Midi Recorder and Sequencer

CS122A: Fall 2018

David Swanson

Table of Contents

Introduction **2**

Hardware **2**

- Parts List 2
- Block Diagram 2
- Pinout (For each microcontroller/processor) 3

Software **3**

Implementation Reflection **4**

- Milestone 4
- Completed components 4
- Incomplete components 4

Youtube Links **4**

Testing **5**

Known Bugs **5**

Resume/Curriculum Vitae (CV) Blurb **5**

Future work **5**

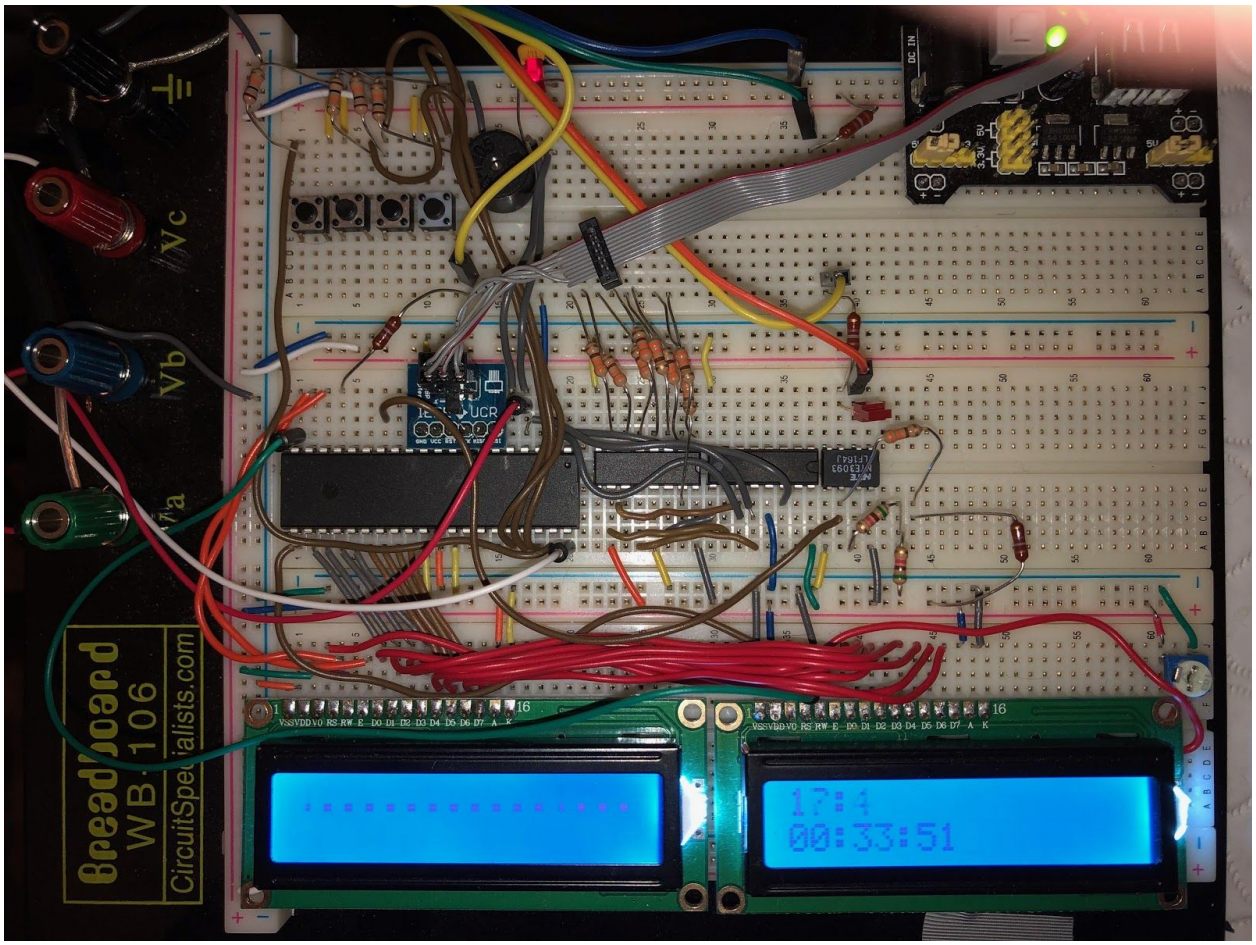
Extra Credit **6**

References **6**

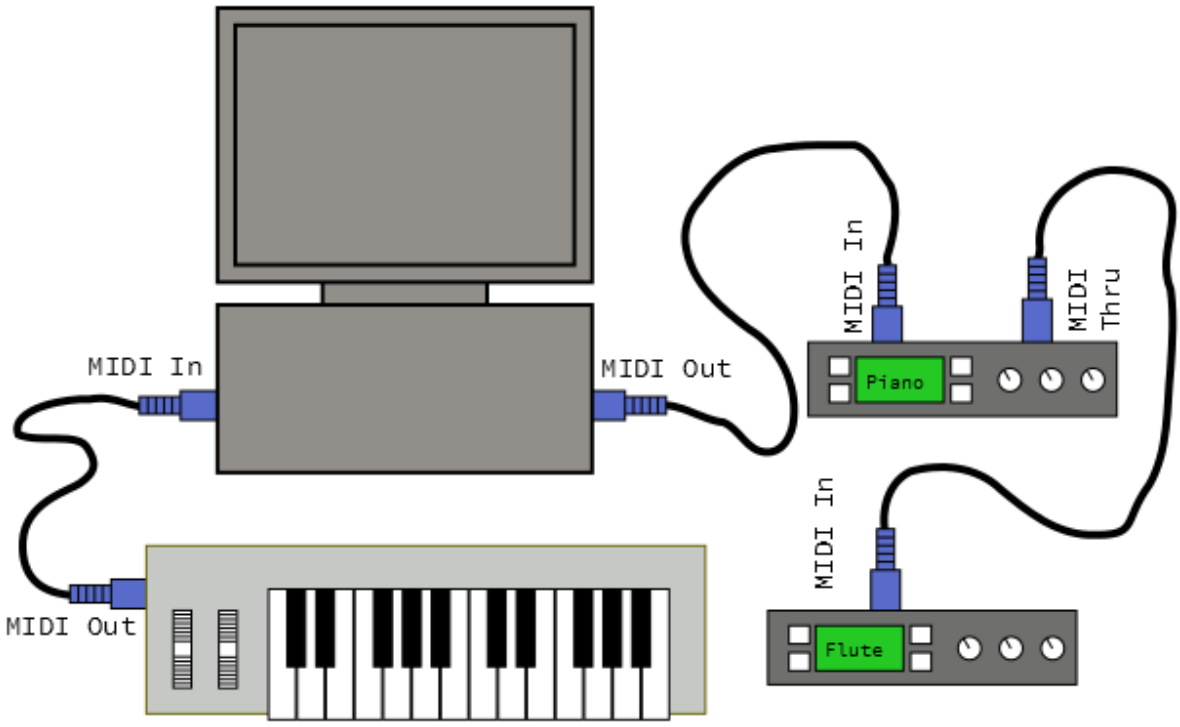
Appendix **6**

Introduction

Midi is a standard developed in the 1980's that allows storage of music in smaller files than audio recordings. Uncompressed audio recordings require at least 176 kilobytes of storage per second (16-bits * 44,100 samples/second * stereo) In contrast, Midi file sizes are on the order of text files. They store a description of a musical performance, not the performance itself. The assumption is that audio will be generated by external devices designed for that purpose. Besides smaller files, the other advantage of Midi is that the sounds can be changed after the performance, modularizing the composition process. Understandably, Midi reshaped music quickly after its introduction.



This diagram shows how to hook a keyboard to a sound module. My project plays the role of the computer, passing messages through and recording them.



In order to complete this project I had to:

- Learn the midi protocol to sort out and merge various message types and lengths.
- Master interrupts and timing on the ATMega1284.

I also tried a lot of cool tricks, such as:

- Passing function pointers to change machine modes.
- Running state machines from their own interrupts or on top of other state machines.
- Altering the given LCD, USART and scheduler code to suit this particular application.

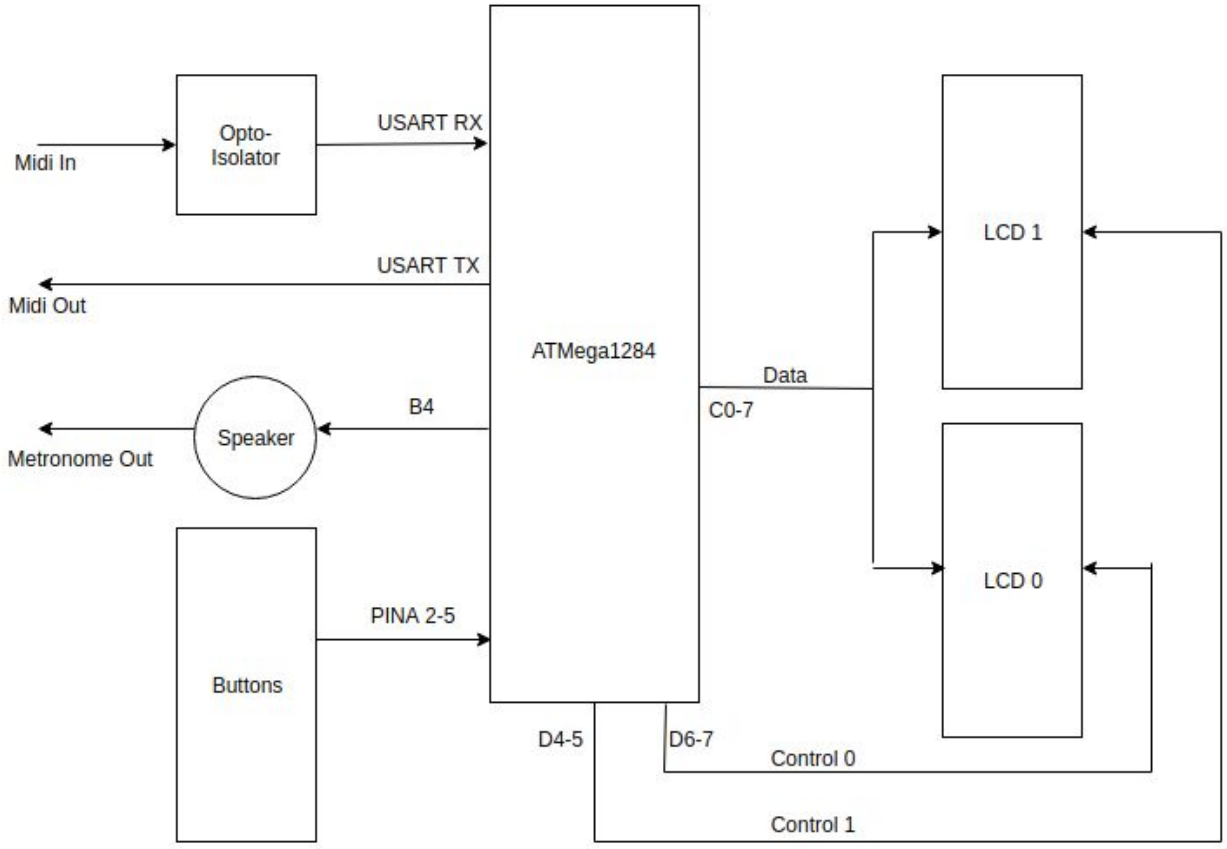
Hardware

Parts List

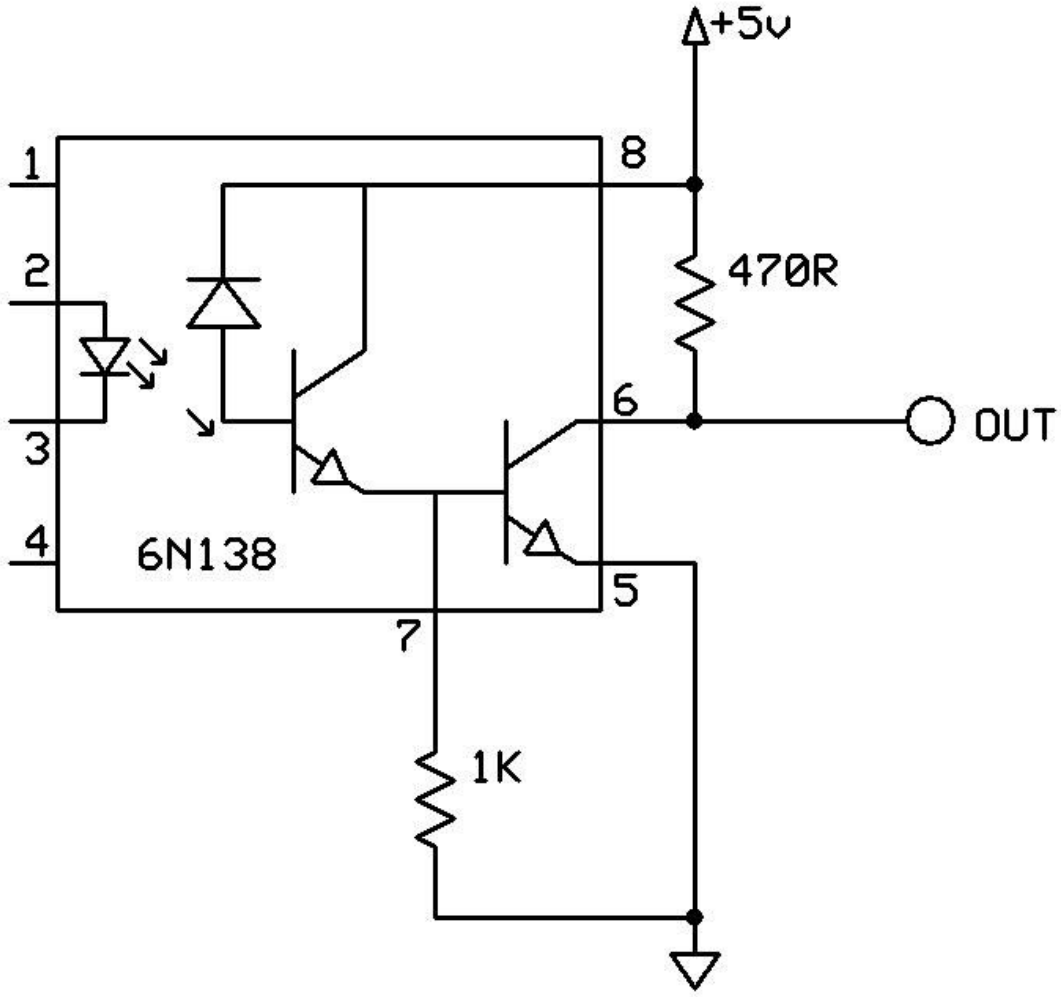
The hardware that was **used** in this project is listed below. The equipment that was not taught in this course has been bolded. *Include part numbers when available*.

Part	Part #	Quantity	Price (optional)
ATMega1284	ATMega1284	1	$5
Opto-coupler	6N138	1	$5
LCD Screen	1602A1	2	$3 * 2
Speaker	CLS0231-L152	1	
Momentary Buttons		4	
		Total	

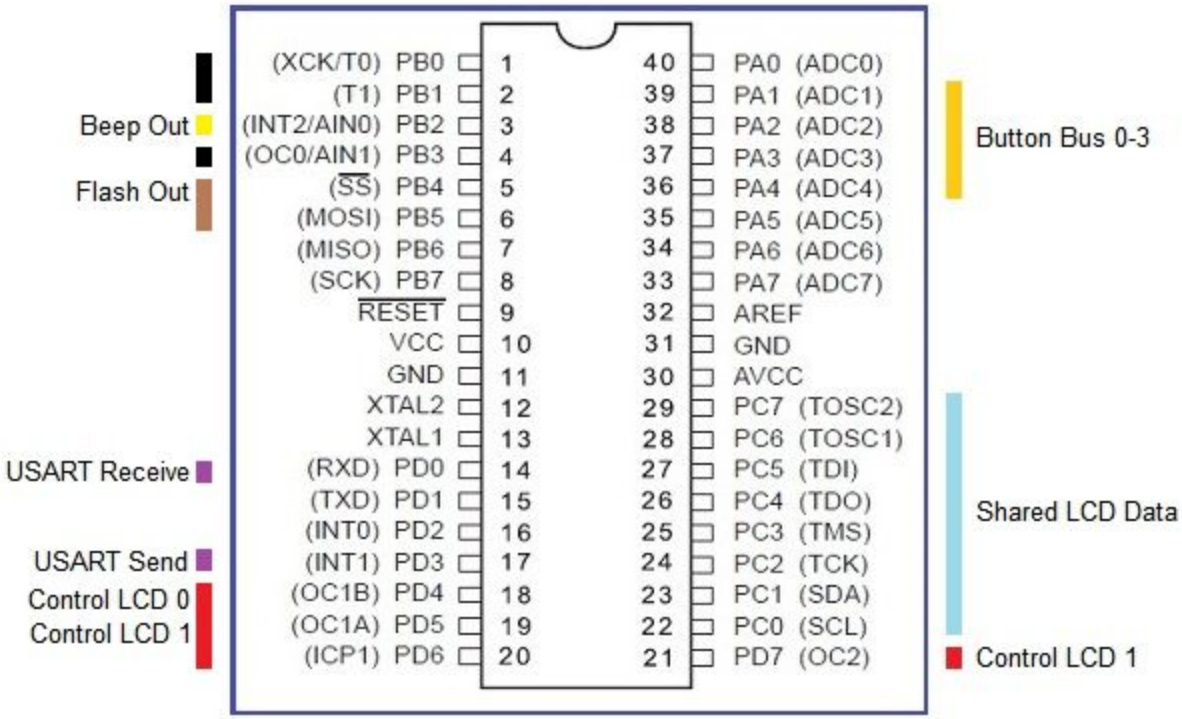
Block Diagram



Opto-isolator hookup

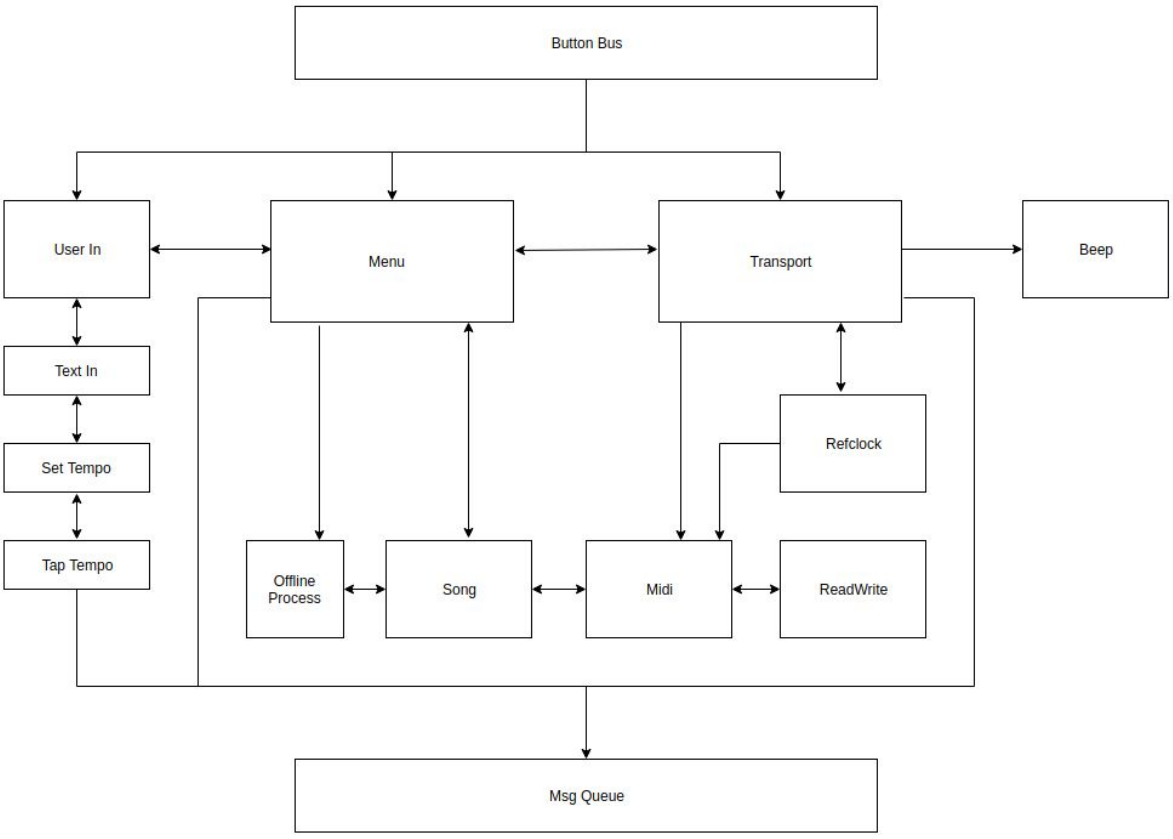


Pinout (For each microcontroller/processor)



Software

The software designed for this project was implemented using the PES standard.
https://github.com/sugarvillela/ATMega1284
The overall design as a task diagram is included below.



- Dual LCD display with 4-message queue
    - Messages are timed to display and clear after a set interval.
    - Each LCD screen can run a message while three more are waiting.
    - Ability to disable queues, allowing standard LCD writes.
    - An ‘Err’ mode to override disabled queue for error messages, then restore state.
    - Can bind function pointers to time out, triggering some action on clear screen.
    - LCDs share the same tick function, holding state in an array of structs.
    - This is a ‘general’ component, more than is needed for this project. I ended up using the ‘simple’ mode for most of it, where message timing is user-controlled or linked to the metronome.
- Button bus with up to 8 buttons (4 used in this project)
    - The same button-polling code is used for all device modes.
    - Assignable start point (I used pins A2-A5) to reserve A1 for A/D input.

    - Adjustable debounce time, for reliable clicks.
    - Each button records number of presses and counts hold time.
    - Button states pollable: isPressed(), wasPressed() etc. Some states are sticky, meaning their values are reset by whoever is polling.
    - Button functions assignable, with function pointers as described already. Can bind to button press, double-press or hold.
    - Buttons share the same tick function, holding state in an array of structs.
    - This is another 'general' component, useful for any project.
    - In general, the four buttons are left, down, up, right. Hold left exits a function; hold right triggers some special function. Hold up or down increments or decrements faster.

- Audible metronome with reference clock, for display and timestamping
    - Beep on record and silent on play. You can change that in the menu.
    - Displays measures, beats, time, and flashing REC/PLAY/STOP
    - Displays moving marker to indicate beat position.
    - Reference clock counts 10 ms increments, seconds and minutes.
    - Actually three state machines that work together: transport, refclock and beep (not using the PWM).

- Interrupt-based USART for midi messages
    - Incoming messages are sent to the queue (recording).
    - A copy of every incoming message is sent from queue to output (echo).
    - Playback messages are sent to output, merging with echo messages.

- Data storage in RAM, implemented on a single array.
    - Playback data is read from the bottom of the array.
    - Recorded data is sent to the top of the array.
    - Echo data is read from the top just after it arrives.
    - On stop, the array is 'defragged', sorted by timestamp.

- Data storage in eeprom
    - Header contains song title, tempo and data size.
    - Data is copied in and out of eeprom using AVR block functions
    - Menu preferences are saved as flags in a single byte.

- User menu across both screens, implemented as a two-dimensional linked list
    - Each 'leaf' has a function pointer pointing to an action.
    - Where appropriate, menu items display the item setting or state.
    - The four buttons control left, down, up, right.
    - Menu items listed in the next section.

- User Input (three modes)

    - Text input: like finding a movie in Netflix. Supports [0-9A-Z]. Hold up/down increments/decrements faster. Right button saves, left button exits.
    - Numeric input: up/down sets a value. Hold up/down increments/decrements faster. Right button saves, left button exits.
    - Tap input: inverse of the average time between button clicks. Right button saves, left button exits. Either middle button taps.

- Offline actions with progress bar (more than an Easter Egg)
    - Quantize, change tempo, array sort and file dump are run offline because they may take longer than a tick.
    - Interrupts are disabled by the progress bar function, re-enabled on process complete.

- Event list
    - Decodes and displays stored midi messages with note name, velocity and time (in refclock ticks).
    - Scrolls through a long list of values using up/down buttons. Hold left button to exit.

Implementation Reflection

One lecture about polling stuck with me and influenced my design. Thereafter I deliberately avoided the scenario of one machine polling an input while a second machine polls the first one. Instead I allow the second machine to pass a function pointer to the first one, so the one polling can call the function directly. Similarly, polling the USART continuously makes less sense than using the transmit and receive interrupts designed for that purpose. These arrangements lead to a more event-driven system, with fewer machines hooked to the main timer loop.

With multiple interrupts come concurrency issues. I like the way this class begins to connect to CS153 in that respect. Also, the project's handling of bulk data brought up other issues like file systems and fragmentation, familiar from CS153 and CS161.

Another major change I made was to take the state out of the task structures; tick functions now return void rather than integer. I keep a lot of data for each function in a struct in that function's c file. It made more sense to keep the state there too, especially since some functions (LCDs and buttons) use arrays of structs and states. I considered pulling inactive tasks off the main task array, but decided it would not save much time.

What was difficult about the project was knowing where in the system to look for problems. The optocoupler seemed to be a source of uncertainty, as the resistor values set a balance between signal voltage and response time. The rate at which messages arrive was also a worrisome point. It initially appeared that I was skipping incoming messages, but upon further research I

found out midi uses ‘shorthand’. This is a standard in which it leaves duplicate on and off messages out of the stream. So in the end I was not missing anything.

If I did the project again I would have better design goals. Initially I added some complexity to the project for complexity’s sake, but none of it was necessary. Just four buttons and a generous LCD display are sufficient for the human interface. The project followed a path perhaps similar to a real company’s development of a production device, cutting superfluous and expensive parts along the way. In the end, it contained exactly what it needed.

Milestone

I set milestones for November 13 and 27.

- For the first milestone I predicted a lot of preparation and setup of peripheral items. I met all goals except:
    - LED matrix. This was more of a late design decision than a problem, as I will discuss in the next section.
    - Clap-tempo. I developed tap-tempo and left it at that.
- For the second milestone, I predicted a nearly-complete project.
    - I met these goals, needing just a few tweaks before demo day.
    - The midi turned out harder than I expected.
- I did not fall short, but some things are not as pretty as I would like. It is good to refactor code a few times, rearranging, simplifying and optimizing. Much of the project is first-draft.
- I would not have taken the project further in this quarter. I already put in more hours than required: partly because I am slow, and partly because everything takes longer without good debugging ability. (It might be a good idea to introduce JTag early in the quarter so students have time to incorporate it.)

Completed components

These are functional in the project:

- Dual LCD display with 4-message queue
- Button bus
- Audible metronome
- Measure:Beat and Minute:Second display
- Reference clock, for display and timestamping
- User menu across both screens
- User Input (three modes)
- Offline actions with progress bar
- Event list

Menu Items

Here is the entire menu:

- Song
    - New
    - Save
    - Save As
    - Open
- Play
    - Resume
    - From Top
- Record
    - Resume
    - From Top
- Time
    - Tempo
        - Set
        - Tap
    - Time Signature
        - ¼
        - ⅛
        - 1/16
- Process
    - Event List
    - Quantize
- Option
    - Metronome beep on playback
    - Display Minute:Second or total frames (raw time)

Incomplete components

The components I left out are ones that probably would have been chopped from a production version due to cost or impracticality.

- LED Matrix. I had envisioned a display like the matrix lab, dancing in time with the metronome. It was too much.
    - CPU cost: to do intricate patterns you need to shift 16-bits at a rate of 30/second, which is 30*16 per second. I speeded up the ticks to 200 microseconds to fool my eye, but that wastes CPU time I need to catch incoming midi messages.
    - Real cost: A manufacturer would need to weigh the snazziness of the display against the $8/unit price.
    - What stage of development was it when I stopped? I had diagonal pattern ticking back and forth, like a mechanical metronome.

    - Given more time I would rethink it so it doesn't tax the CPU.

- Clap-tempo: The proposal describes a process of clapping in time to set the tempo.
    - I tested out the math with a tap button.
    - The next step would be to write a tick function tracking peaks on the AD converter. I scribbled out the algorithm but then said enough is enough.
    - Given more time it would take 3-4 hours to implement.

Youtube Links

Make sure they are publicly viewable!

- https://www.youtube.com/watch?v=B882YZxPmp0&feature=youtu.be
- https://www.youtube.com/watch?v=KzETBm-gUOs&feature=youtu.be

Testing

For every big function in the project, I first wrote it in Cloud9's C environment. Then I wrote other functions to simulate various inputs. That's a lot fewer mouse clicks than dumping the program onto the chip every time. Plus you can print anything you want.

After a point, things can't be simulated. Once the function needed to be on the chip, I devised tests to verify proper function as each feature was added. In some cases the best test is just to use it (I usually let my wife break my code).

These are tests that correspond to the list of completed items above.

- Dual LCD display with 4-message queue
    - Call the message function four times and watch the messages display in sequence.
    - The welcome screen demonstrates this pretty well.

- Button bus
    - Display button counter to verify that de-bounce works.
    - Adjust release times for proper double-click speeds
    - Adjust hold times for the proper feel.

- Audible metronome with reference clock
    - Set the tempo to 60 or 120 and clock it with a stopwatch.
    - Do the same thing with the reference clock.

- Interrupt-based USART for midi messages
    - For very early testing, activate an LED when the ISR triggers.

    - Display a received value from USART on LCD screen.
    - Write a function to put known values into the memory, then push play and hope the right sounds come out.
    - Develop the event list display early on. Use it to verify that all notes have been stored.

- Data storage in RAM, implemented on a single array.
    - Arrays are pretty standard. It works in Cloud9 and on the chip.
    - Discover how much RAM is really available by parsing sizes until it crashes into the stack. Oops...

- Data storage in eeprom
    - Save header values, reload and verify correctness.
    - Use the same function to put known values into the memory, then write all to eeprom, reopen and verify.

a

- Dual-screen menu, implemented as a two-dimensional linked list
    - Build the menu with stubs that display the name of the action triggered,
    - Connect functions as they are developed and verify that they are called

- User Input (three modes)
    - Make sure up/down and forward/back work. Verify that save does not insert an extra letter (it uses the same button as forward)
    - Numeric input: verify that up/down and save work.
    - Tap input: verify that the tempo displayed is accurate using a stopwatch.

- Offline actions with progress bar
    - Verify that actions complete without interruption.
    - Give progress bar a minimum speed so it is visible on short operations.
    - Verify that increasing the tempo moves timestamps closer together
    - Verify that the quantize function changes timestamps to match local beat values.

- Event list
    - Verify that event list displays every event.
    - Verify that message decoder displays proper note and time.

Known Bugs

- Quantize: This was buggy in the demo, even though it worked fine in testing.
    - There may be some issue with float multiplication. I saw strange numbers appear during development, which were correct on the next run.

    - Given more time I would implement float multiplication using integers. Float math execution time was not an issue because quantize is an offline process (see above).

- Indeterminate behavior in some cases
    - I think it fails to record on a song loaded rom memory.
    - I dealt with a lot of issues like that in development. My system for activating the midi is a little too complex. The play, echo and record pointers must be set.
    - At one point in development, I went through the flow of control for the one sequence of actions that worked and wrote down every step. Then I compared it to the paths that didn't work to see what was omitted.
    - Given more time I would do that for each possible case.

Resume/Curriculum Vitae (CV) Blurb

Implemented an inter-device communication protocol on an embedded system.

- Designed and built a **midi** (Musical Instrument Digital Interface) recorder
- Established data exchange between an ATMega1284 chip and commercially available music keyboards and sound modules.
- Included features you expect to find on a factory-built product, such as:
    - Record, playback and overdub
    - Audible metronome with adjustable tempo and time signature
    - Time display in Minutes:Seconds and Measure:Beat
    - Song memory to give a song a name and save after power off
    - Tap tempo
    - Event list
    - Interactive menu
- Originated general software usable for any embedded project, such as:
    - Event listeners, input and output queues, display and menu functions.

Future work

If I continued and expanded upon this project I would:

- Convert the primitive MIDI cable to USB
    - Midi devices now implement the midi language over USB
- Miniaturize it.
    - There is really not much going on in a midi sequencer. It should be possible with professional manufacturing techniques to make it as small as a flash drive.
    - This is beyond the scope of what I could do personally, so I would raise money for a startup company and contract a chip company to make it.
- Connect it to a smartphone via bluetooth.

    - You can't trust a phone to do anything in a fixed time frame, so I would let the midi device run on its own clock.
    - Use the phone to control the menu: the device is too small, so let the phone be its user interface.
- Explore non-musical options
    - Midi can control lights and robotics too. Using a known protocol simplifies interfacing between devices.

Extra Credit

https://www.instructables.com/id/Midi-RecordPlayOverdub-With-5-Pin-Connections/

References

All the code is mine, but I relied heavily on sources for wiring and protocol information.

Essentials of the Midi Protocol:
https://ccrma.stanford.edu/~craig/articles/linuxmidi/misc/essenmidi.html

Introduction to Computer Music:
http://www.indiana.edu/~emusic/etext/MIDI/chapter3_MIDI4.shtml

Midi Tutorial:
https://learn.sparkfun.com/tutorials/midi-tutorial/hardware--electronic-implementation

Arduino:
https://www.instructables.com/id/Send-and-Receive-MIDI-with-Arduino/

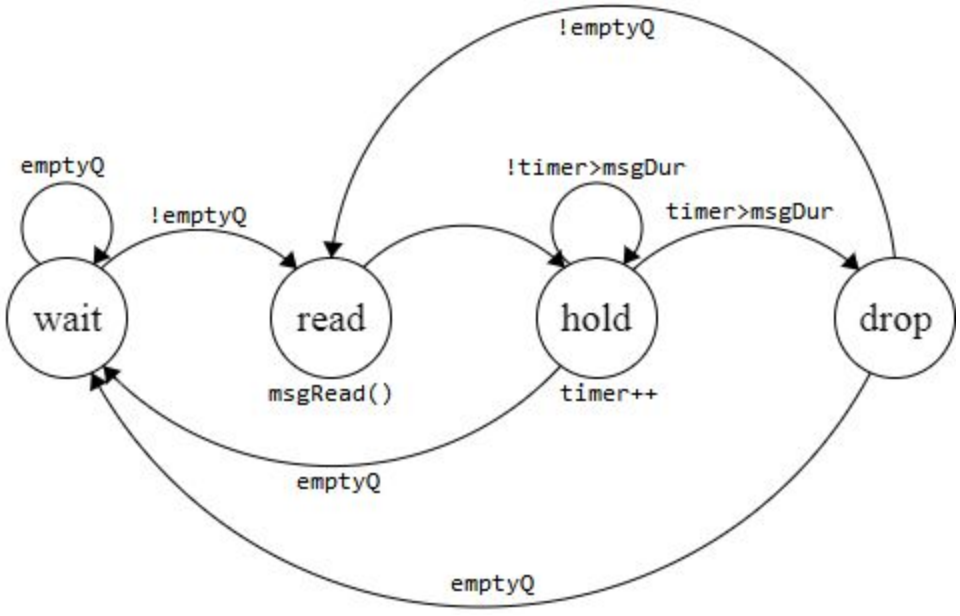
Appendix

Notes on State Machine diagrams:

- Some of these are simplified for clarity: the states are all there but some of the counter names are shortened or consolidated.
    - For example, I change something like msgQ[screen].msgDur to msgDur.
    - Easier to follow what's going on.
- Many State Machines have external controls, meaning they are enabled from a menu command.

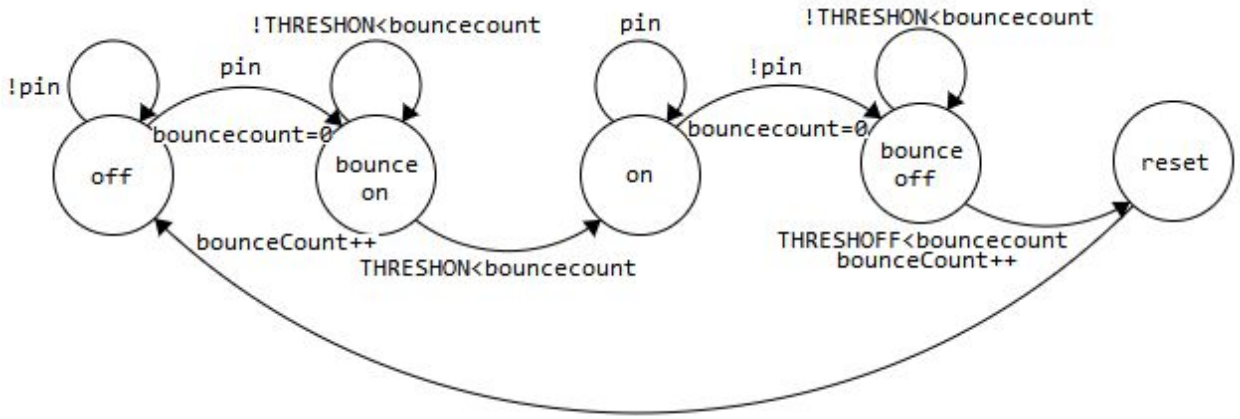
Dual LCD display with 4-message queue

- Two screens share the same tick function
- Each screen has a struct with its own queue
- Array of structs, size 2
- Tick function is shared, with separate states and counters
- Can be externally disabled, in which case the function returns immediately



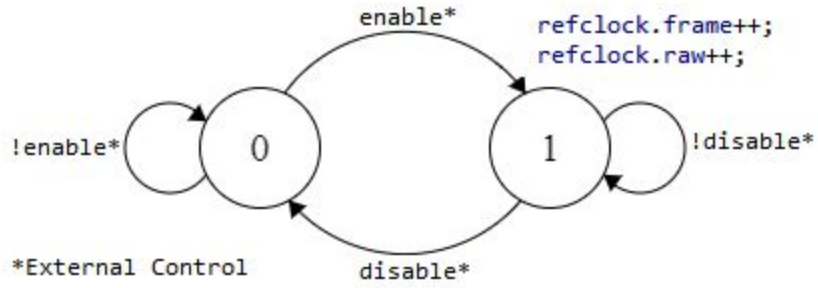
Button bus

- Four buttons share the same tick function
- Each button has a struct on an array of size 4
- Tick function is shared, with separate states and counters



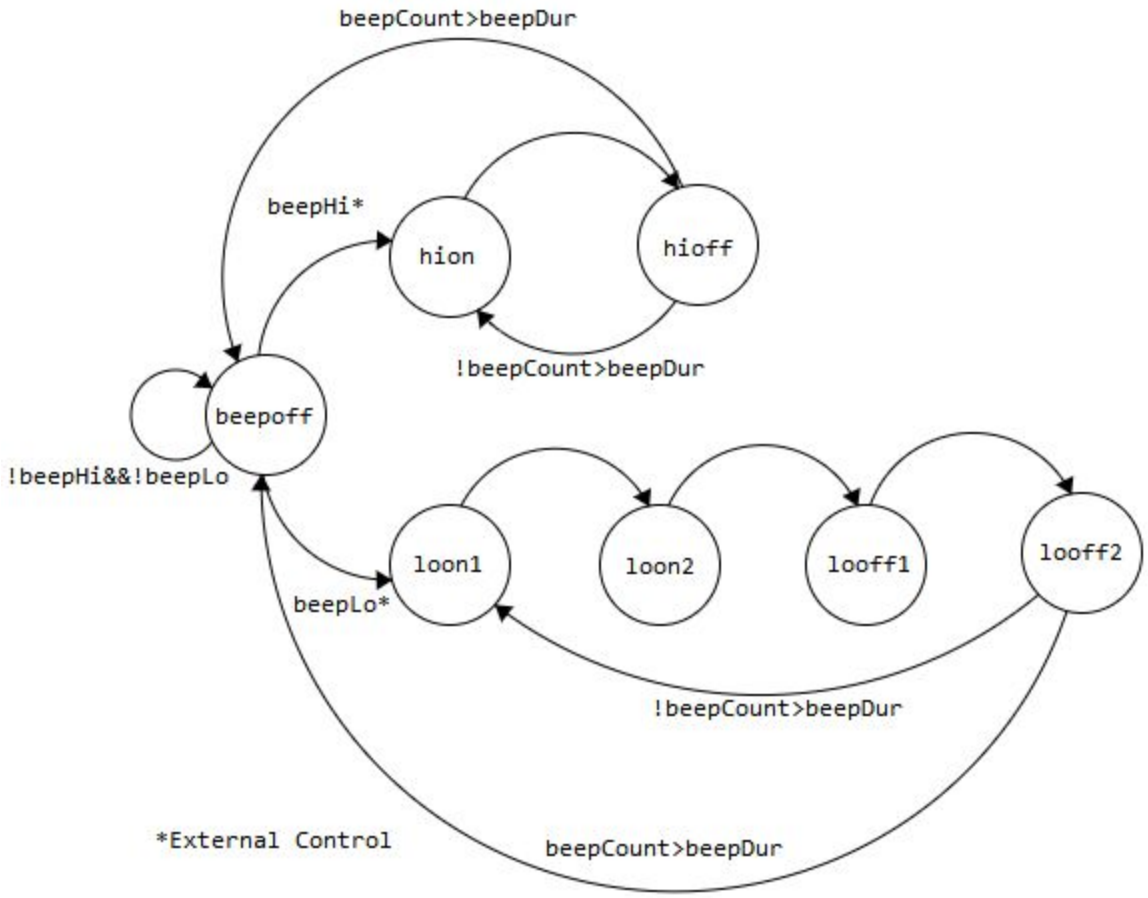
Refclock

- Simple counter
- Increments seconds on frame=100; increments minutes on seconds=60
- Externally enabled by transport
- Externally reset: will pick up exactly where it left off unless zeroed or overridden.



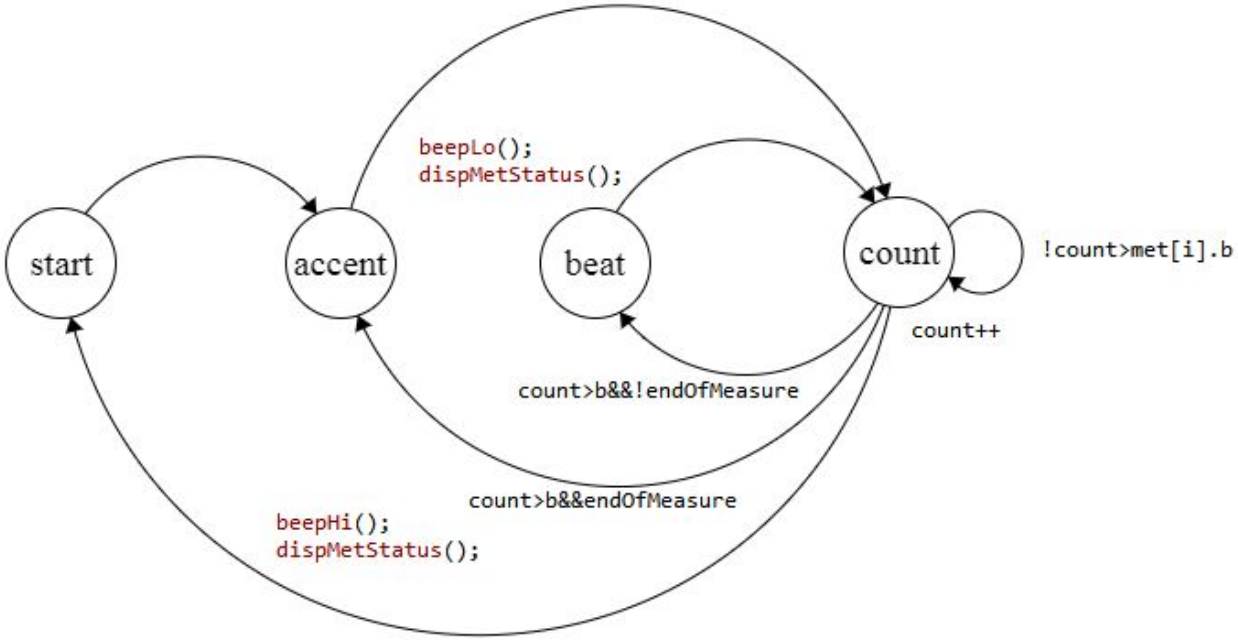
Metronome Beeper

- PWM gave me issues so I made my own sounds
- High beep modulates twice as fast as low beep
- Turned on externally by transport; shuts itself off after timeout
- Transport sets it to high or low beep



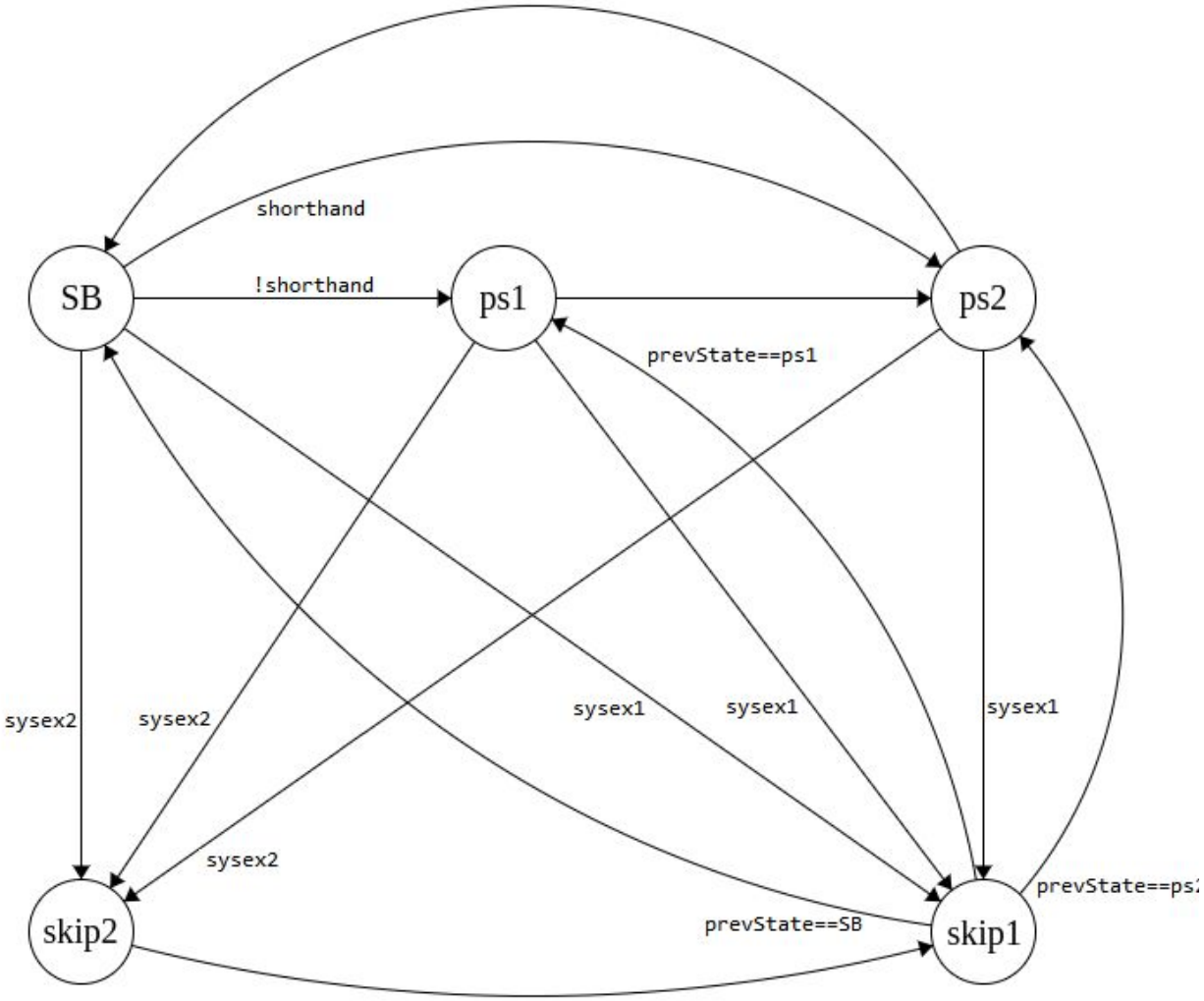
Transport (Metronome)

- Managing enabling of beeper, refclock and midiIn machines
- Calculates beat values from tempo in terms of refclock ticks.
- Advances beats when refclock time exceeds beat value
- Eliminates jitter by adding current error to next cycle
    - Since it polls refclock, it will be a tick behind. Adding the error puts it ahead so the error does not accumulate
    - This is a challenge in implementing variable tempos on state machines of finite resolution
- Externally enabled by menu button
- ‘Start’ state is for externally resetting it to the top of the measure. Otherwise it will pick up exactly where it left off



Interrupt-based USART receive

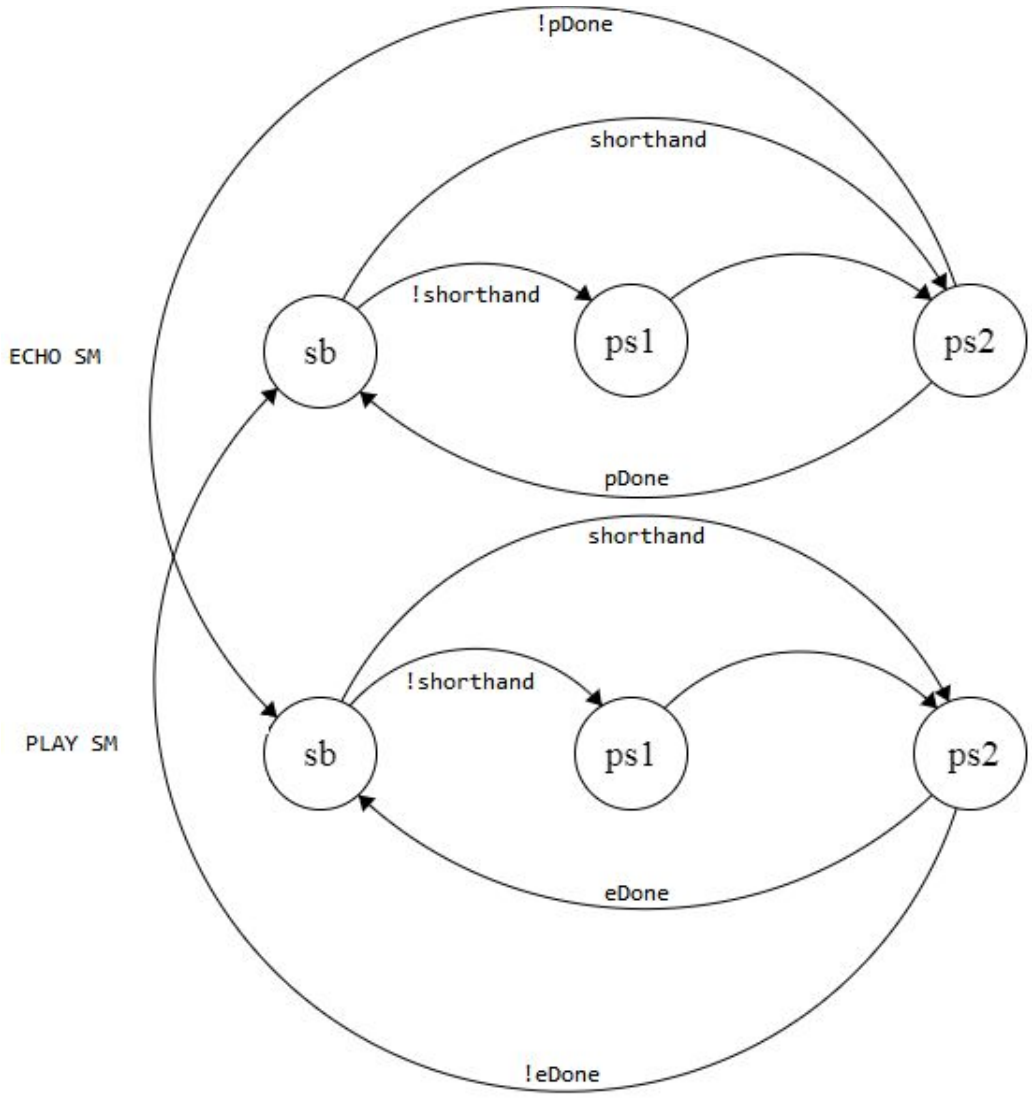
- Three main states, SB, ps1 and ps2, are based on standard three-byte midi message
    - SB is status byte: turns note on and off
    - Ps1 is the note number, 0-127; Ps2 is the note velocity, 0-127
- Protocol may skip some status messages (midi shorthand), in which case do sort of an epsilon transition and end up at ps2 (simplified here as an arrow to ps2)
- On detecting a system-exclusive message, save state wherever it is at and move to skip state.
    - Skipping irrelevant messages is sufficient handling (enough not to mess up the relevant data stream)
    - Skip one or two bytes depending on message type
    - After skip, return to saved state



Interrupt-based USART send

Essentially two state machines, but limited to one ISR call

- Toggles between echo and play
- If both echo and play run out of data, shuts off interrupt
- Refclock turns interrupt back on every tick, to check for more data



Tap Tempo

This is a timer like refclock, but to keep it simple I just gave it its own state machine

- Externally enabled by menu
- Waits until first button press to start calculating