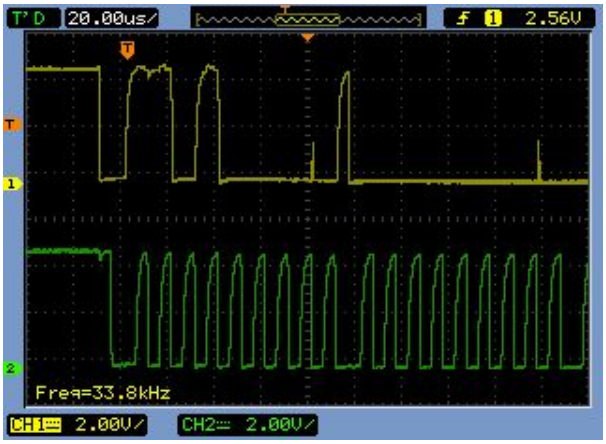
The `noteOn` function, called after a complete message is received, uses a phase increment to iterate through wavetable data, and places the required data into the output data array. A higher frequency note requires a larger phase increment, effectively "squeezing" the data together. A lower frequency note requires a smaller phase increment, which "stretches" the data out. The function fills the entire output array, and ensures the last piece of data loops back around to the first, eliminating skips or unintended frequency modulation. Ideally the `noteOn` function should superimpose it's pitch data onto the output data, however this wasn't implemented. The noteOff function sets all output data to `0x07FF`, a 12-bit flat middle voltage, turning off all sound. Again this function should ideally linearly subtract it's pitch data from the output data, but this wasn't implemented.

The DMA is controlled by its control structure which is allocated on the heap. It is given the end address of the data to be transferred, the address where the data should be placed, the expected size of the data, and the number of transfers to complete. Upon finishing a transfer cycle, the assigned DMA channel requests to repeat the process, and the module arbitrates the request based on priority. Because this process only uses one channel, it simply allows the channel to repeat the process. Unfortunately we couldn't figure out how to make the DMA automatically reset to the correct length of data, so an interrupt is thrown when the DMA completes a cycle, which resets the DMA to its initial length, thus creating a continuous cycle of output from the channel output array to the `EUSCIB0 TX` buffer. Because I2C never worked, this functionality was tested using `EUSCIA0` with UART. The DMA successfully transfers the correct data upon note-on and note-off messages, which we observed with RealTerm. If I2C worked, this data would be sent to the DAC, which would output sound. Because we were unable to measure frequency in this fashion, we are unsure if the code would be bug free in it's actual sound output. All of the code discussed can be seen in the configuration, midi, and main source files.

Extensions to the project:

I2C:

Our initial aim was to output sound with a digital to analog controller (DAC) using I2C. This would have allowed us to not only allow each key to play any sound we would have liked but also program a button on the MIDI to start playing a sequence of sounds on click. Unfortunately we were unable to get the I2C code to work. During testing we were able to see our clock cycle on the oscilloscope but not the start, stop and data bits were suppose to see. Following the technical reference manual, after initialization in master transfer mode, the MSP432 should begin communication by sending a start and addressing byte. A clock cycle is output for each bit sent. For some reason, the I2C bus was left busy without any output data, and the clock simply cycled endlessly. No combination of code ever resulted in any data sent along with the clock, and debugging always showed the I2C bus to be busy.



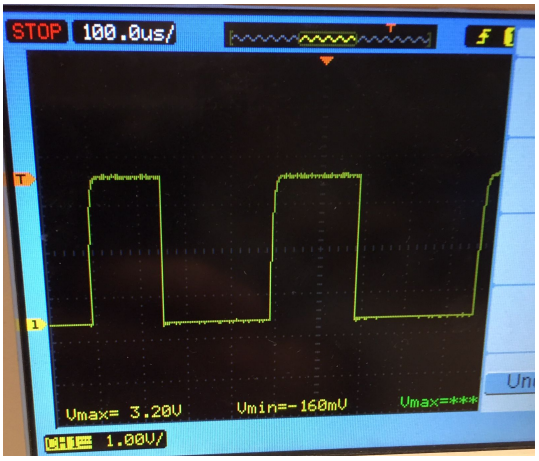


Figure (left) - showing what the data and clock output should have looked like, taken from: "What Happens If I Omit the Pullup Resistors on I2C Lines?" Atmega - What Happens If I Omit the Pullup Resistors on I2C Lines? - Electrical Engineering Stack Exchange. N.p., n.d. Web. 15 Dec. 2016.

Figure (Right) - showing what our data looks like.

From the figure on the left we can see the start, stop and data bits being shown with the wave on top, and the clock cycle with the wave on the bottom. The figure on the right shows what our data looks like, which is just the clock. Output data is not shown in this picture, however it was usually a flat reference voltage with some noise that seemed to correspond to the clock pulses. From our research we faced two issues with I2C. The initial issue was the pull-up resistor being high, which we were able to change by changing the value of the pull-up resistor. The second problem that we faced, which we were unable to fix within the time frame, was the bus being busy. The bus will always be busy as long as the stop bit is not being sent.

```
void configure_I2C(void) {
   UCB0CTLW0 = UCSWRST;
   UCB0CTLW0 |= UCMODE_3 | UCSSEL_2 | UCMST | UCSYNC;
   UCB0CTLW1 = EUSCI_B_CTLW1_ASTP_2;    // auto-stop
   UCB0TBCNT = 0x0002;   // number of bytes to be sent
   UCB0BR0 = 0x14;     // ~100khz
  UCB0CTLW0 &= ~UCSWRST;
   UCB0I2CSA |= DAC_SLAVE_ADDR;
   UCB0CTLW0 |= UCTXSTT | UCTR;
   UCB0IE |= UCRXIE | UCNACKIE | UCBCNTIE;
   NVIC_EnableIRQ(EUSCIB0_IRQn);
}
```

I2C Configuration function (not working)

Other possible additions to the project:

- Add a display that shows what note is being played.
- Allow users to play more than one note a time (polyphony).
- Program a button to play a sequence of pre-recorded sounds. This could be implemented with the timer module in our "plan B" implementation, or as a pre-made wavetable with the sound to be output with DMA/I2C/DAC implementation.
- Program certain keys to play a sound of our choice. Requires I2C/DAC.
- Add an amplifier to allow volume knobs to increase volume rather than just decrease the volume. This could be implemented in software with functioning DAC.
- Implement velocity, pitch bend, sound modulation, and other controls from MIDI. All require functioning DAC.
- Wirelessly connecting the speakers to the MIDI controller (would have required I2C).

Conclusion:

Ultimately we were satisfied with the final product of our project. It required a lot of work, debugging, and general anguish trying to make everything work together. Once we got notes playing out of a speaker, and they were the correct notes, it was very satisfying. We also had fun with the presentation of the synthesizer in a gift box. Although we faced failure in setting up our DAC, we realized success in most other areas of the project. We learned a great deal about data structures, interrupt handling, and working with a deadline. Additionally we learned a lot about sound and waveform production, MIDI/serial communication, and the basics of direct memory access modules. All of these things will contribute to each of us becoming better engineers and general problem solvers in our future careers.

Appendix: All code is presented in this section for the final implementation of the digital synthesizer. Sections of code that are used in the DMA/I2C/DAC implementation are included alongside the workaround code. These sections of code were not run at demo time, however they are included for completeness.

- Main.c -

```
#include "msp.h"
#include <stdlib.h>
#include "configuration.h"
#include "waveform_generator.h"
#include "midi.h"
#include "uart_put.h"
#include "message_buffer.h"
#include "fake_sqr_wave.h"

channel_message current_message;
message_buffer message_buf;
midi channel1;
uint8_t two_byte_flag = 0;

void main(void) {
	WDTCTL = WDTPW | WDTHOLD;  // Stop watchdog timer
	configure_clocks();
	configure_pins();
	configure_UART();
	configure_I2C();
	configure_timer();

	initializeBuffer(&message_buf, 3); // init buffer for 3 byte messages
	uint8_t temp_buff;
	uint32_t * dma_ctlbase = (uint32_t *)malloc(32);
	uint16_t * sqrWave = squareWave();
	initializeMidi(&channel1, sqrWave);
	configure_DMA(dma_ctlbase, ((uint32_t)channel1.output + (WAVE_SAMPLES * 2) - 1),
		0x4000100E); // currently address of EUSCIA0 TX Buffer
	__enable_interrupt();

	while(1) {
	 while(!isFull(&message_buf)); // wait for 3 byte message to be gathered.

	 // get bytes from buffer, place in current message in expected order.
	 getItemFromBuffer(&message_buf, &temp_buff);
	 current_message.status = (channel_status)((temp_buff & STATUS_BIT_MASK) >> 4);
	 getItemFromBuffer(&message_buf, &current_message.pitch);
	 getItemFromBuffer(&message_buf, &current_message.velocity);

	 // do things based on the status of the message.
	 if (current_message.status == NOTE_ON) {
		noteOn(&channel1, current_message.pitch, current_message.velocity);
		clearBuffer(&message_buf);

	// DMA needs to be re-enabled every time the output is changed, not sure why yet.
		*(uint32_t *)(DMA_Control->CTLBASE + 0x08) = DST_INC_NONE | XFER_SIZE |
			BASIC_MODE | DST_BUFFERABLE;
		DMA_Control->ENASET |= BIT0;
		P1OUT |= BIT0; // turn led on, note should be playing.
	 }
```

- main.c (cont.) -

```
      if (current_message.status == NOTE_OFF) {
            noteOff(&channel1, current_message.pitch);
            clearBuffer(&message_buf);
            *(uint32_t *)(DMA_Control->CTLBASE + 0x08) = DST_INC_NONE | XFER_SIZE |
                  BASIC_MODE | DST_BUFFERABLE;
            DMA_Control->ENASET |= BIT0;
            P1OUT &= ~BIT0; // turn led off, no note should be playing.
      }
     }
}

void EUSCIA0_IRQHandler(void) {
    /* EUSCIA0 UART is used for logging, and if desired sending midi messages.
     * device will accept these messages as if they were received from
     * a midi controller. */
    uint8_t rxdata;
    if(UCA0IFG & UCRXIFG) {
      UCA0IFG &= ~UCRXIFG;
      if (!two_byte_flag) {
            uart_putchar(UCA0RXBUF); // log function here.
            addItemToBuffer(&message_buf, UCA0RXBUF);
      }
      else if (two_byte_flag == 1) {
            two_byte_flag = 0;
            clearBuffer(&message_buf);
      }
      if (rxdata == 0xD0) {
            two_byte_flag++;
      }
    }
}

void EUSCIA2_IRQHandler(void) {
     /* EUSCIA2 UART comms with the midi controller (31250 BAUD)
      * The RX interrupt logs received messages over EUSCIA0 and
      * places the received data into the message buffer.
      * two_byte_flag ensures 2 byte messages do not cause sync
      * issues with the message buffer. */
    uint8_t rxdata;
    if(UCA2IFG & UCRXIFG) {
      UCA2IFG &= ~UCRXIFG;
      if (!two_byte_flag) {
            uart_putchar(UCA2RXBUF); // log function here.
            addItemToBuffer(&message_buf, UCA2RXBUF);
      }
      else if (two_byte_flag == 1) {
            two_byte_flag = 0;
            clearBuffer(&message_buf);
      }
      if (rxdata == 0xD0) {
            two_byte_flag++;
      }
    }
}

void EUSCIB0_IRQHandler(void) {
    // DAC -> I2C interrupts
    if(UCB0IFG & UCRXIFG) {
      __no_operation();
    }
```

- main.c (cont.) -

```
    if(UCB0IFG & UCTXIFG) {
      __no_operation();
    }
}

void DMA_INT1_IRQHandler(void) {
    // this interrupt is called when the DMA completes a cycle through the whole waveform. I think
    //there's a way to make it repeat cycles without interrupt, but I haven't figured it out yet.
    *(uint32_t *)(DMA_Control->CTLBASE + 0x08) = DST_INC_NONE|XFER_SIZE|BASIC_MODE|DST_BUFFERABLE;
    DMA_Control->ENASET |= BIT0;
}

void TA0_0_IRQHandler(void) {
    // flip a pin for square wave generation
    TA0CCTL0 &= ~CCIFG;
    P6OUT ^= BIT5|BIT4;
}

void DMA_ERR_IRQHandler(void) {
    // called when DMA controller hits an error. should perform logging.
    __no_operation();
}

void PORT2_IRQHandler(void) {
    // big red button emergency reset:
    // this clears the message buffer, turns off all output, and sets the
    // timer CCR value arbitrarily high so that it may exit a loop
    uint16_t i;
    if (P2IFG & BIT7) {
      P2IFG &= ~(BIT7);
      for(i = 0; i < 2000; i++);
      clearBuffer(&message_buf);
      noteOff(&channel1, 1);
      P1OUT &= ~BIT0;
      TA0CCR0 |= 9000;
    }
}
```

- configuration.h -

```
#ifndef CONFIGURATION_H_
#define CONFIGURATION_H_

#define DAC_SLAVE_ADDR (0x60)
#define SYSCLK (12000000)
#define BAUD_RATE (31250) // standard midi baud rate is 31250

// DMA macros
#define DST_INC_NONE 0xC0000000
#define DST_BUFFERABLE 0x00600000
#define ARB_SIZE_1024 0x00028000
#define XFER_SIZE 0x000007F0  // n-1 transfers  07F = 128 - 1
#define XFER_AUTO_REQ 0x00000002
#define BASIC_MODE 0x00000001

void configure_pins(void);

void configure_UART(void);

void configure_I2C(void);

void i2c_transmit(void);
```

```
void configure_timer(void);

void configure_DMA(uint32_t * ctlbase, uint32_t src_end_address, uint32_t dst_end_address);
    //ctlbase should be preallocated with 12 bytes to store DMA configuration.

void configure_clocks(void);
#endif /* CONFIGURATION_H_ */
```

- configuration.c -

```
#include "msp.h"
#include "configuration.h"
#include <stdlib.h>

void configure_pins(void) {
    P1DIR |= BIT0;          // 1.0: red led (note on indication)
    P1OUT &= ~BIT0;
    P6DIR |= BIT5|BIT4;  // 6.4: wave generation pin (speaker)
    P6OUT &= ~BIT5|BIT4; // 6.5: wave generation pin (speaker)

    //i2c (euscib0)
    P1SEL0 |= (BIT6 | BIT7);
    P1SEL1 &= ~(BIT6 | BIT7);
    P1DIR &= ~(BIT6 | BIT7);
    P1REN |= (BIT6 | BIT7);
    P1OUT |= (BIT6 | BIT7);

    // uart (eusciA0 and eusciA2)
    P1SEL1 &= ~(BIT2|BIT3);
    P1SEL0 |= BIT2|BIT3;
    P3SEL1 &= ~(BIT2|BIT3);
    P3SEL0 |= BIT2|BIT3;

      // on-board buttons
    P1DIR &= ~(BIT1|BIT4);
    P1OUT |= BIT1|BIT4;
    P1REN |= BIT1|BIT4;
    P1IFG &= ~(BIT1|BIT4);
    P1IES |= BIT1;
    P1IE |= BIT1|BIT4;
    NVIC_EnableIRQ(PORT1_IRQn);
      // big red button
    P2DIR &= ~BIT7;
    P2OUT |= BIT7;
    P2REN |= BIT7;
    P2IFG &= BIT7;
    P2IES |= BIT1;
    P2IE |= BIT7;
    NVIC_EnableIRQ(PORT2_IRQn);
}

void configure_UART(void) {
    UCA0CTLW0 |= UCSWRST;  // Put eUSCI in reset mode
    UCA0CTLW0 |= EUSCI_A_CTLW0_SSEL__SMCLK;
    UCA0BR0 = 0x80;  // baud rate, should be parameterized
    UCA0BR1 = 0x01;         // overflow
    UCA0CTLW0 &= ~UCSWRST; // Initialize eUSCI
    UCA0IE &= ~UCTXIE;
    UCA0IE |= UCRXIE;       // enable USCI_A0 RX interrupts.
    NVIC_EnableIRQ(EUSCIA0_IRQn);
    UCA2CTLW0 |= UCSWRST;
    UCA2CTLW0 |= EUSCI_A_CTLW0_SSEL__SMCLK;
```

- configuration.c (cont.) -

```
    UCA2BR0 = 0x80;
    UCA2BR1 = 0x01;
    UCA2CTLW0 &= ~UCSWRST;
    UCA2IE &= ~UCTXIE;
    UCA2IE |= UCRXIE;
    NVIC_EnableIRQ(EUSCIA2_IRQn);
}

void configure_DMA(uint32_t * ctlbase, uint32_t src_end_address, uint32_t dst_end_address) {
    DMA_Control->CTLBASE = (uint32_t)ctlbase;
    *(uint32_t *)(DMA_Control->CTLBASE + 0x08) = DST_INC_NONE|XFER_SIZE|BASIC_MODE|DST_BUFFERABLE;
    *(uint32_t *)(DMA_Control->CTLBASE + 0x04) = dst_end_address;
    *(uint32_t *)(DMA_Control->CTLBASE) = src_end_address;
    DMA_Channel->CH_SRCCFG[0] = 1; // source: euscia0 TX interrupt flag
    DMA_Channel->INT1_SRCCFG = DMA_INT1_SRCCFG_EN; // channel 0 finished interrupt
    DMA_Control->ENASET |= BIT0; // enables channel 0 of DMA
    DMA_Control->CFG |= BIT0; // enable DMA controller
    NVIC_EnableIRQ(DMA_ERR_IRQn);
    NVIC_EnableIRQ(DMA_INT1_IRQn);
}

void configure_I2C(void) {
     UCB0CTLW0 = UCSWRST;                          // Enable SW reset
     UCB0CTLW0 |= EUSCI_B_CTLW0_MODE_3 | UCSSEL_2 | UCMST | UCSYNC;
              // I2C mode, smclk, I2C Master, transmitter, synchronous mode
     UCB0CTLW1 = EUSCI_B_CTLW1_ASTP_2;    // auto-stop enable
      UCB0TBCNT = 0x0002;                   // number of bytes to be received
      UCB0BR0 = 0x14;                       // ~100khz
      UCB0CTLW0 &= ~UCSWRST;              // Clear SW reset, resume operation
      UCB0I2CSA |= DAC_SLAVE_ADDR;  // set slave address
      UCB0CTLW0 |= UCTXSTT | UCTR;
      UCB0IE |= UCRXIE | UCNACKIE | UCBCNTIE;     //Enable RX and TX interrupt
     NVIC_EnableIRQ(EUSCIB0_IRQn);
}

void configure_timer(void) {
    // timer is reconfigured at run time by note on/off messages.
    TA0CTL |= TASSEL_2 | TIMER_A_CTL_MC__UP;  // SMCLK, up-down mode
    NVIC_EnableIRQ(TA0_0_IRQn);
}

void i2c_transmit(void){
    //while (UCB0CTLW0 & UCTXSTP);         // Ensure stop condition got sent
    while(UCB0STAT & UCBBUSY);
    //UCB0CTLW0 |= UCTXSTT;                // Transmit mode and generate start
    UCB0TXBUF = 0xAA;
    while(!(UCB0IFG & UCTXIFG));
    UCB0TXBUF = 0xFF;
    UCB0CTLW0 |= UCTXSTP;                          // Stop condition
}

void configure_clocks(void) {
    CS->KEY = 0x695A;              // unlock CS module for register access
    CS->CTL0 = 0;                  // reset tuning parameters
    CS->CTL0 = CS_CTL0_DCORSEL_3; // set DC0 to 12 MHz (middle of 8-16 range)
    // Select ACLK = REPO, SMCLK = MCLK = DCO
    CS->CTL1 = CS_CTL1_SELA_2 | CS_CTL1_SELS_3 | CS_CTL1_SELM_3;
    CS->KEY = 0;                   // lock CS module for register access
}
```

- uart_put.h -

```
#ifndef UART_PUT_H_
#define UART_PUT_H_
#include <stdint.h>

void uart_putchar(uint8_t tx_data);
void uart_putchar_n(uint8_t * data, uint32_t length);
void uart2_putchar(uint8_t tx_data);

#endif /* UART_PUT_H_ */
```

- uart_put.c -

```
#include "uart_put.h"
#include "msp.h"

void uart_putchar_n(uint8_t * data, uint32_t length) {
    volatile uint32_t i;
    for(i = 0; i < length; i++) {
      uart_putchar(*data++);
    }
}

void uart_putchar(uint8_t tx_data) {
    UCA0TXBUF = tx_data;    // load data onto buffer.
}

void uart2_putchar(uint8_t tx_data) {
      //eusciA2 putchar function.
    UCA2TXBUF = tx_data;    // load data onto buffer.
}
```

- midi.h -

```
#ifndef MIDI_H_
#define MIDI_H_

#define STATUS_BIT_MASK 0x70 // remember to shift right 4 bits to get value of status
#define CHANNEL_BIT_MASK 0x0F // not used for now

typedef enum channel_status_t {
    NOTE_OFF, // turn a note off
    NOTE_ON, // turn a note on
    AFTERTOUCH, // pressing a key after it's already on (bottoming out)
    CONTROL_CHANGE, // lever/knob value change
    PROGRAM_CHANGE, // patch number change. this specifies the sound used.
    CHANNEL_PRESSURE, // highest pressure key of all keys pressed. we probably dont need it.
    PITCH_BEND, // bend the note
}channel_status;

typedef struct channel_message_t {
    channel_status status; //note on, note off, etc.
    uint8_t channel; // channel number, 0x0 to 0xF (0-15)
    uint8_t pitch; // note
    uint8_t velocity; // how hard the note is hit
    uint8_t size;  // number of bytes in the message
}channel_message;

typedef struct midi_t {
    uint16_t * program; // ptr to wavetable currently in use.
    uint16_t * output; // ptr to current output wave data
}midi;
```

```
void initializeMidi(midi * channel, uint16_t * wavetable);
/*     initializeMidi is called after a midi channel is declared and a wavetable is generated.*/

float midiFrequency(uint8_t pitch);
/* returns the designated frequency of a midi number pitch */

uint8_t phaseIncrement(uint8_t pitch);

void noteOn(midi * channel, uint8_t pitch, uint8_t velocity);

void noteOff(midi * channel, uint8_t pitch);

#endif /* MIDI_H_ */
```

- midi.c -

```
#include "msp.h"
#include "waveform_generator.h"
#include "midi.h"
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include "fake_sqr_wave.h"

void initializeMidi(midi * channel, uint16_t * wavetable) {
    uint32_t i;
    channel->program = (uint16_t *)calloc(WAVE_SAMPLES, 2);
    channel->output = (uint16_t *)calloc(WAVE_SAMPLES, 2);
    memmove(channel->program, (void*)wavetable, WAVE_SAMPLES * 2);
    for(i = 0; i < WAVE_SAMPLES; i++){
      *(channel->output + i) = MAX_N_DAC / 2;
    }
}

float midiFrequency(uint8_t pitch) {
    return pow(2.0, (((float)pitch - 69.0) / 12.0)) * 440.0;
}

uint8_t phaseIncrement(uint8_t pitch) {
    return (WAVE_SAMPLES * midiFrequency(pitch))/SAMPLE_RATE;
    // truncated for now. this can get complicated if we want it more precise.
}

void noteOn(midi * channel, uint8_t pitch, uint8_t velocity) {
    generateFakeWave(pitch); // work-around for I2C not working.

    // wavetable synthesis:
    uint8_t i;
    uint8_t p = phaseIncrement(pitch);
    for (i = 0; i < WAVE_SAMPLES; i++) {
      // mod wave_samples so if pointer exceeds length of wavetable it wraps around.
      *(channel->output + i) = channel->program[((p*i) % WAVE_SAMPLES)];
    }
}

void noteOff(midi * channel, uint8_t pitch) {
    TA0CCTL0 &= ~CCIE; // disable timer interrupts
    P1OUT &= ~BIT5; // ensure output off.

    // DAC note off:
    uint8_t i;
```

```
    for (i = 0; i < WAVE_SAMPLES; i++) {
      *(channel->output + i) = MAX_N_DAC / 2; // flat middle voltage
    }
}
```

- message_buffer.h -

```
#ifndef MESSAGE_BUFFER_H_
#define MESSAGE_BUFFER_H_

typedef struct message_buffer_t{
    uint8_t * buffer;
    uint8_t * first;
    uint8_t * last;
    uint8_t length;
    uint8_t max_size;
}message_buffer;

typedef enum buffer_test_t{
    SUCCESS,
    BUFFER_FULL,
    BUFFER_EMPTY,
    ALLOCATION_ERROR
}buffer_test;

buffer_test initializeBuffer(message_buffer * buf, uint8_t max_size);

buffer_test addItemToBuffer(message_buffer * buf, uint8_t data);

buffer_test getItemFromBuffer(message_buffer * buf, uint8_t * container);
/* removes the oldest item in the buffer, places in container.*/

void clearBuffer(message_buffer * buf);

uint8_t isEmpty(message_buffer * buf);

uint8_t isFull(message_buffer * buf);

#endif /* MESSAGE_BUFFER_H_ */
```

- message_buffer.c -

```
#include "msp.h"
#include "message_buffer.h"
#include <stdlib.h>

buffer_test initializeBuffer(message_buffer * buf, uint8_t max_size) {
    buf->buffer = (uint8_t *)malloc(max_size);
    if(!buf->buffer) {
      return ALLOCATION_ERROR;
    }
    buf->first = buf->buffer;
    buf->last = buf->buffer;
    buf->max_size = max_size;
    buf->length = 0;
    return SUCCESS;
}

void clearBuffer(message_buffer * buf) {
    buf->first = buf->buffer;
    buf->last = buf->buffer;
    buf->length = 0;
}

uint8_t isEmpty(message_buffer * buf){
```

- message_buffer.c (cont.) -

```
    if(buf->length == 0) {
      return 1;
    }
    else return 0;
}

uint8_t isFull(message_buffer * buf){
    if(buf->length == buf->max_size) {
      return 1;
    }
    else return 0;
}

buffer_test addItemToBuffer(message_buffer * buf, uint8_t data) {
    if(isFull(buf)) {
      return BUFFER_FULL;
    }
    buf->length++;
    *buf->first = data;
    buf->first++;
    return SUCCESS;
}

buffer_test getItemFromBuffer(message_buffer * buf, uint8_t * container) {
    if(isEmpty(buf)) {
      return BUFFER_EMPTY;
    }
    *container = *buf->last;
    buf->last++;
    if(buf->last > (buf->first)) {
      buf->last = buf->buffer; // edge
      buf->first = buf->buffer;
      buf->length = 0;
      return SUCCESS;
    }
    buf->length--;
    return SUCCESS;
}
```

- waveform_generator.h -

```
#ifndef WAVEFORM_GENERATOR_H_
#define WAVEFORM_GENERATOR_H_

#define WAVE_SAMPLES 128 // max samples allocated for each waveform:
                         //** when this is changed, DMA XFER COUNT must be updated as well! **//
#define SAMPLE_RATE 5000 // DAC sample rate
#define MAX_N_DAC 0x0FFF  // max DAC integer

/*  WAVE GENERATOR FUNCTIONS:
 *   -> always return a pointer to an allocated array of 16bit wave values on the heap (wavetable)
 *   -> pointer should be passed into midiInitialization() or program change function to be assigned
 *      to a midi channel
 *   -> generated wavetables remain allocated for reuse, but can be deallocated using free(). */
uint16_t * sineWave();
uint16_t * squareWave();
uint16_t * triangleWave();
uint16_t * sawWave();

#endif /* WAVEFORM_GENERATOR_H_ */
```

- waveform_generator.c -

```c
#include "waveform_generator.h"
#include <msp.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>

#define PI (3.14159265359)

uint16_t * squareWave() {
    uint16_t * sqrWave = (uint16_t *)calloc(WAVE_SAMPLES, 2);
    if (!sqrWave) {
      return 0;  // logging or pin flip for allocation error
    }
    uint16_t i;
    for (i = 0; i < WAVE_SAMPLES; i++) {
      if (i > WAVE_SAMPLES / 2) {  // half-way through period, symmetrical square.
             sqrWave[i] = MAX_N_DAC;
      }
      else {
             sqrWave[i] = 0;
      }
    }
    return sqrWave;
}

uint16_t * sineWave() {
    uint16_t * sinWave = (uint16_t *)calloc(WAVE_SAMPLES, 2);
    uint16_t i;
    float phase = 0;
    for (i = 0; i < WAVE_SAMPLES; i++) {
      sinWave[i] = (MAX_N_DAC/2) + ((MAX_N_DAC/2) * sin(phase));
      phase = phase + ((2.0 * (float)PI) / (float)WAVE_SAMPLES); // increment by the phase resolution
    }  // phase should never exceed 2*pi or else it won't wrap around.
    return sinWave;
}

uint16_t * sawWave() {
    return 0; // implement later
}

uint16_t * triangleWave() {
    return 0;  // implement later
}
```

- fake_sqr_wave.h -

```c
#ifndef FAKE_SQR_WAVE_H_
#define FAKE_SQR_WAVE_H_

void generateFakeWave(uint8_t pitch);

#endif /* FAKE_SQR_WAVE_H_ */
```

- fake_sqr_wave.c -

```c
#include "msp.h"
#include "fake_sqr_wave.h"
#include "configuration.h"
#include "waveform_generator.h"
#include "midi.h"
#include <stdio.h>
```

- fake_sqr_wave.c (cont.) -

```
void generateFakeWave(uint8_t pitch) {
    uint16_t freq = (uint16_t)midiFrequency(pitch);
    uint16_t divider;
    if (freq < 93) {
      TA0CTL &= ~TIMER_A_CTL_ID_MASK;
      TA0CTL |= TIMER_A_CTL_ID_3;  // divide clk by 4 (3 MHz)
      divider = (((SYSCLK/4)/freq)/2);
      if (divider > 50) {
             TA0CCR0 = divider;
             TA0CCTL0 |= CCIE;
      }
    }
    else if (freq < 184) {
      TA0CTL &= ~TIMER_A_CTL_ID_MASK;
      TA0CTL |= TIMER_A_CTL_ID_2; // divide clk by 2 (6 MHz)
      divider = (((SYSCLK/2)/freq)/2);
      if (divider > 50) {
             TA0CCR0 = divider;
             TA0CCTL0 |= CCIE;
      }
    }
    else {
      TA0CTL &= ~TIMER_A_CTL_ID_MASK;  // no clk division (12 MHz)
      divider = (SYSCLK/freq)/2;
      if (divider > 50) {
             TA0CCR0 = divider;  // count up to this value, interrupt will flip pin.
             TA0CCTL0 |= CCIE;   // enable timer interrupts
      }
    }
}
```