Full Scale Piano - Compe213 Project 2

Group #11: Vincent Allen, Dakota Ewigman, Sam Itschner

## Project Description

Rather than only implement the required 3 ‘piano key’ buttons, our group implemented a full-scale piano with 7 octaves (default middle C, 3 octaves up and 3 down). Both the 7 A-G notes and 5 sharp/flat notes were included for each octave, for a total of 84 possible keys. See the 8051 architecture portion of this report for calculations for the notes. Given the increase in scale with respect to the number of keys, it is not surprising that our team hit the memory limit (code and data) within only a couple days of coding.

Vincent Allen’s individual contribution to the project was the backend code for this full-scale piano. Sam Itschner’s individual contribution was the ‘record’ mode for said piano and the ‘play’ mode for the recorded song and for the two stored songs (which he also created with very limited memory). Dakota Ewigman’s individual contribution was the use of serial communication in the last mode to increase and decrease octaves and play keys for specified durations. The common code between everyone in the group was the void main method (default global variable and register values and the infinite loop switching between modes) and the code in the mode.h, leds.h and common\_utils.h files (mostly prototypes). In addition, we all contributed to the code-optimization process. In terms of effort, we all contributed 100%. Sam and Vincent spent the most (about the same) time in the lab, with the majority of their effort placed toward optimization, merging files near the end, and debugging the initial data memory issue. Vincent worked individually initially to write the backbone of the code and in particular Sam put significant effort towards writing two condensed stored songs that would fit in our code given our code-memory restriction. Dakota put in time on his own to improve the serial mode beyond what was required. He implemented sharps, flats and increases and decreases in octaves. In addition, near the end of the project the transmission was not working with Putty, but was using the other serial program. We could have just presented the project using the other program, but he went the extra mile and got it to work in both.

The first mode, designed by Vincent, was what could be best described as full-scale piano. Seven buttons in this mode were dedicated to playing all the musical notes, A through G(including sharps), at different octaves. The starting note was C at the fourth octave. Sharps were played by holding one button down while playing normally. To change octaves, one had to hold down the sharp key and tap one of two buttons simultaneously to go up or down octaves. The range of octave changes was limited with a verticality of three.

A second mode, designed by Sam, was dedicated to recording a song played by the user using the various buttons corresponding to discrete musical keys; an extension of the first mode. Record mode, known as mode 2, stores notes and durations in two different arrays. The notes are stored directly from the TL0 and the durations are counted in the number of half periods. The recording happens directly after the user changes keys, changes modes, or hits the note limit. Recording is accomplished by setting the values in the arrays at index Note\_Count at the end of the button press. Utilizing the 8051 architecture, the memory allocated for these arrays had to be smaller than we would have liked due to memory limits which was only five notes.

The third mode, designed by Sam, was utilized in order to play back the previously mentioned user-made song and also to play two hard-coded songs from program memory. To play a song, one of the three middle row buttons were pressed to initiate a playback. If the user desired, they were able to stop the song in the middle of play. This was accomplished by checking if the mode-change button was pressed after every note played. When we first started our project, the first song that we made was complex and required a large amount of program memory to operate. Several *for* loops were utilized as well as two large arrays to store the notes and durations. While the song itself sounded great, the memory footprint was just too costly to maintain(about a fifth of the total memory allowed) and thus resulted in the final two songs that are mostly accomplished through code iteration. Both final songs combined took up less memory than our original song.

A fourth mode, designed by Dakota, was created to play from serial input. When a note wasn’t already being played, the program would read in serial input, then use this to determine which note was meant to be played(C,D,E,F,G,A,B,C#,D#,F#,G#,A#), what octave the note was meant to be played in, or how long the note was meant to be played. An ‘A’ denoted the A note, ‘B’ denoted a B note etc. A# was represented by an ‘A’ followed by a ’B’, C# denoted by a ‘C’ followed by a ‘D’ etc. An increase of octave was represented by a ‘^’, while a decrease of octave was represented by a ‘v’. multiple of these keys could be used for one note to increase or decrease the octave multiple times. The duration was input as anywhere between 1 and 3 digits, the expected value for the duration is anywhere between 1 and 127, where the input number is the number of seconds the note would play. If a number, d, between 128 and 255 were to be input, then the duration of the note would be (1/(256-d))s. Once a ‘;’ was received, the program would play its note for the set duration.

We wired a 7-segment display to the simon2 board with 1 NOT chip, 2 AND chips, and 3 OR chips. Each key was output on the display as it played (for all modes) and the decimal point on the display corresponded to a sharp. For example, a C# (aka D*b*) would display as C with the decimal lit. Note that AND gate combinations were shared across circuits for multiple pins.

We first encountered the data memory limit when trying to store the number of machine cycles per half period for all the middle C notes (removed at first then later added to save code memory) along side a float array for the octaves (later reduced to a single float) and a plethora of global variables and constant variables (commented out at the very end of the project). It took several hours of debugging to figure out the issue initially, and once we discovered the (data memory) issue lied with the number of global variables included in the .h files for the project, we immediately removed as many unnecessary global variables as possible and migrated some of them to switch cases in code-memory (e.g. the ‘get\_key\_state’ method). Despite a continuous stream of optimization performed by all of us in addition to the development of new features, we very nearly ran out of code-memory before completing all the minimum requirements for the project. Some of the weight was distributed back to the data memory after Sam created a condensed version of the reg9332.h file to free up some space. Such was our lack of code-memory space at the end of the project that the difference between a successful compilation and not was the replacement of a ‘~’ and ‘+1’ for a 2’s complement with a ‘-’.

In our initial design, we attempted to play stored notes (from the ‘record’ mode) by storing the note, whether we were in sharp/flat mode at the time it was played, and the octave at the time. If for instance we wanted to store 10 notes, the octave storage alone would have been a significant weight on the data-memory (40 bytes!), so we attempted to store them as characters. The conversion coupled with tracking the sharp/flat mode at the time of the recording proved to be too much of a weight on code-memory, so in the end we instead stored the TH0 and TH1 values at the time of the note to fix the issue.

An additional issue we encountered was with the 7-segment display. Only leads a, b and g were working when it was first tested. Using an LED to test for expected output while certain keys were pressed, the issue was traced to the first OR chip on the board. After swapping it with the other OR on the board, the problem was immediately resolved (no rewiring necessary). This leads me to believe the issue was not with the gate, and that the first OR gate originally causing the issue was probably just not implanted completely in the breadboard.

## Use of 8051 architecture in project design

There is only one aspect of our code lacking - during the record mode notes played for less than 1 second are played during the stored mode for a much shorter duration. After reviewing the key-interrupt method repeatedly, I assume this error is due to difference in the amount of time taken (different logic paths) in the interrupt to reach the sound port toggle. This would have to be resolved by observing the disassembly view of the code and adding appropriate delays to one of the logic paths.

We used serial communication in our project to allow outside output/input. We used a modified version of the UART library to implement our serial communication. This communication was utilized to send the title of the stored songs being played back to the user, as well as to play songs via data being received serially while in serial play mode. We chose not to use serial interrupts, as we didn’t need or want anything else happening as we were transmitting or receiving data.

The timers and timer interrupts were used to allow for our playing of audio. We would set the timer for the half period of whatever frequency we wanted to play; then in the timer interrupt, we would reset the timer to the half period again and toggle the sound port. This would continue until we turn off the timer.

Regarding the frequency calculations, etc. concerning the timer see the attached spreadsheet (frequency values taken from Wikipedia) and the excerpt from the individual report below.

----------------------------(from individual portion)-------------------------------

-----------------------------------------------------------------------------------------