Technical Documentation of the STM32 MIDI Piano

Introduction

The STM32 MIDI piano is a music making device built with the STM32F Discovery board and custom circuitry. It features seven buttons which perform the following actions:

1. Play/turn off C5 (switch 1)
2. Play/turn off E5 (switch 3)
3. Play/turn off G5 (switch 5)
4. Play/turn off B6 (switch 7)
5. Volume down (switch 9)
6. Volume up (switch 11)
7. On/Off switch (switch 13)

It also features a four digit seven-segment display. The first two digits display the note being played (C5, E5, etc. or 00 if no note is being played), while the last two digits display the volume, with 1 being the quietest and 12 being the loudest.

This project was done by Dominick Moreno, a University of Maryland undergraduate student in Professor Hawkins ENEE 440 Spring 2015 class. It was done with significant contributions and help from former/fellow students Jacob Knapo, Tony Abboud and Professor Hawkins himself.

Architecture Overview

All of these features are implemented using three of the Discovery board's numerous timer interrupts as well as an interrupt handler to manage the DMA controller.

TIM2	**TIM3**	**TIM4**	**DMA1_Stream5**
Manages switch presses. Records which switch is pressed or released, as well the timestamp that corresponds with it. Data is enqueued to the global switch queue, which is dequeued in TIM3	Manages switch events for buttons. Maps a switch release (not press, although presses are recorded as well) to an event; e.g. playing a note, changing volume, etc. Also writes values to 7-seg display	Manages volume based on switch releases. TIM2 registers a release, TIM3 recognizes it changed the volume, and TIM4 applies that change	Manages the ping-pong buffer system used by the DMA controller to play notes. Has a C-wrapper that calls directly to assembly code.

Communication between these interrupts is done using global variables. Most of these are just flags, but there are two structures of particular interest. They are:

The switch queue: a circular First In, First Out object-like structure which records switch events into the aptly named switch event structures. A switch event contains the button used, if it was pressed or released, and a counter value representative of the time that the event occurred. The switch queue contains an array of these events, a back and front pointer, and a size field. It is interacted using the functions:

init_switch_container: a constructor

enqueue_sw_event: adds a switch event to the back of the queue, if there is room

getswitch: dequeues the oldest switch event if not empty

is_not_null_event: used to check if a switch event is a null one – i.e. one that follows a specific format to indicate it is a garbage value

vdisplay: a five element array which contains the values to write the 7-seg display. Elements [0, 3] correspond to 7-seg digits [1, 4], with element 4 mapping to the punctuation mark.

Seven-Segment Display

All of the functionality the MIDI piano needs to implement has been wrapped into one convenient function which can be called simply from C. That function is:

void write_sev_seg(unsigned int sev_seg_dig, unsigned int value);

Where **sev_seg_dig** is the digit (on [1, 4]) to write to, and value (on [0, 13]) is the value to be written to that digit. Note that it writes only one digit at a time, but there are four digits to write to. This is done by keeping track of the current digit (in the global variable **dig**) and writing that value once per TIM3 interrupt. TIM3 is clocked at such a rate that all four segments are visible to the human eye at once.

sev_seg_dig is written in assembly, but in practice called only from C. It performs three actions. First it clears the cathode of any values. This is to prevent ghosting of the preceding digit's value onto the current one. The second operation it performs is to clock the appropriate values into the anode to select the digit. Finally it clocks in the 7-seg value provided in the function argument into the cathode. These values are the digits [0, 9], and alphabet characters 'c', 'E', 'b', and 'g'.

Because there are 14 potential values that we may wish to switch-case on, but our instruction set's table branch byte only allows for a maximum of six values, we use three **tbb** instructions. It's also important to note that for some reason this method of switching fills up our immediate literal pool rather quickly. It's for this reason we use the pre-assembler instruction **.ltorg** between each table branch.

The actual method of writing values to the cathode and anode come from code written by former 440 student Tony Abboud, who wrote the macros **ANODE_write** and **CATHODE_write**, as well as all of their helper macros.

Switches

The switch values are read directly from hardware using the pulse-and-sample technique discussed in class. In particular we write a logic-0 to a given switch's GPIO standard output pin, pulse the cathode clock, and read its standard input pin. This functionality is written into an assembly function which takes a switch number (on [1, 13]), and writes/reads to the corresponding switch's input/output pins. It then returns a 0 or 1 if the button was pressed or not pressed, respectively.

In practice the **read_switch** function is only called from the TIM2 interrupt in C. However only a handful of switches are read per interrupt, in order to keep the amount of CPU time used by the interrupt down and to act as a debounce mechanism. It will read switches in the following order: [1, 3], [4, 6], [7, 9], [10, 13], [1, 3], …. The previous and current values of the switch are kept track of in order to determine if a switch was pressed. This looks like:

Previous	Current	Result
0	0	Hold press/on (do nothing)
0	1	Release (record event and do something)
1	0	Press (record event but do nothing)
1	1	Hold release/off (do nothing)

Note that TIM2 makes no distinction between which key is pressed. That is up to TIM3 to parse. Also note that a table branch byte with half-words (as opposed to bytes) is used to switch-case on the switch number. I'm not entirely sure why it worked there and not for the value select in the 7-seg switch-case.

The **SWITCH_read** and all helper macros were written by Tony Abboud.

Audio

Audio is enabled using the DMA controller initialization and ping-pong buffer scheme provided by Professor Hawkins. This code is flashed into memory at the address 0x08010000 using the ST-Link utility. This provides the code to initialize DMA 1, Stream 5 in order to facilitate audio out of the auxiliary jack on the microprocessor. It uses a ping-pong buffer scheme to perform a digital-to-analog conversion of a waveform in two parts. The DMA controller reads from the active ping/pong buffers hard coded into an address space.

The four notes are waveforms I constructed using the included MATLAB file 'Frequencies.m'. They are supposed to be the notes C5 (523.5 Hz), E5 (659.1 Hz), G5 (783.99 Hz), and B6 (932 Hz). These notes make up some chord that my more musically inclined friend assures me sounds pleasant. Unfortunately, because our ping-pong buffer only contains 32 elements we are of course limited to 32 samples. This leads to aliasing issues in our representation of the notes' waveforms which (I think) puts these notes on unintended octaves. It additionally means their sound quality is terrible. Despite this technical shortcoming the tones are distinct from one another.

Which note is being played depends on the status of the switches. Once the piano is turned on pressing one of the four keys will play the corresponding note until that key is pressed again, a new key is pressed to play a different note, or the piano is turned off. Each note's waveform is hardcoded into memory, and if selected is placed into the active ping/pong buffer's location in memory. The DMA controller then sees this waveform the next time its interrupt is called and will send it to the DAC to be played.

The volume at which the note played at is managed by the volume up/down buttons, which control a global variable **volume** bounded in [1, 12]. The waveform in the active ping/pong buffers are shifted to the left (to become louder) or right (to become quieter) by **(12 – volume)**. This method works very well and provides a natural transition between volume levels.

Features Not Implemented and Technical Limitations

There are several items discussed in class or provided in the project specifications that are not implemented in the project. This section discusses them.

The most obviously missing component is the student written DMA initialization sequence. This component was provided by the professor in obfuscated assembly code. I chose not to implement this because the work-reward ratio was simply too high. The professor's interface was more than enough to do the 'fun' parts of the project.

Astute users of the STM32 MIDI piano will notice the rotary encoder has no functionality. This is because when building the custom board attached to the microcontroller I did not find any specifications for what component to use to wire the rotary encoder to the rest of the board. I was too afraid I might short something if I connected a straight wire to the GPIO pins. However, once attached, interfacing the piano's volume with the rotary encoder would be fairly straightforward. Tony Abboud wrote nicely documented code on how to read values from the encoder which would be easy to add. The current implementation of volume would be simple to switch to reading the kind of output provided by Abboud's code.

The bottom button switches (2, 4, 6, 8 and 10) have no functionality attached to them. Unfortunately this project was my first time I had to work with something I soldered myself by hand. Something went terribly wrong with one (all?) of my switches on the bottom row. There appears to be a (semi?) permanent short in one of the bottom switches, which renders my ability to read presses in that row impossible.

A concept discussed in class was context switching, with the main loop or some high priority interrupt acting as a thread dispatcher. While an important concept, I found it to be unnecessary. All of my operations run perfectly fine by using three timer interrupts: values are written to the ping-pong buffers in time, switch events are enqueued and dequeued appropriately, the 7-seg display is written to without apparent ghosting, and volume is controlled in a timely fashion.

Sometimes the 4th digit on the 7-seg display doesn't work light up. As far as I can tell it's not a software error, since it sometimes works and I can see that everything is being written to as it should be in the debugger. I've also chalked this one up to faulty soldering.