AutoTabber: A Frustration-Free Guitar Tabbing System

Taryk Alsagoff, CSE, Michael Murphy, EE, Michaela Shtilman-Minkin, CSE, and Matthew Wojick, EE

Abstract—**We introduce AutoTabber, a product designed to aid musicians, particularly electric guitarists, in transcribing what they play into the readable musical format known as "tablature." The system is designed to be a modular attachment to already-existing guitars without permanently damaging or modifying them. AutoTabber utilizes the hexaphonic split-pickup, most commonly seen in guitar synthesizers to detect notes played on each string separately allowing for a near-exact representation of notes played.**

I. INTRODUCTION

DUE to the design of the guitar, there is a significant degree of ambiguity in the interpretation of standard music notation when playing the instrument. That is, there is no 1:1 mapping from standard music notation (also known as staff notation) to what is actually played on the guitar. Because of this, it is extremely difficult for amateur guitarists to learn to read standard music notation [1]. Instead, the idea of the tablature ("tab" for short) was introduced to alleviate this difficulty [2]. A tab, rather than just a note, represents a physical location along the neck of the instrument in a 1:1 fashion, thus leaving the guitarist with no questions as to where to play a given note.

However, the process of creating a tab is far from trivial. When a user is playing a new song by ear, they are not typically conscious of what fret numbers are utilized. So they must play a small section of the song, memorize the fret numbers, type it into tablature, and then repeat. This is a very tiring process, and it can discourage players from creating and sharing their songs with others.

Originally, tabs were only written by hand or using a text editor. Now, various software applications exist for the writing and editing of tablature, such as Guitar Pro [3]. Additional features such as MIDI-based playback, practice modes, and metronomes have been implemented in these software applications. However, these only improve the quality of the tabs; creating a tab from this type of software is not much easier than before.

Numerous attempts have also been made at purely software-based conversions to tablature via machine learning, pathing, and optimization algorithms with varying success [4][5]. Others have attempted to read the output of the guitar's native pickups and process that signal [6]. However, these systems ultimately end up guessing the optimal way of playing a sequence rather than preserving what was actually played. By using a hexaphonic pickup, as well as additional hardware, AutoTabber is different than previous purely software-based implementations in that it can determine exactly which string the notes are coming from. This would allow no ambiguity as to where the notes are being played. Such a solution would make the process of tablature generation a much more efficient and enjoyable experience to those who want to share their music to the world.

II. DESIGN

A. Overview

Our goal is to create a compact module that can be unobtrusively installed on a guitar without permanent guitar modification. This module will do analog to digital conversion and some basic signal analysis before sending that data to the software to be interpreted and converted to tablature. We would like to be able to output tablature that is >80% accurate when playing basic songs, such as Mary Had a Little Lamb.

The hardware will consist of a guitar pickup for each string, a basic preamp on each of the pickup coils, 3 ADCs (each handling two of the strings), and a microcontroller (see Figure 1). It will initially be powered by, and communicate over USB. Future features such as battery power and wireless are a possibility. Doing some processing in hardware on the guitar will reduce bandwidth requirements between the module and the PC, simplifying the communication. The ADCs we are considering have programmable signal conditioning features built in; most importantly, an adjustable amplifier with automatic gain control. Having these features inside the ADC greatly simplifies our hardware design, lowers cost, and decreases required PCB size. The most important operation of the microcontroller will be to perform Fourier transforms on the 6 input signals and send relevant information to the PC so that notes can be identified. After considering a few different microcontrollers with different advantages and disadvantages, we chose to use the high-performance, but low-power, TMS320C5535 DSP from Texas Instruments (TI). The hardware FFT implementation on the C5535 is 3.8 times faster than performing the same computation on the CPU and 6 times more power efficient. It can perform a 1024 point FFT in hardware in under 100 nanoseconds.

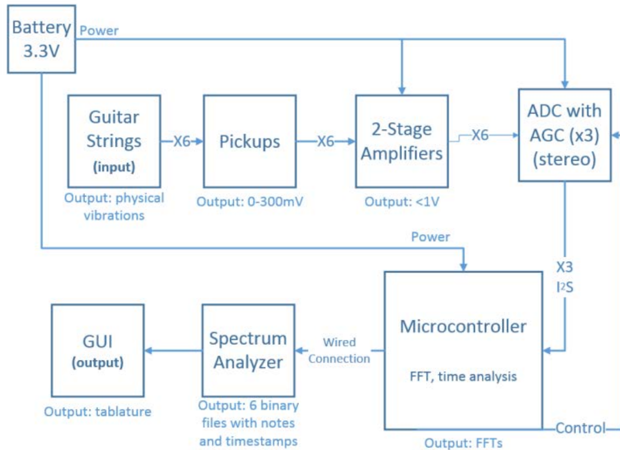


Figure 1: Block Diagram

Block I: Pickup

The pickups are part of what makes AutoTabber substantially different from other systems. Instead of looking at the output of the guitar's native pickups, AutoTabber utilizes a magnetic hexaphonic pickup as a modular insert for the end-user's guitar. This will allow six separate signals to be processed while preserving the original string information. This is not possible with the guitar's native pickups as they typically use only one coil which generates a single signal.

The pickup will serve as the entry point for our system. Mechanical vibrations from the end-users guitar will be translated into a small electrical signal by the pickup which will be passed on to an amplifier.

Our prototype pickups are built using a Solidworks CAD model with a high-end 3D printer. Each coil is a separately printed piece wound with 44 gauge enamel coated wire. Currently, the wire is wrapped around each piece manually using a power drill.

The specifications for each pickup coil are as follows:

Single Pickup Specifications

Height	<10mm
Diameter	<10mm
Output Voltage	>10mV
Variation b/w coils (ohms)	< ±10%

The pickup dimensions come from the spacing between the strings on the guitar as well as the available spacing under the strings. In addition, we want the six coils on the guitar to be consistent, such that they are each produce the same output voltage, and thus hold similar resistive values (< ±10%). This means that the pickups need to be wound with a similar amount of turns each time. This is very difficult to achieve with a power drill, so we are designing an automatic pickup winder machine that will automate the process, while achieve greater consistency.

The prototype pickups have successfully generated a signal around 300mV on average (averaged between six separate pickups). However, because there is a large variance in the input signals (the volume the guitar is played at may not be constant), characterizing the pickups has proven difficult with just oscilloscopes, since we do not have a way to consistently pluck the string each time. A method that can be used to reduce the number of variables involves propagating a signal from a function generator into a pickup, which will then propagate to another pickup through magnetic fields. With this we can generate a reliable set of frequency response curves that will help us better analyze the pickups. Classes such as Physics II and Fields and Waves have helped us understand the physical behavior of pickups and how to better optimize them for our desired specifications.

Block 2: Amplifier and ADC

These two sub-blocks are concerned with taking the analog signals from the pickups, and converting them to digital. Before the signal can pass through an analog to digital converter (ADC), the signal that is generated from the pickups must be amplified to be usable for the ADC. The amps will be placed right next to the pickups on the guitar (there will be six of them, one next to each pickup), so that the weak signal from the pickups do not travel far and collect too much noise before the amp. In order to amplify the signal, we will be using operational amplifiers (op amps), in a non-inverting configuration [7]. Classes such as Electronics I & II and Circuit Analysis I & II have provided us sufficient background knowledge to deal with these design considerations.

In our first prototype, we utilized the TL072 op amps from TI. The main issues with these are their high minimum rail-to-rail supply voltage needed (7V), and relatively low GBW (3MHz) [8]. We needed an amplifier that would work on a single 3.3V rail (due to the voltage requirements of the other chips in the system), and one with a higher GBW (that would support 100+ gain and 20 kHz bandwidth at the same time). So we decided to switch to the OPA320s from TI. The OPA320 series is ideal for low-power (min 1.8V rail-to-rail), single-supply applications, such as ours. Low-noise (7nV/√Hz) and high-speed operation (20 MHz GBW) also make them well-suited for driving sampling ADCs [9]. These will allow us to amplify the signal in a single gain stage, rather than the two stages needed previously. As for now, though, the two stage configuration meets the required gain/bandwidth, as can be seen in Fig. 2.

While testing both pickups and amps together (with an oscilloscope), we noticed that there is a wide range of amplitudes that could be detected, depending on how loud the

guitar is being played. Because of this, there is a possibility that the signal will get clipped if there is too much gain, or will not be visible if there is too little gain. Therefore, we also need to be concerned with gain control. This can be achieved with a separate circuit dedicated to automatic gain control (AGC), or we can use the integrated AGC that is found in the ADC that we have chosen.

The ADC we will be testing is the PCM1863 from TI. Each chip contains 2 ADCs, so we will need 3 chips in total for all 6 pickups. This ADC has our preferred control and digital audio interfaces (SPI and I^2S, respectively), has a maximum sample rate of 192 kHz (we require around 40 kHz), and 24 bits of resolution (we require 16 bits) [10]. It also has a useful feature called 'Energysense' [10], which will allow the ADCs to power down if they are not currently in use (that is, the strings associated with those particular ADCs are not currently being played). The ADC needs to be able to successfully sample the analog signal coming from the amplifier at the specified rate. In order to test that it is working, we can graph the memory contents to a computer using TI's free IDE, Code Composer Studio (CCS) [11]. The debugger can plot data from a specified range of memory addresses when code execution is paused, which can be done over JTAG [12]. This can also be done (albeit slower) by simply dumping the memory contents to a spreadsheet using a serial port.

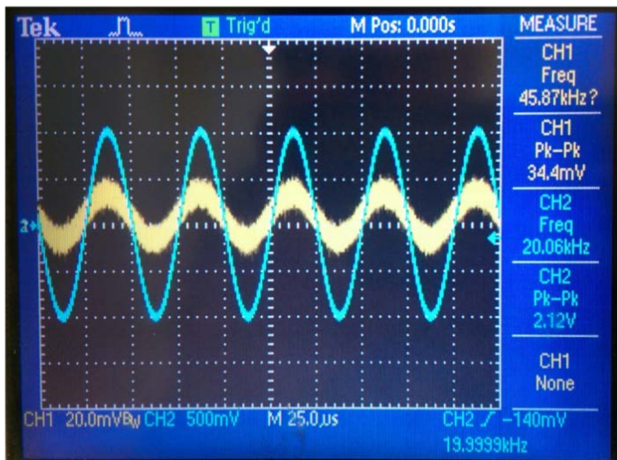


Fig. 2. This shows an amplification of ~100 at 20 kHz.

Block 3: Microcontroller

The microcontroller involves taking the signal produced by the ADC and converting this digital time domain information into frequencies. We have decided to do this conversion in hardware rather than in software in order to save bandwidth. In other words, when we convert the signal to the frequency domain, we will be able to cut off frequencies that are not of interest, therefore reducing the amount of information that needs to be sent to software. This will allow us to utilize low bandwidth connections, such a Bluetooth.

The microcontroller we have chosen to use is the TMS320C5535 from TI, with the deciding factor being the FFT coprocessor in TI's C5535 DSPs, since our project depends heavily on FFT performance. Performance data published by TI suggests that it is capable of completing over 9,000 1024 point FFTs per second, with example code for extending this to 2048 points using software [13]. The BGA package the chip comes in is difficult to work with and will increase PCB costs, but performing FFTs in hardware will greatly increase performance (see Figure 3). We chose the C5535 processor because unlike the C5505 and C5515, it can be used on a 4 layer board. The 5505 and 5515 have more pins and a tighter pitch. This requires a 6 layer board, which we cannot get cheaply. The C5535 only differs by not having an external memory controller [14]. The processor has 4 I^2S busses with DMA [14], although we only require 3 in order to stream 6 channels of audio.

TI provides a developer board (dev board) with many peripherals and sample code to operate it, and, with Code Composer Studio, the dev board can be programmed using a USB Bootloader. We can connect the other hardware in our system to the dev board's edge connector until we are ready to order a PCB.

FFT Performance on HWAFFT vs CPU (Vcore = 1.3 V, PLL = 100 MHz)

Complex FFT	FFT with HWA		CPU (Scale)		HWA versus. CPU	
	FFT + BR [1] Cycles	Energy/FFT (nJ/FFT)	FFT + BR [1] Cycles	Energy/FFT (nJ/FFT)	x Times Faster (Scale)	x Times Energy Efficient (Scale)
8 pt	92 + 38 = 130	36.3	196 + 95 = 291	145.9	2.2	4
16 pt	115 + 55 = 170	49.3	344 + 117 = 461	241	2.7	4.9
32 pt	234 + 87 = 321	106.9	609 + 139 = 748	414	2.3	3.9
64 pt	285 + 151 = 436	151.3	1194 + 211 = 1405	815.7	3.2	5.4
128 pt	633 + 279 = 912	336.8	2499 + 299 = 2798	1672.9	3.1	5
256 pt	1133 + 535 = 1668	625.6	5404 + 543 = 5947	3612.9	3.6	5.8
512 pt	2693 + 1047 = 3740	1442.8	11829 + 907 = 12736	7823.8	3.4	5.4
1024 pt	5244 + 2071 = 7315	2820.6	25934 + 1783 = 27717	17032.4	3.8	6

[1] BR = Bit Reverse

Figure 3: The table shows that for the test conditions used, HWAFFT is 2.2-3.8 times faster than the CPU

Block 4: Software Analysis

The spectrum analyzer will output frequencies, amplitudes, and timestamps from the microcontroller. The PC's main task is to interpret and analyze the data in order to determine the most probable notes and their corresponding timestamp. One that is done, the results will be packaged in a format readable to the GUI such that they could be displayed to the user.

The interpretation, analysis, and packaging steps will all be done in Python. The following steps will be done for each of the 6 streams, each representing a guitar string, coming from the microcontroller. First, the onset of the notes will be determined using the Python reference implementation of the SuperFlux onset detection algorithm [15][16]. The program will then use the onset timestamps to divide the stream of FFTs into individual chunks each representing a singular

note. For each partition, the software will run several algorithms that will estimate the fundamental frequency. Those algorithms include estimating by counting zero-crossings, estimating frequency from the peak of the FFT, estimating frequency using autocorrelation, and estimating frequency using harmonic product spectrum. The program will then determine the most likely fundamental frequency based on these results and assign it to the note it represents for the given string. If there is significant uncertainty, the program may be trained to estimate undetermined notes through artificial intelligence and machine learning techniques, such as Hidden Markov Model or the Viterbi algorithm. Once completed, the list of notes and their corresponding time stamps will be packaged and sent to the GUI.

The development of this block required techniques from several courses and out of school experience. Signals & Systems (ECE 313) provided a general understanding of signals, sampling, and FFTs such that I am now able to determine which algorithms would work best for determining the fundamental frequencies. Introduction to Algorithms (CMPSCI 311) provided me with a more in-depth knowledge of the design and runtime of algorithms and dynamic programming techniques. The rest of the techniques used in this block self-taught and through experience gained at a summer internship as a software engineer. Most of the basic knowledge for designing this block is in place, so much of the work will involve learning more about Python syntax and libraries [17]. However, a more in-depth understanding of signal processing and analysis will allow for better algorithm design and implementation.

In order to ensure the notes are getting correct assignments, several recordings of different and similar notes from different strings will be processed in order to simulate the input to the block. These processed inputs will be then be analyzed by the code, which will return what it determined to be the correct notes. The analysis of this experiment is fairly straightforward. If the notes the system returns correspond to the notes that were placed through the system, then we will know that the system is able to suitably identify notes. However, if the notes are incorrect, further testing and experiments will have to be designed in order to determine the issue, which may be caused by various factors. The new wave of tests will have to account for the quality of the signal received by the microcontroller, the processing done by the microcontroller, and the various algorithms mentioned earlier that are used to determine the fundamental frequency.

Block 5: Graphical User Interface

The GUI will be a minimalistic user interface allowing for file I/O, viewing, and editing of basic tablature. For compatibility purposes, the application will be written in Java. Specifically, the Java Swing library is a strong option for developing compatible user interfaces. Furthermore, polymorphism and inheritance, practiced heavily in GUI development, are particularly simple to implement with Java. As an object-oriented language, unit-testing for each class will be possible. For interfacing between spectrum analysis software and the GUI, a simulated test file will be written in the same fashion as it will be in the final product for independent testing. UX feedback will be incorporated in later stages of development.

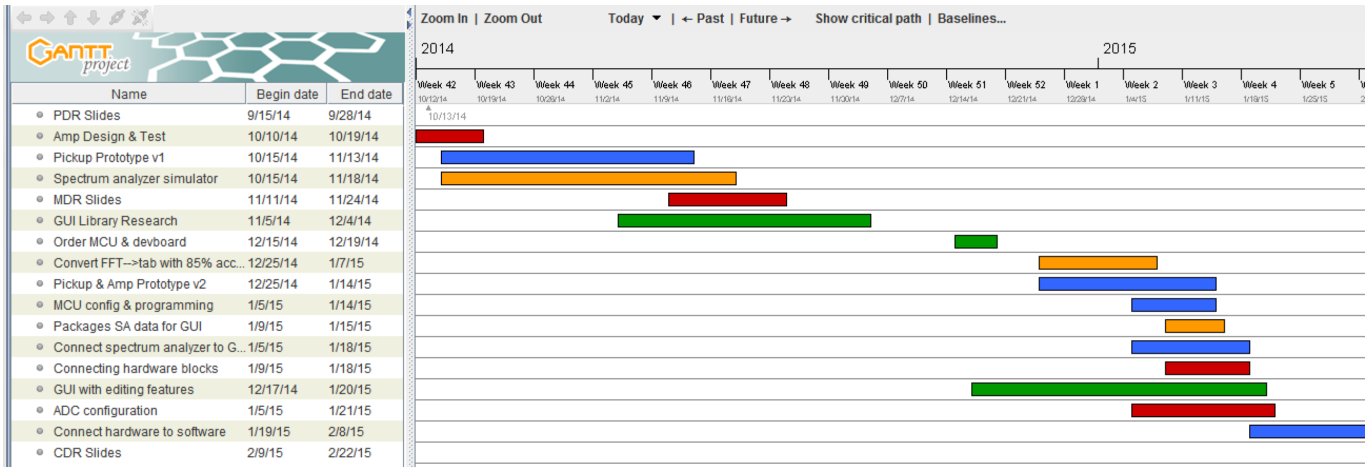
III. PROJECT MANAGEMENT

MDR Deliverable	What has been accomplished?
Purchase and prototype pickup	The prototype worked, and successfully generated a signal around 300mV
Generate and propagate signal to AFE	The signal was successfully propagated.
Amplify signal from pickup	The signal was amplified with 100+ gain
Select and purchase hardware capable of FFTs	Hardware has been selected. The order has yet to be placed due to financial considerations.
Identify a single note along with its corresponding onset timestamp	A note has been successfully identified and the timestamp is correct.
GUI with tablature display	Due to the C++ widget’s incompatibility with Windows 8.1, a different approach has been designed and will be implemented in the coming weeks.

Our team is certainly multitalented. Mike does mechanical design and prototyping, 3D modeling, hardware testing. He has experience working with microcontrollers and PCB design. Matt does analog design for the amplifiers. Michaela designs the algorithms for the microcontroller and implements algorithms for the spectrum analyzer to detect the correct notes and their onset. Taryk designs user-friendly graphical interfaces that present meaningful data. All team members have been in constant communication both in-person and via email and discussing the project at least 3 times a week and holding an additional weekly meeting with the faculty advisor.

IV. CONCLUSION

Most of what we had planned to accomplish by MDR is working as described with the exception of the GUI. We are expecting some issues with programming the MCU and with software portability and reliability. Another major issue is accurate note and onset detection. Our future plans have been laid out in the Gantt chart below.



Gantt Chart: Fall 2014-Spring 2015

V. References

[1] Reid, H , "On Guitars and Musical Notation". Retrieved December, 2014 Available: http://www.woodpecker.com/writing/essays/guitarnotation.html

[2] Krenz, S , "Guitar Tab vs. Standard Notation". Retrieved December, 2014 Available: http://www.learnandmaster.com/guitar-blog/gibsons-learn-master-guitar/guitar-tab-standard-notation

[3] "Guitar Pro". Retrieved December, 2014 Available: http://www.guitar-pro.com/en/index.php

[4] Tuohy, D.R. et al., A Genetic Algorithm for the Automatic Generation of Playable Guitar Tablature , Artificial Intelligence Center

[5] Boley, S. et al., AutoTab Automatic Guitar Tablature Generation, 2014

[6] Barbancho, I et al., Inharmonicity-Based Method for the Automatic Generation of Guitar Tablature, IEEE Transactions on Audio, Speech, and Language Processing, vol 20 , no 6, Aug. 2012.

[7] Sedra & Smith, Microelectronic Circuits. pg. 67. The Noninverting Configuration.

[8] "TL072". Retrieved December, 2014 Available: http://www.ti.com/product/TL072

[9] "OPA320". Retrieved December, 2014 Available: http://www.ti.com/product/opa320

[10] "PCM1863". Retrieved December, 2014 Available: http://www.ti.com/product/PCM1863/description

[11] "Code Composer Studio (CCS) Integrated Development Environment (IDE)". Retrieved December, 2014 Available: http://www.ti.com/tool/ccstudio

[12] Tomar, A , "Texas Instruments: Code Composer Studio (CCStudio) IDE Overview". Retrieved December, 2014 Available: http://www.element14.com/community/docs/DOC-39427/l/texas-instruments-code-composer-studio-ccstudio-ide-overview

[13] McKeown, M , "FFT Implementation on the TMS320VC5505, TMS320C5505, and TMS320C5515 DSPs". Retrieved December, 2014 Available: http://www.ti.com/lit/an/sprabb6b/sprabb6b.pdf

[14] "TMS320C5535 (ACTIVE) Fixed-Point Digital Signal Processor ". Retrieved December, 2014 Available: http://www.ti.com/product/TMS320C5535/datasheet

[15] Sebastian Böck and Gerhard Widmer, "Maximum Filter Vibrato Suppression for Onset Detection", 2013

[16] Sebastian Böck and Gerhard Widmer, "Local Group Delay Based Vibrato and Tremolo Suppression for Onset Detection", 2013

[17] Mark Lutz, *Learning Python*, 5th edition. Sebastopol: O'Reilly Media, 2013