Tyler Ekaireb & Geffen Cooper
Professor Brewer

ECE 153A
December 8, 2020

Lab 3B: The Chromatic Tuner

Purpose & Goals

The goal of this final lab was to employ all of the knowledge we have gained this quarter (embedded systems design techniques, interrupts, QP-Nano, signal analysis, etc.) to develop a complete and polished system (the chromatic tuner).

Design Methodology

Our main design goal was to create a chromatic tuner with a simple and sleek interface that was accurate and robust. This favored a design philosophy oriented around incremental change and quality over quantity of features.

At the start of this lab we had several segments of working code from prior labs including peripheral initialization code, interrupt initialization and handler functions, button debouncing, helper functions we created for the LCD, a QP-Nano based HFSM (Hierarchical Finite State Machine), and an FFT (Fast Fourier Transform) algorithm. Initially we brainstormed a simple structure for a GUI that would link these components together. The details are discussed in the user interface section of our report. Next, we incrementally integrated these working components to create a rough, working prototype for our tuner that we gradually polished, optimized, and debugged over the past two weeks.

An important aspect of our design methodology was trying to make the code modular so that we could easily add and improve features without interfering with the system as a whole. This made debugging easier and the details are discussed in the code integration portion of the report.

In the remainder of the report we discuss the different aspects of the project in detail, including user interface, code integration, and testing methodology, and the design decisions we made for each part.

Improving the Accuracy of Frequency Measurements

One big goal of this lab was improving the accuracy of our FFT. Initially we had a fast implementation of the FFT algorithm from lab 3A with a runtime of ~10 ms and wide frequency range with a Nyquist Frequency of ~20KHz. However, this implementation was very limited (it only had a single mode) and had very poor resolution for lower octaves. To improve the quality of our FFT algorithm we made it octave dependent and tuned certain parameters (decimation amount, effective sampling frequency, bin width) for each octave to get the best performance. The following table shows the results of calculations we made when tuning the parameters for each octave.

$$\#\,bins = \frac{Total\ Samples}{Decimate} \qquad Effective\ F_s = \frac{48828.125}{Decimate} \qquad Bin\ Width = \frac{Nyquist\ Freq}{\#\,bins}$$

$$Min\ Note\ Spacing = C\#_n - C_n \quad n = octave$$

Octave	Decimate	Total Samples	# bins	Effective F_s (Hz)	Nyquist Freq (Hz)	Bin Width(Hz)	Min Note Spacing (Hz)
0	64	4096	64	762.939	381.470	11.921	0.972
1	64	4096	64	762.939	381.470	11.921	1.945
2	64	4096	64	762.939	381.470	11.921	3.889
3	64	4096	64	762.939	381.470	11.921	7.779
4	32	4096	128	1525.879	762.939	11.921	15.557
5	16	4096	256	3051.758	1525.879	11.921	31.114
6	8	4096	512	6103.516	3051.758	11.921	62.228
7	4	2048	512	12207.031	6103.516	23.842	124.457
8	2	1024	512	24414.063	12207.031	47.684	248.913
9	1	512	512	48828.125	24414.063	95.367	497.826

Table 1: Octave Parameters

This table shows 7 parameters we took into consideration for each octave. The goal of the table was to get the highest resolution for a given octave while also minding the Nyquist frequency so we could detect the highest frequency in each octave without aliasing. We did this by maximizing the decimation while also maintaining a wide enough frequency range.

We found that 11.921 Hz was the smallest bin width we could get with the max sample size and sampling frequency. These values were hard coded into our implementation for each octave. We experimented with different combinations of parameters as well which is discussed in the testing section of this report. During runtime, when an octave is selected, our code essentially grabs the according row of this table and then begins running the FFT.

One of our key design decisions regarding the FFT portion of the code was to avoid recomputation by hard coding parameter values and the reordering algorithm for each octave. While this sacrifices code flexibility and readability, it drastically improved performance and accuracy which was paramount to our design philosophy.

Our final implementation uses slightly different parameters than those shown in the table that we found to be the most accurate through testing. The exact values can be found in the source file `octave_selector.c`.

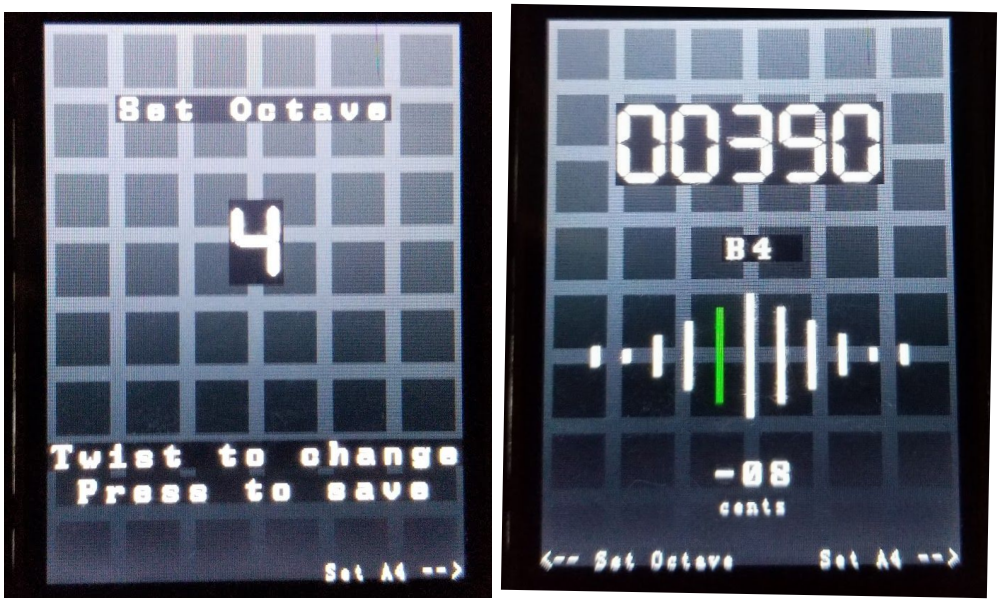
The User Interface (Control Flow and Structure)

In order to make our code more easily testable and modular, we devised a page system that is uniform and decoupled from the content of the pages themselves. A function (init_page) accepts a parameter which specifies the ID of the target page. Code running on the current page is stopped, the new page is initialized and drawn to the screen, and its necessary code is

executed with a single function call. This enabled us to focus on each page's content separately as loosely coupled systems, mostly communicating with only a few shared variables.

The UI (User Interface) consists of two menu pages and a tuner page. The two menus are highly similar for design consistency and code reusability. Both use the rotary encoder to increment and decrement values, while clicking the rotary encoder returns the user to the main screen (tuner page). We wanted the menus to be easily readable and understood without prior knowledge of the system. A title bar indicates the value that is selected (either the octave or A4 frequency), and concise instructions at the bottom indicate how to use the menu. The UI will not allow the user to exceed minimum and maximum bounds of the value (e.g., 0-9 for the octave). Pressing the push buttons will take the user to the other menu, and all values are immediately saved upon mutation (so changes are preserved when switching pages). Pressing the rotary encoder button will exit the menus and return to the tuner page.

The tuner page displays the detected frequency, note, and error (in cents). There is also an error bar which visualizes this error in 10-cent increments with white vertical bars of varying heights. The relevant bar will change color to reflect the proximity to the note (bars on the periphery become red, closer bars become yellow, and the middle bars become green). The RGB LED on the FPGA also changes color to reflect the accuracy of the note. These are discussed in more detail in the Extra Features section.



The images above show some of the interface that we constructed. We chose a dark background with light text for high contrast and readability. Each page contains brief instructions near the bottom of the screen to indicate the controls. "Twist to change, press to save" reminds the user how to mutate and save their selection using the rotary encoder. The left and right push buttons can be used to view the Octave and A4 menus respectively, as noted at the bottom of the screen. The frequency and note are shown in the right image. The error is conveyed numerically, spatially, and chromatically with the colored bars and "cents" readout below them.

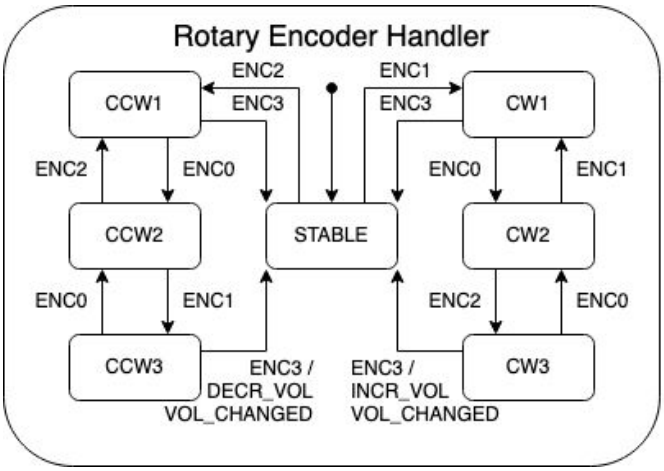
Tyler Ekaireb & Geffen Cooper
Professor Brewer

The Code Integration Process

One of the main challenges of this lab was integrating working code from prior labs into a single project. This was difficult because each piece of code from prior labs had a different control flow with their own main() that needed to run together. For example, the encoder state machine runs constantly as well as our GUI and other peripherals in parallel with the FFT algorithm.

As a result we tried to make our design modular by creating many independent helper functions for each component that we could easily call from higher level functions. For example, the initialization for SPI, the button and encoder interrupts, and the other peripherals are all independent functions and to draw to the LCD we created helper functions to display text and draw the background. Above these functions we had pages, as discussed in the user interface section, that can easily call these helper functions to implement the desired behavior for a certain state in our system.

For example, all the FFT and octave parameter associated code is independent from the rest of the system so that it can just be called when needed on the main tuner page.



The diagram above shows how our rotary encoder is handled by a state machine implemented in QP-Nano. There are three intermediate states for each direction of rotation and one stable state. Handlers are only called when transitioning from the last state in a rotation to the stable state.

Testing Methodology

We created our tests to match our design philosophy which emphasizes accuracy and robustness. To test accuracy we exhaustively tested each octave and adjusted the octave parameters through a heuristic process in which we tried to get the best result for octaves 3-9.

Our testing procedure used the online szynalski tone-generator. We found varying results based on how the audio is being produced (computer, phone, earbud) and found that placing a phone close to the microphone worked well. For each octave we started at the A note and gradually moved higher or lower making sure all the notes are correctly identified. If a note was incorrectly identified, then we adjusted our octave parameters (for the final outcome almost all notes in octave 3 and above were correctly identified). Next, we made sure that most notes were

accurate within +/- 10 cents and tried to stabilize certain notes that jump between frequencies through averaging.

For testing, we prioritized cents over the frequency value displayed because certain frequencies were off by 1 or 2 Hz due to rounding. The cents calculation used the floating point value returned from the FFT and is therefore a better source for measuring uncertainty. We did have trouble with some notes that kept bouncing between frequencies but these were within +/- 15 cents which we think is reasonable.

To test robustness, we tried to “break” our GUI by using it incorrectly. For example, the encoder is supposed to do nothing while we are measuring frequencies so we pressed it and twisted it a lot while the FFT was running. This produced some vibrations which did interfere slightly if the audio source was not close to the microphone. We also pressed the buttons (including the ones we don’t use) many times in quick succession to see if we could get the system stuck in some weird state. We attempted to exceed the bounds of input values, such as the octave number, by rapidly spinning the rotary encoder after reaching maximum and minimum values.

Finally, we let family members try to use the system with only a brief explanation of how to use it. This not only tested robustness but also ease of use and fluidity of the system from an outsider's perspective.

Extra Features

The error bar consists of very thin rectangles of various heights. When a frequency is detected and the error has been calculated, the corresponding bar (spaced at 10-cent intervals) changes color. Both its proximity to the center and color (red, yellow, or green) reflect the degree of error. Because the main background is dark, both white and colored bars are visible without any sort of special background. The error bar was designed to have very few pixels so that it can be updated quickly.

The RGB LED relies on code that was heavily influenced by prior labs. We use the same function to write to it, but in this case, we write green for accurate frequencies and red for inaccurate frequencies. Before exiting the tuner page, we turn off the RGB LED, as it is no longer relevant when the FFT is disabled.

Challenges and Future Improvements

The constant updates to the display were consuming CPU resources and affecting the accuracy of the error calculation. In order to reduce the impact of writing to the display, we decoupled this aspect of the UI from the FFT and error calculation. The FFT and error calculation run constantly, while the UI on the page is updated periodically at set intervals. This resolved our accuracy issue by freeing up resources for the algorithm, and by continuing to update the display many times per second, the change is not noticeable to the end user.

We had some difficulty with concurrency, as this is not natively supported by QP-Nano. For example, the rotary encoder is always in some state, but this should not be tied to the state of a particular page. The page system and the rotary encoder system should run concurrently. To solve this, we used QP-Nano to manage the rotary encoder as a single unit. Changes on the rotary encoder call a handler function to update values. In order to determine what is updated,

this handler checks other state values, e.g., the current page, to determine what value to change. This way, QP-Nano always keeps track of the rotary encoder, but it is not tied to the page state or UI state, as these systems are only loosely connected. This obviates the need for duplicate code (e.g., a state for each combination of states of the rotary encoder, page system, UI, etc.).

One future improvement is automatic octave detection. Currently, the user must select an octave before playing a sound. Ideally, they could simply turn on the device and play an instrument, and the device would automatically register the current octave. This could be done with a fast but coarse FFT to determine the octave followed by a fine FFT that is more precise. We could also begin with an FFT with a high sampling rate and step down until the frequency is within the FFT's optimal range.

As concurrency was a serious issue for us, a future improvement might be to rebuild the application within a framework that supports concurrency, so that everything is consistent. In its current iteration, the ad hoc concurrent systems were designed to work for this application, but are not easily extended. Having a single framework that supports this sort of orthogonality would be ideal.

If we were to release such a device as a standalone product, we would certainly package it differently. While the FPGA is a wonderful development environment, many of its ports go unused in this project and its size would make it a cumbersome tuner. We could instead load the code onto a small processor or ASIC and 3D print a housing with a slimmer form factor. We might also offer presets in the A4 menu for specific instruments.