EE3102

9 February 2012

University of Minnesota

Preliminary Design Report

Project Members:
Patrick Barrett
Mark Pennebecker
Kayiita Johnson
Prashant Dhakal

Submitted to:
Professor Higman

Abstract

The general purpose of this project is the design and construction of a music tuner. Specifically, this tuner will be checking whether or not an input pitch matches 440Hz (A). The input will be read from either a microphone or a coax cable fed from an oscilloscope. The purpose of a tuner is not only to test on whether or not a pitch matches a note, but also to see if the input pitch is sharp or flat (whether the pitch is above or below the given pitch). Our tuner will indicate this information through a series of LEDs. We plan to use a microcontroller to analyze the input and control the output LEDs.

1 Introduction

The methodology for finding pitch in out tuner is largely based upon the theory behind musical tones and how those pitches are measured.

The pitch we were measuring is called A440, which is a standard tuning pitch for musical instruments. The 440 denotes the frequency in Hz of the pitch, which is called A by convention. An octave is defined as a ratio of 2:1 between 2 pitch frequencies. In this case, the pitch A which is an octave above has a frequency of 880 Hz while the pitch A below the standard A has a frequency of 220Hz. Each octave is split into 12 'semitones', which are considered distinct pitches within the octave. Since we are trying to determine if the input is in tune (matching the 440Hz frequency) we require further precision past the semitone measurement. Thus, we need to measure the accuracy of the input pitch using the unit known as the cent. A cent is a further division of the interval between 2 semitones. The interval between 2 semitones is divided into 100 cents. This means that each octave is divided into a total of 1200 cents and that the cent represents a different range of frequency depending on the octave.

We were tasked with measuring the difference between the input pitch and A440 in terms of 5 cent intervals. This specification of the interval of measurement was given by the project requirements, but it does make sense given that there is little audible difference between adjacent cent frequencies.

2 Overview

The basic flow of the signal is shown in Figure 1. After the input is received from the oscilloscope (microphone input will need a few more steps), the signal will pass through a bandpass filter to help remove any noise from the signal. After being filtered, the signal will pass through a preamp, to boost the signal so that it can be further processed by the later stages. The signal is then sent to a square wave converter, which will take the analog signal

and convert it into a series of square pulses to be handled by the microcontroller. If necessary there may also be an output buffer stage. The microcontroller will take the signal generated from this stage and measure the frequency of the square pulses over a certain period of time. Using this information, the microcontroller will choose light up an LED to display how close the signal is to the desired pitch.

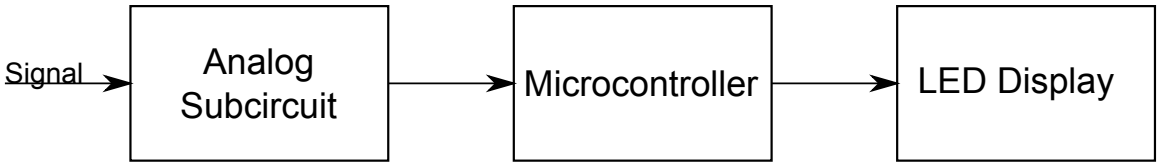

Figure 1: Block Overview

3 Analog Subcircuit

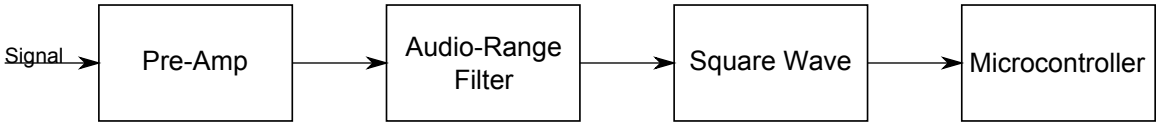

Figure 2: Block Diagram of Analog SubCircuit

4 Timing

In order to calculate the frequency of a specific cent frequency, the following equation was used:

$$b = a * 2^{\frac{n}{1200}} \tag{1}$$

where b is the desired frequency, a is a lower frequency than b, and n is the number of cents in between a and b. For the positive cent range, we used a = 440Hz and the above formula where n was the number of cents above 440Hz. For the negative cent range, a = 220Hz and n was calculated from

$$n = 1200 - |\text{cent offset}| \tag{2}$$

so if we wanted to find the frequency of -5 cents from A440, n would be 1195. The frequencies found from this formula can be seen in Table X.X. Knowing the frequencies corresponding to the cent ranges is useful, but it must be manipulated somewhat for the microcontroller to interpret. The CCP module on the microcontroller measures events by using an internal timer which increments on the instruction clock (Fosc / 4). Therefore, the frequencies calculated in the previous stage need to be converted into a number of instruction cycles. The following equation describes this calculation:

$$(\text{cent frequency})^{-1} * \left(\frac{\text{Instruction Cycles}}{\text{Second}}\right) \tag{3}$$

The converted values calculated from this equation can also be found in table X.X. The Fosc, which is the internal clock frequency, is 16MHz. This means that there are 4 *106 instructions per second. We could increase the resolution of our measurements by increasing the clock speed of the microcontroller, but we felt that keeping our design low powered was more important than the increase in resolution gained by using higher voltage on the microcontroller.

The lower and upper bounds were calculated by averaging adjacent cycle counts. It is important to note that the cycle counts are whole numbers since it is not possible to have a fractional cycle. All cycle values were rounded to the nearest integer value, which means that the ranges intervals may be off by at most 1-2 cycles, which we deemed as an acceptable imprecision.

Our tuner will take an input signal to be tested. When the input pitch matches A-440Hz, the center LED will light up. As the pitch moves out of tune, other LEDs will light up to signify how sharp or flat the pitch is compared to A. These LED's will be set to 5 cent intervals ranging from -30 to 30 cents around A. Anything above or below these ranges

Cents from A440	Frequency (Hz)	Cycles @ Fosc = 16MHz	Lower Cycle Bound	Upper Cycle Bound
-35	431.1939264	9276	9263	TMR
-30	432.4410634	9250	9236	9263
-25	433.6918074	9223	9210	9236
-20	434.946169	9197	9184	9210
-15	436.2041585	9170	9157	9184
-10	437.4657865	9144	9130	9157
-5	438.7310635	9117	9104	9130
0	440	9091	9078	9104
5	441.2726067	9065	9052	9078
10	442.588941	9038	9025	9052
15	443.8288729	9012	8999	9025
20	445.1125537	8986	8974	8999
25	446.399944	8961	8948	8974
30	447.6910645	8935	8922	8948
35	448.985916	8909	1	8922

Table 1: Cent Step Calculations

will simply light up the leftmost or rightmost LED depending on whether they are above or below the required pitch. The decision of which LED to light up will be handled by a microcontroller.

5 Microcontroller

We decided to use a microcontroller for the signal processing element of our tuner. Microcontrollers allow quite a bit of flexibility in the processing of the audio signal. We chose the PIC18f4550 partially because this microcontroller had enough ports to drive our LED array. However, the primary reason that this microcontroller was chosen was due to our prior familiarity with the architecture and functionality of the device. Given the limited requirements of the tuner device, there were no special requirements of the microcontroller, which gave us a great degree of flexibility with our choice.

The code for the microcontroller is reliant on three main functionalities of the PIC18f4550: the TMR1 module, the CCP module, and the Primary Idle Mode.

The first module we used is called the TMR1 module, which is a simple timer module. This timer was configured to increment on each instruction cycle to ensure the highest resolution possible. This timer was chosen because it had the ability to hold 16 bit numbers, which was necessary for the number of cycles we needed to count. This timer also works in conjunction with the CCP module, which will be discussed in greater detail later. We used the timer to count the number of cycles for each period of the input signal, but it also had another important function. If the timer were to roll over (meaning that it attempted to increment above 216), then the program will vector to an interrupt and reset the averaging function used in the CCP interrupt. If this event occurred, then that would mean that the input signal was no longer transmitting, meaning that it would be necessary to clear out the stored averaging values in order to maintain a clear reading the next time an input was detected on the port.

The CCP (Capture, Control , Pulse Width Modulation) Module has a wide variety of uses. For this application, only the capture mode will be used. While configured in the capture mode, the microcontroller waits for a trigger on an event occurring on a specific port on the . This event, which the user can define as a falling or rising edge transition, will generate an interrupt and vector the CPU of the microcontroller away from the main function. We chose to configure the capture module to copy the contents of the TMR1 counter register into the CCP1REG (a register for use by the CCP module). This transfer is done automatically by the microcontroller. Basically this interrupt is generated for every cycle of the input signal, meaning that the timer module will record the number of instruction cycles for each period of the signal. The cycle counts corresponding to the cent ranges are shown in Table X.X. The cycle count is processed within the interrupt and averaged to ensure that the measurements accurately represent the frequency of the input signal.

The final module that our code utilizes is called the Primary Idle Mode. Since the majority of time our code is spent waiting on interrupts to be generated, much of the microcontroller can be disabled during the idle periods. The primary idle is a configuration of

the sleep mode available for use on the PIC18f4550. This configuration disables the CPU of the microcontroller, meaning that no more instructions will be processed. Although the CPU is disabled, the peripheral devices, such as the timer modules, will continue to be clocked from the primary clock source. There is a bit of a delay in restarting the microcontroller from the idle mode, but since the TMR1 module is copied over to the CCP1REG in hardware, this does not affect the accuracy of our measurements. Also, since our

instruction clock was on the order of about 3dB faster than an octave above the desired pitch frequency, we assumed that the wake up time was fairly negligible. We made the same general assumption about the interrupt service routine which calculates the average value.