

Enhanced Synthesizer Guitar
With MIDI Output and Automatic Tablature Composing

Nathan McKay Monroe

Advisor: Gim Hom

Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

May 2013

Abstract

An acoustic guitar was enhanced with digital and analog circuitry to add functionality for automatic tablature composing and MIDI output. A Programmable System on a Chip (PSoC) was used as the basis for the system. Digital note sensing was used in a fashion similar to keyboard matrix circuits to detect note information. This information was sent to a PC using I2C, and a companion python script was used to generate guitar tablature. Digital note sensing was augmented with custom string-selective pickups, providing digitized vibration information for each string. This information was used to extract ‘note off’ events, as well as a decision making algorithm for guessing notes during chord play. This data was output from the PSoC as MIDI, allowing the guitar to act as a fully functional MIDI synthesizer.

Table of Contents

Abstract	2
1 Author's note	4
2 Background	4
2.1 Guitars and Guitar Tablature	4
2.2 MIDI	6
2.3 Existing Solutions	8
3 Overview of Solution	10
4 Automatic Tablature Functionality	13
4.1 Digital Sensing of Note Input	13
4.2 PSoC I2C Code / Hardware	15
4.3 Arduino Code / Hardware	15
4.4 PC-Side Handling	16
4.4.1 Chord Functionality	17
5 MIDI Output Functionality	17
5.1 Analog Sensing Hardware (External)	17
5.2 Analog Sensing Hardware (Internal)	19
5.2.1 Amux	19
5.2.2 Programmable Gain Amplifier	19
5.2.3 ADC	20
5.3 Initial Data Processing	20
5.4 Autocalibration feature	22
5.5 MIDI Hardware	23
5.6 MIDI Note On Event (note to midi mapping)	23
5.7 MIDI Note Off Event	24
5.8 Note Collision Decision Making (G chord diagram)	25
6 Future Work	26
7 Conclusion	27
8 Appendix	28
8.1 PSoC Hardware	28
8.2 PSoC Code	29
8.3 PSoC Pinout	37
8.4 Arduino Code	39

8.5 Python Code.....	39
----------------------	----

1 Author's note

The author would like to thank his parents, Drs. Mark Monroe and Julie McKay, for their endless support throughout MIT. He would also like to thank his thesis advisor, Gim Hom, for his support and advice along the way. He would also like to thank Cypress Semiconductor and Professor Steven Leeb for providing the PSoC hardware, software, and instruction. This work was supported by the MIT department of Electrical Engineering and Computer Science.

2 Background

2.1 Guitars and Guitar Tablature

A guitar is an acoustic musical instrument. It consists of six metal strings, and between 12 and 24 metal frets. When playing a note, a musician will 'fret' a note, which consists of pressing down a string onto a fret to shorten the string's vibrating length. The musician will then displace the string, causing it to vibrate and sound out. The strings are conventionally numbered 1-6 from bottom to top, and frets are numbered 1-n, with 1 starting at the end of the neck opposite the body of the guitar. A string that is played 'open', or without any frets, is typically referred to as being played with the "zeroth fret". This convention will be followed in this report. A guitar is inherently an analog device. Although many guitars have no electronic components, some guitars do have magnetic transducers, which combine the vibration of all six strings into a single voltage signal.

Music for guitarists is typically notated in two ways. The first is classical music notation (see fig. 1a). In this notation, pitch is represented vertically, and time is represented horizontally along the X axis, with increasing time from left to right. “Chords”, or multiple notes being played simultaneously, are represented with notes in a column at a given time. Classical notation includes timing information, notably how long each note should be played.

The second way is called tablature (see fig. 1b). Tablature is typically favored by guitarists, because its physical mapping is analogous to a guitar and is thus more intuitive to read. Guitar tablature consists of six horizontal lines, with one line corresponding to each guitar string. The top line in tablature corresponds to the first string, the lowest string physically on the guitar (but the highest string pitch-wise). As in classical notation, time is represented along the X axis, with increasing time from left to right. Notes are represented numerically along each string. The number corresponds to what fret is being played on that string. As in classical notation, chords are represented as columns of notes. Crucially, guitar tablature omits some timing information. It contains no information of how long a note is held, or of any pauses between notes. It only includes the order which notes are played. Thus, while it is favored by guitarists because of its intuitiveness, this comes at the cost of losing timing information.



Figure 1. Guitar music notated in a) classical notation and b) tablature notation.

Whether being represented in classical or tablature notation, guitar music is typically input into a computer manually, either with specialized software or occasionally a simple text editor in the case of tablature. This process of inputting the notes by hand is long and tedious, and wastes precious time that could be spent actually creating music. In addition, it stifles musical creativity. Finally, the tedious process discourages many musicians from ever transcribing their music into a written form, suppressing the proliferation of musical information and ideas.

2.2 MIDI

MIDI (Musical Instrument Digital Interface) is a digital communication protocol designed for communicating musical information between devices. It is instrument-independent. MIDI allows compatible instruments to communicate with each other, and with other electronic

devices. For example, a MIDI-enabled keyboard could be connected to a computer, where notes played are recorded. This allows music from a MIDI device to be easily captured, saved, and transmitted. MIDI software also typically provides features such as being able to map note information onto another instrument, as well as useful musical transformations such as transposing the notes into another key, editing played notes, or speeding up or slowing down the notes. This is the foundation for digital synthesizers, where any instrument can sound like any other instrument, using MIDI and appropriate software and synthesis software. For example, MIDI allows a keyboard player to sound as if their music is being played by a violin or any other instrument, or even an entirely new tonal sound that doesn't come from an existing musical instrument.

A typical MIDI protocol packet consists of three parts: MIDI note being updated, the note's velocity, and 'on' or 'off' event. The MIDI note comes from a standard mapping of musical note to MIDI note (see fig. 2). Velocity indicates how hard the note is being played, from 0 to 100, with 100 being the loudest. This is useful for dynamic instruments, such as a MIDI drum set, where notes can be played at different volumes. The event information communicates if the note in question is being turned on or off. In the MIDI protocol, a note 'on' event with velocity of 0 is equivalent to a note 'off' event.

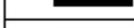
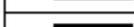
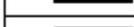
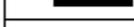
MIDI number	Note name	Keyboard	Frequency Hz	Period ms
21	A0		27.500	36.36
23	B0		30.868	29.135
24	C1		32.703	30.58
26	D1		36.708	34.648
28	E1		41.203	38.891
29	F1		43.654	22.91
31	G1		48.999	46.249
33	A1		55.000	51.913
35	B1		61.735	58.270
36	C2		65.406	15.29
38	D2		73.416	69.296
40	E2		82.407	77.782
41	F2		87.307	11.45
43	G2		97.999	92.499
45	A2		110.00	103.83
47	B2		123.47	116.54
48	C3		130.81	7.645
50	D3		146.83	138.59
52	E3		164.81	155.56
53	F3		174.61	5.727
55	G3		196.00	185.00
57	A3		220.00	207.65
59	B3		246.94	233.08
60	C4		261.63	3.822
62	D4		293.67	277.18
64	E4		329.63	311.13
65	F4		349.23	2.863
67	G4		392.00	369.99
69	A4		440.00	415.30
71	B4		493.88	466.16
72	C5		523.25	1.910
74	D5		587.33	554.37
76	E5		659.26	622.25

Figure 2. Standard mapping from musical note to MIDI note. Courtesy of <http://www.phys.unsw.edu.au/jw/notes.html>

2.3 Existing Solutions

No known solution exists for converting guitar musical information directly into either classical or tablature representation, other than inputting notes manually. Optical character recognition (OCR) technology is improving, allowing music which has been composed on paper by hand to be converted into a digital representation, but this still requires manual transcribing onto paper. This void in an adequate solution is an obvious gap in accessibility for guitar players, and a solution would help countless musicians.

In terms of MIDI, many solutions exist for other instruments. For example, MIDI saxophones exist which give saxophone players access to the features described above. Once

music is already recorded in MIDI information, it is simple to translate to classical or tablature notation with existing software packages.

Unfortunately, attempts at MIDI implementation for guitar are few and far between. Many solutions attempt to extract musical notation from the output signal of existing magnetic guitar pickups. The main disadvantage here is that signals from all six strings are combined into a single voltage signal. Systems attempt to low-pass filter the signal, take FFT and estimate note from frequency content by comparing peaks in frequency content to a threshold. However, in the case of polyphonic input (chords), frequencies overlap and it becomes much more difficult to extract any useful information from the frequency domain. As a result, most of these implementations are monophonic, meaning they only function with a single note at a time. This prevents guitarists from playing chords, drastically reducing the implementation's usefulness. In addition, this implementation is only compatible with electric guitars, leaving out guitarists with solely acoustic instruments.

Another implementation currently on the market is based on custom-designed guitar pickups, where each string has its own voltage signal to convey vibration information. Since each string has a maximum of one note playing on it at a given time, this implementation breaks down into six monophonic implementations being run in parallel. Each individual string's signal can be filtered and processed to extract its note, and the notes can then be combined into chords, or polyphonic functionality. There are a few disadvantages to this implementation. For one, the added custom sensing hardware, plus the signal processing hardware and software are expensive. Existing systems with this implementation run in the hundreds of dollars, not including the guitar. This cost is prohibitive for many musicians. Second, this implementation necessitates

installing and mounting third party hardware in an existing guitar, which many musicians are either reluctant to do, or lack the necessary technical knowledge or equipment.

3 Overview of Solution

The main goals of this project were the following:

- Design a system surrounding an acoustic guitar, allowing for automatic tabbing functionality where a guitarist plays the guitar and tablature is automatically and seamlessly produced based on what is played, including chords

- Implement polyphonic MIDI output functionality such that the guitar can act as a synthesizer controller, and is compatible with existing MIDI equipment and software

- Total system cost to be below competing polyphonic MIDI guitar systems

The primary inspiration for this design was derived from keyboard matrix circuits (see fig. 3).

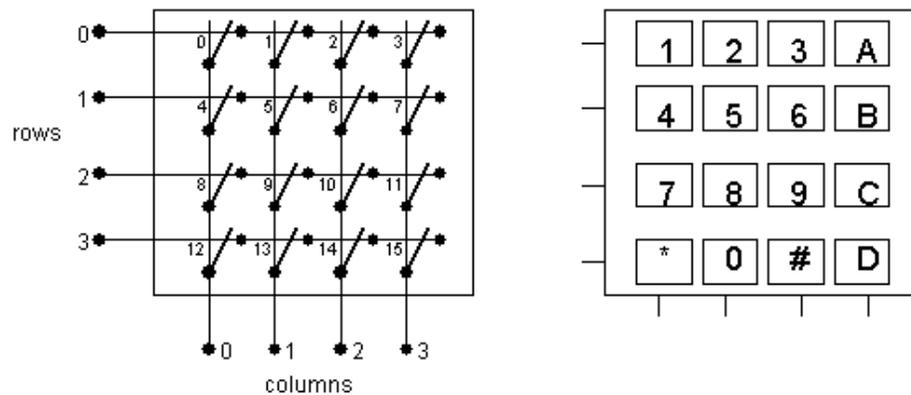


Figure 3. A standard configuration for keyboard matrix circuits. Courtesy of <http://www.gsl.net/pa3ckr/bascom%20and%20avr/keyboards/keyboard-4x4.gif>

In a keyboard matrix, there are rows of horizontal wires and columns of vertical wires running throughout the keypad, creating a grid. When a key is pressed, an electrical short is created between the corresponding horizontal and vertical wires. A microcontroller will ‘scan’ the keys, by applying pulses of logic ‘high’ voltage on each row of wires, and sensing voltage on the columns of wires. If a key is pressed, a column will read logic ‘high’, and the identity of the key can be determined from the row/column combination, much like X/Y triangulation.

This design was based on a similar principle, where the metal guitar frets are analogous to the wire rows and metal guitar strings analogous to wire columns. When fretting a note, an electrical short is created between the row and column, enabling detection of hand position and thus note. The ‘scanning’ action is analogous to a metal pick, which is connected to a logic high. When a string is picked, the musical note information is determined by the combination of fret and string which return logic high. This information is communicated to an Arduino microcontroller and then to a PC, where a companion Python script converts it into guitar

tablature (see fig. 4). Thus, musical information is determined by examining hand position, rather than any direct analysis of signal output.

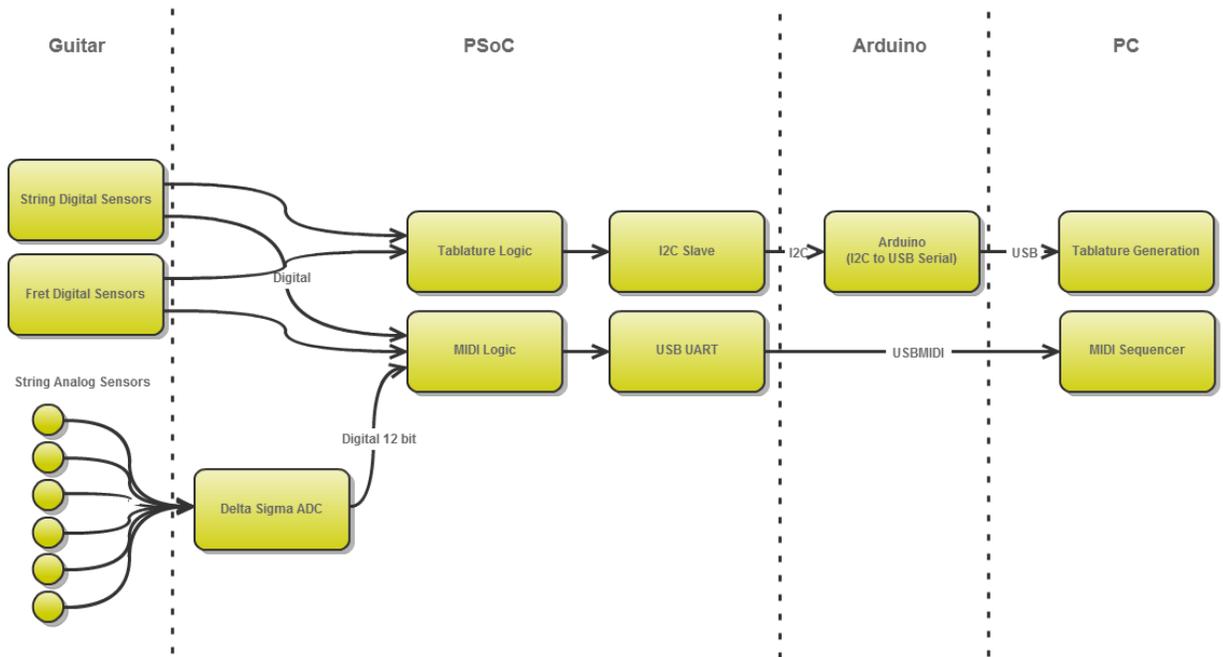


Figure 4. Block diagram representing the tablature and MIDI subsystems.

Signal output analysis is used for the MIDI component of the system. The above detection scheme is adequate for guitar tablature because although it has no sense of when a note is finished playing, this information is not necessary for guitar tablature. However, MIDI functionality necessitates note “off” events, so more information is required. This information comes from custom-built single-string pickups, which provide to the PSoC an analog signal representing string vibration for each individual string, which is then digitized using an ADC. In a general sense, note “on events” are determined by the above keyboard matrix fashion. Note “off” events are triggered when the peak amplitude of a string’s sinusoidal output signal goes below a threshold. This information is processed by the PSoC, and is transmitted to a PC via a

UART configured for the USB MIDI protocol. Automatic calibration for the ADC signals is implemented as well, to reduce the effect of EM noise from the pickups, sampling noise, steady-state error, and variances due to the proximity of the metal pick to the string pickups. String analog signals are also used to resolve note “collisions”, when certain note configurations create ambiguity in what note is being played.

4 Automatic Tablature Functionality

The general purpose of this block is to translate hand position into note information, process it, and display it in standard tablature notation on a PC.

4.1 Digital Sensing of Note Input

Digital sensing begins with wires soldered onto each of frets 1-12 of the guitar (see fig. 5a). Wires were soldered such that they do not interfere with typical guitar playing. There are also electrical probes attached to each of the six strings (see fig. 5b). They are attached below the bridge in a nonvibrating part of the string, such that they are electrically connected to the string but don't affect vibration. These signals all lead to input pins on the PSoC. In addition, a copper guitar pick with soldered wire is connected to logic high. To prevent interference with ADC readings (see section 5.2), pick is connected to analog VDD, VDDA.

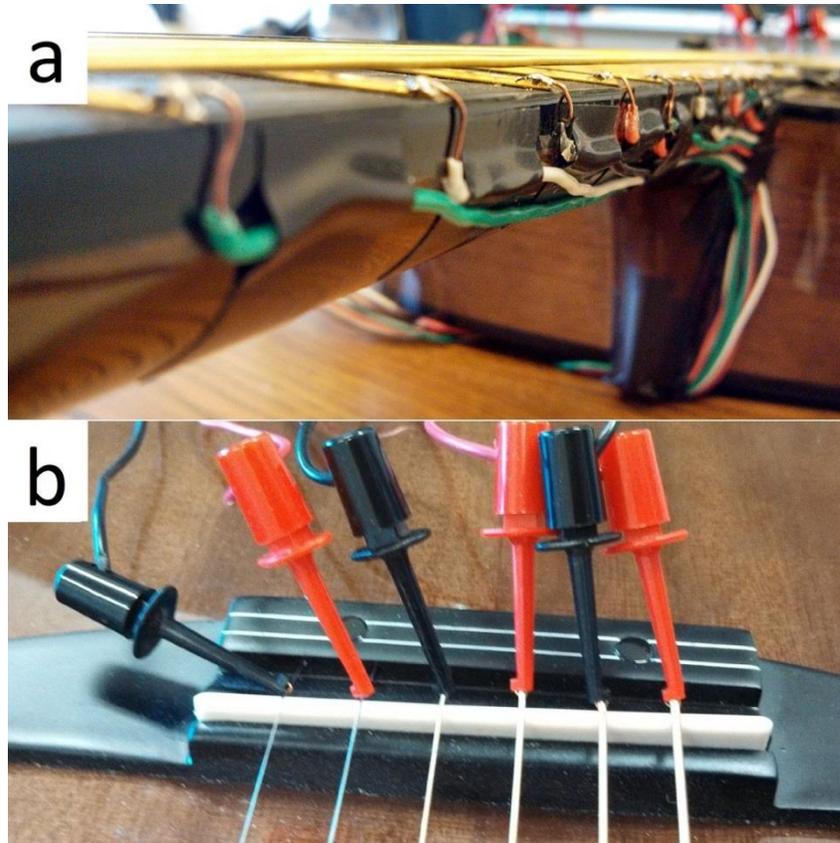


Figure 5. Electrical connections for a) fret digital sensing and b) string digital sensing.

On the PSoC side, input pins were configured as digital input pins with resistive pull down. CMOS threshold levels were used to correspond with VDDA. See Appendix 8.3 for PSoC pin connections.

In software, logic was based on one simple assumption, that only one string will be high at a given time. This holds true for individual notes. Even for chords, where on a macro level multiple notes are being played simultaneously, there is in fact only one string being picked at a given time, with a very brief delay between successive strings for a strummed chord. Thus, notes could be handled on a one-at-a-time basis, with chord processing done on the PC side based on timing information. This assumption about strings being mutually exclusive was proven to be

false in certain occasions, which are being referred to as note collisions. This is discussed in section 5.7, with the result being analog information being taken into account when deciding on string/note combinations.

The information conveying which string and fret is high is decided simply by looking at which string/fret returns a logic high. If a string reads high but no fret does, it is assumed the note is being played open, with no frets. String/fret information is passed on to the I2C buffer.

4.2 PSoC I2C Code / Hardware

I2C stands for Inter-Integrated Circuit. It is a simple serial communication protocol and in this case was used to communicate string and fret information to the PC. On a hardware level, I2C was implemented in the PSoC using the EZI2C Slave module. The module was configured as an I2C slave with a bitrate of 100kbps, and slave address of 0x36.

I2C buffer slave-side was implemented as a 3 byte array. The first byte holds string information, the second byte holds fret information, and the third byte, the “event” byte, relays if a string is being played or not. String and fret information are constantly updated into buffer[0] and buffer[1]. Buffer[2], the event byte, only transitions to 0x1 if any string is logic high, and is 0x0 otherwise. This raw information is processed by the PC to parse out non-note garbage, as well as preventing repeat notes.

4.3 Arduino Code / Hardware

The original intent was to use an I2C to USB debugger (minipro3 provided by Cypress Semiconductor) to input I2C note information to the PC. However, attempts to extract the data in a usable form for a Python or Matlab script proved unfruitful. Data could only be read inside Cypress’ proprietary software, Cypress Bridge Control Panel. This eliminated the possibility of

scripting. Instead, the decision was made to use the Arduino microcontroller as an intermediary, simply taking in I2C information from the PSoC and relaying it to the computer via serial USB. Although this is not the cheapest, most elegant, nor least-parts solution, the Arduino was already freely available so this was the fastest and cheapest solution.

Arduino serial connection to the PC was implemented at 57.6 KBaud through USB using the Arduino Wire library. Three bytes were read from the PSoC I2C buffer, stored briefly in an Arduino buffer, and then sent over USB serial to the PC. String information was preceded by an 's', fret information by an 'f', and note event information by an 'n'. A 5ms delay was added to the end of each read-write cycle to roughly match data rates and prevent buffer overflow or buffer starvation at any point in the signal chain.

4.4 PC-Side Handling

Note information was read over serial connection by a Python script using the PySerial library. Output was stored as an array of strings, where each string corresponds to a row in the tablature output. Information was parsed using the 's', 'f', and 'n' tags from the Arduino. Implementing software edge detection on the 'n' signal prevents a single note from being reported multiple times. The serial buffer was flushed before each serial reading, to add robustness and protect against buffer overflow, and also to prevent any frame shift errors from data loss. The proper fret was appended to the corresponding string, plus a dash for spacing to improve readability. Keeping with standard tablature convention, dashes were added to strings with no note being played at that time. This also ensures that no string gets ahead of another based on what notes are being played. Finally, strings are output to the user. A maximum length of 83 notes is implemented to prevent strings from wrapping around the screen and messing up the output. Text could be easily copied and pasted by a user for recording.

4.4.1 Chord Functionality

Functionality for dealing with chords was included in the Python implementation. The logic for this is simple. If two successive notes are on adjacent strings and occur less than 0.2 seconds apart, then they are part of a strummed chord. If this is the case then the notes are added at the correct time and location to depict simultaneous playing, or a strummed chord. To allow for chords with more than two notes, the 0.2 second timer resets after each new note. The obvious assumption here is that any two notes less than 0.2 seconds apart are part of a chord, and anything more than that are distinct notes. This threshold can be adjusted by the user, as some guitarists play faster and may go below the 0.2 seconds with notes meant as distinct notes.

5 MIDI Output Functionality

The purpose of the MIDI block is to translate sensed note information into a universal musical communication protocol, MIDI, allowing the guitar to be compatible with other MIDI devices and importantly, act as a synthesizer controller. Extra hardware is needed for the MIDI feature, as information is needed to convey when a note event is ending. This information cannot be conveyed by the sensing schemes described above. Thus, the hardware and software described below generates that new information, then processes it and the existing information to produce MIDI protocol signaling.

5.1 Analog Sensing Hardware (External)

In the context of MIDI, the first point of contact between the PSoC and the guitar is through custom made string-selective pickups (see fig. 6). This is similar to a conventional guitar pickup, which uses magnetic coils to convert string motion into an analog voltage signal. The

primary difference is that in conventional guitar pickups, coils from all six strings are wired in series or parallel, resulting in a single signal for all six strings. In this modified design, each string has its own pickup and is entirely independent of the other strings. The output is six analog signals, one for each string. The pickups were made using modified 50 ohm buzzer coils, with casing and diaphragm removed. Buzzer coils are ideal because they are designed to convert a voltage signal into motion of a metallic object at audio frequencies, and are generally good at working the other way around. The six analog signals are input to analog sensing pins on the PSoC.

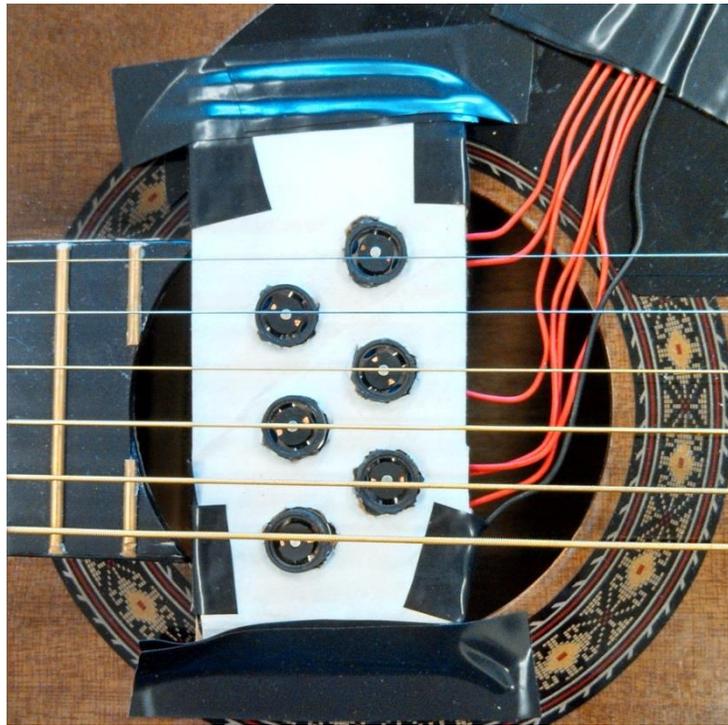


Figure 6. String-selective magnetic analog pickups.

5.2 Analog Sensing Hardware (Internal)

One excellent advantage of the PSoC is its flexibility and compatibility with both digital and analog signals. Pins can be configured as input or output pins, and as analog or digital pins. In this case, the pins were configured as high-impedance analog input pins.

5.2.1 Amux

The six incoming signals connect internally to an analog multiplexer. Because the PSoC only has three analog to digital converters (one DelSig and two SAR), it is necessary to multiplex the input signals. The mux was configured to have maximum isolation, which maximizes isolation between the input signals, at the cost of switching time. The mux was operated in 'round robin' style, where a string was sampled once before switching to the next string (see section 5.3). This switching scheme results in a maximum number of switching events, with one switch per sample. An alternative that was considered was to sample a single string multiple times before switching to the next string if mux switching time was an issue. However, the switching time was estimated at 4us, which is negligible relative to the speed of the CPU code, especially when ADC conversion rate is taken into account. The round robin scheme was used.

5.2.2 Programmable Gain Amplifier

A Programmable Gain Amplifier (PGA) was used in the signal chain after the analog mux. The purpose of the PGA was to maximize resolution and dynamic range of the ADC in order to maximize sensitivity. The PGA was configured to be referenced to Vss, with a gain of 48. There is a tradeoff here between gain and frequency response. The PGA acts as a lowpass filter, and with a gain of 48 the cutoff frequency is approximately 48KHz. Guitar signals have a

fundamental frequency in the tens or hundreds of Hertz, so the frequency response is not a factor. In addition, the ADC sample rate was set at under 6KHz (see section 5.2.3), so the higher frequencies are eliminated anyways.

5.2.3 ADC

The analog to digital converter (ADC) was implemented as a single ended Delta-Sigma ADC. The ADC was configured with a sample rate of 5862 samples per second at 12 bits resolution. The initial choice of sampling frequency was 6KHz, this maximum was reduced by constraints due to ADC configuration. The 6KHz sample rate is multiplexed over 6 strings, resulting in 1KHz sampling per string. 1KHz is adequate because a guitar signal's primary energy is in the tens or hundreds of hertz. At the Nyquist frequency, there is little to no harmonic content so aliasing is not a concern. The ADC was configured in multi-sample burst mode. In this mode, the ADC's modulator is reset and filters are flushed between each sample, eliminating the risk of contamination between successive samples. This is necessary because each successive sample is from a different independent string, and sample contamination would have a drastic effect on signal integrity. The ADC was configured with an input range from 0 to V_{ref} , with V_{ref} at 1.1V. The ADC provided an additional buffer gain of 8. Finally, the ADC was configured in 'level shift' mode, shifting the center range of the input signal and allowing signals with negative and positive components to be read properly. The level shift value was not equal to V_{ref} , resulting in a steady-state error. This was rectified in software.

The ADC was controlled using software, with sampling being explicitly halted during mux switching to eliminate any chance of data contamination. The sampling period is 170us.

5.3 Initial Data Processing

At a very simple level, all note decisions necessitating analog string sensing were based on string vibration amplitude. However, the raw magnitude itself cannot be used. Differences in magnetic properties between each string and varying distance between pickup and string, as well as manufacturing differences between the buzzers results in an uneven gain from the string vibration amplitude to the voltage signal amplitude, and thus an uneven gain from vibration amplitude to digitized sample between different strings. If two strings are vibrating at the same volume, their sampled values are not necessarily peaking at the same amplitude.

The solution implemented here was to choose thresholds normalized to the maximum amplitude the given string ever sees. For example, the threshold for a note being “off” may be $\frac{1}{4}$ of the maximum amplitude which the string will ever show. The result is uniform mapping from vibration to digital processing between strings. This scheme is based on the assumption that a guitarist will roughly pick with the same intensity between notes, which is generally a valid assumption. As a result, two values are of particular interest.

One value of interest is the absolute maximum value of the strings sampled signal. This is implemented using a simple comparison, and can only increase. This value is used for calculating thresholds and to ensure string measurements are normalized between strings.

The second value of interest is the current peak amplitude of the string’s vibration, which will be compared to thresholds to determine note events. Current peak amplitude is calculated using a sliding window of 100 samples (see fig. 7). At a sample rate of 5862KHz, 100 samples represents a time of 17.05 milliseconds. The lowest fundamental note on a guitar is an E2, with a frequency of 82.4Hz, or a period of 12.13 milliseconds. Thus, in the worst case scenario the sliding window observes 1.5 periods of the fundamental sine wave, guaranteeing that a peak will be seen.

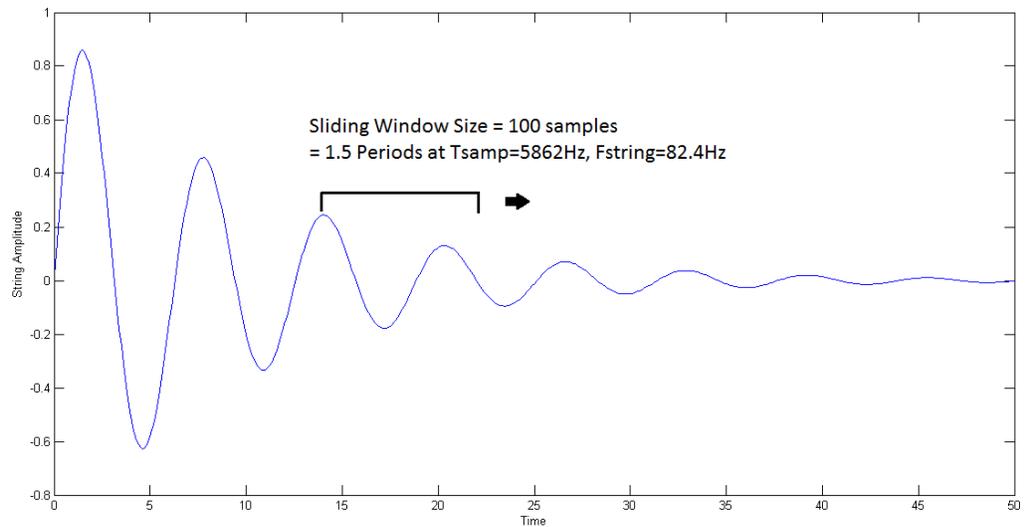


Figure 7. A sliding window of 100 samples for calculating local peak guarantees that a peak will always be seen, even in the worst-case scenario.

5.4 Autocalibration feature

The sampling process introduced steady-state error into the digitized samples, due to the level-shift configuration of the ADC. In addition, the act of playing the guitar consists of moving a metal pick in close proximity to the magnetic pickups. This resulted in an unforeseen effect, a time-varying steady state error in the pickups due to the pick interfering with the magnetic field from the pickups. Because the thresholds described above (as fractions of the maximum amplitude), it is assumed that the string amplitude is represented as an integer between 0 and the maximum value. The steady state error described above could have a drastic effect on operation.

To remove the steady-state error, an autocalibration feature was added to shift the samples between 0 and the maximum value. The autocalibration consisted primarily of a calibrate switch, which when actuated records the present maximum value of each string. It is assumed that the switch will be asserted while strings are at rest. This ‘minimum value’, being roughly equivalent to the steady-state error, was then subtracted out from all samples for a given

string. This ‘minimum value’ is constantly adjusted to ensure a given sample does not go below zero to 0xFFFF, which would drastically impair functionality. It is also adjusted to ensure the adjusted steady state error never goes above 10u. The calibrate switch also resets string absolute maximum values.

5.5 MIDI Hardware

MIDI hardware on the PSoC was implemented as a class-compliant USBMIDI device according to the USBMIDI specification. This consists mostly of a USBFS module with device descriptors configured such that the device is identified as a USBMIDI device.

5.6 MIDI Note On Event (note to midi mapping)

MIDI on events act in a similar fashion to I2C note events. “On” events are determined by sensing string and fret logic levels (see section 4.2). This was implemented in software as edge detection to prevent repeated signals sent for a single played note. Notes on the guitar map onto MIDI notes in the following fashion:

$$midi_{note} = 40 + 5 * (6 - string) + fret$$

Where string is the guitar string that is being played from 1-6, and fret is the fret that is being played. Note that due to irregularities in guitar tuning on strings 1 and 2, this number is decremented by 1 if the string being played is string 1 or 2.

The string and fret being played are saved for use in the note “off” event detection. Because each string can only play a single note at a time, if a note “on” event has been activated

for a string with no corresponding “off” event, and subsequently another note “on” event is detected for that string, the first note will be turned “off”, regardless of meeting any thresholds for a note being “off”. This is to prevent notes being dropped when multiple notes are played on the same string. Note and event packets are sent to the PSoC MIDI output buffer, which is then serviced to send the data over USB to the PC.

5.7 MIDI Note Off Event

MIDI note “off” events were determined by first noting what notes are currently “on” at a given time. For the notes that are on, if the sampled vibration goes below a certain threshold, the note is considered to be off and the corresponding signals are sent to the MIDI output buffer (see fig. 8). For open strings, which are played without a fret, the threshold is $0.5 * \text{the maximum amplitude}$. Notes played with a fret typically vibrate with a lower amplitude so correspondingly for those notes, the threshold is lowered to $0.25 * \text{the maximum amplitude}$. In addition, a delay is added following a MIDI ‘on’ event, to allow the string’s vibration to exceed the threshold before the possibility of an ‘off’ event.

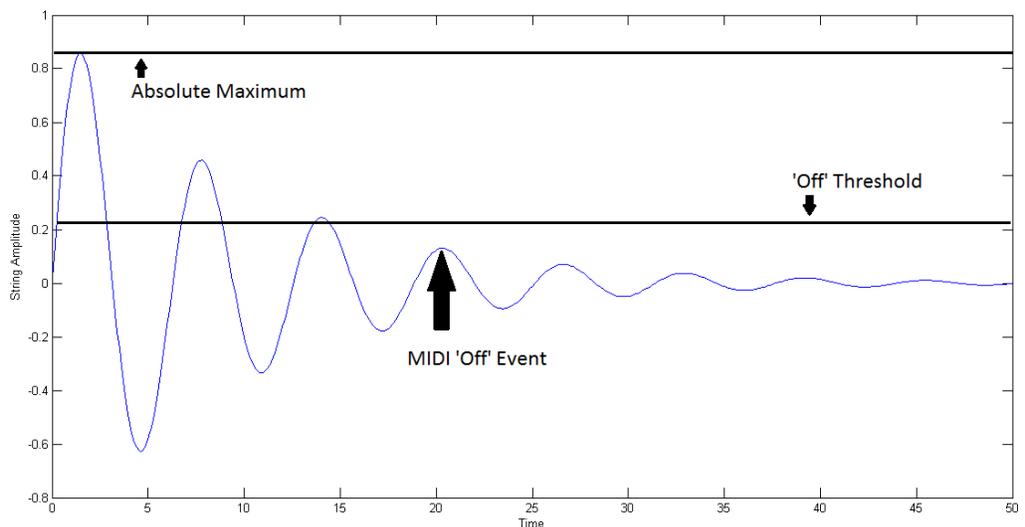


Figure 8. Note off decisions are made by comparing amplitude to a threshold value.

5.8 Note Collision Decision Making (G chord diagram)

Much of the note decision making described above was based on the assumption that only a single string is played at a time, and thus only a single string will return a logic high at a given time. However, due to an unforeseen circumstance, this is not always the case. Consider a G major chord, for example, as seen in fig. 9. In a G major chord, the third fret is asserted on both the first and sixth strings. The strings are shorted together through the fret. Thus, if the sixth string is picked, the third fret and sixth string will return logic high, but so will the first string. This creates an ambiguity in which note is being played. In the case of a strummed chord, this is not an issue because both notes will be played in the chord anyways. However, there are cases where the ambiguity still occurs and both strings aren't played, such as the case of a chord arpeggiation. In this case, the ambiguity needs to be resolved.

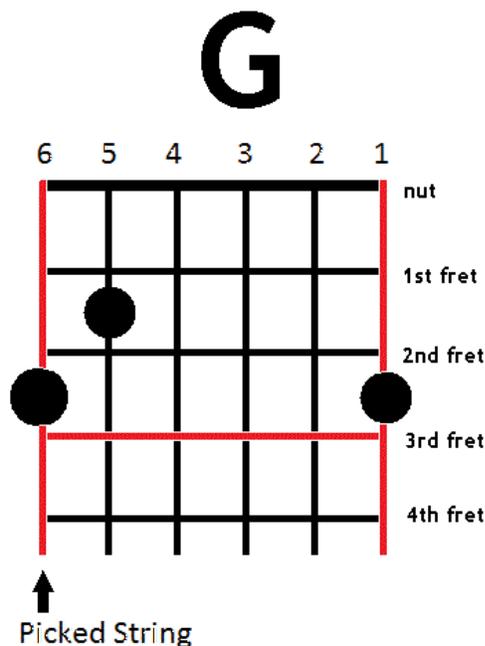


Figure 9. The ambiguous ‘note collision’ case where a chord causes two strings to return logic high when only one string is picked.

In this case, the ambiguity is resolved by observing the sampled analog information. When two strings return a logic high, a delay of $\frac{1}{4}$ period is created to allow strings to reach maximum amplitude. Then, after sampling each string once, the zero-shifted samples of the colliding strings are compared, normalized to the maximum value for each string. The string with the higher normalized magnitude is assumed to be the string that is intended to be played. This resolves the ambiguity at the cost of extra delay. To improve performance, the strings are only queried once per machine cycle of the CPU, preventing delay from adding up in the string collision case.

6 Future Work

While this work does improve upon an existing product, it could still be further improved upon. One simple improvement is to include note velocity information in MIDI output. This could be done easily by referencing ADC samples during a note ‘on’ event. While the delay from the system is acceptable for real-time music applications, it could be further reduced, especially in the case of string collisions.

In addition, note “off” thresholds could be improved upon by updating them dynamically. Since there is a variation in note intensity between guitarists, if the thresholds were updated to reflect this it would improve functionality.

The external hardware also could be improved upon. In its current proof-of-concept state, the system requires extensive modifications to a guitar, and is thus most practical when integrated into a dedicated guitar. To improve upon this, the system could be modified to act as a third-party add-on to an existing guitar. Wires soldered onto frets could be replaced with probes

that attach magnetically, allowing them to be removed easily and with no permanent modifications to a guitar.

It is also possible to implement the system without digital sensing of frets and strings. If only the sampled information was used, note on/off events and volume could be detected from sample amplitude. If each incoming signal had its own FFT, note value could also be determined. However, this would require 6 FFT modules running in parallel, so there is a tradeoff because more digital processing power is needed.

7 Conclusion

In conclusion, all design goals were met. A system was created such that a guitar can be played and tablature is seamlessly generated based on playing, including chord functionality. In addition, polyphonic MIDI functionality was implemented, allowing the guitar to act as a class-compliant MIDI device and synthesizer controller. Finally, cost goals were also met (see table 1).

Table 1. System costs

Item	Cost
Acoustic Guitar	\$45
Buzzer Coil Pickups (total)	\$10
PSoC Cy8ckit-050 development board	\$50 (donated)
Arduino UNO	\$30
Total	\$135

Total cost of the system was approximately \$135. However, this number is not an accurate reflection of actual system cost. First, the cost of the Arduino is unnecessary as it was used solely to easily translate I2C note information to the computer. In reality, this information could be sent directly over USB from the PSoC, or even extracted from MIDI information, eliminating the need for an Arduino. In addition, the PSoC used included a development board for prototyping. In reality, the cost of the PSoC itself (without the board) that would be used in mass production

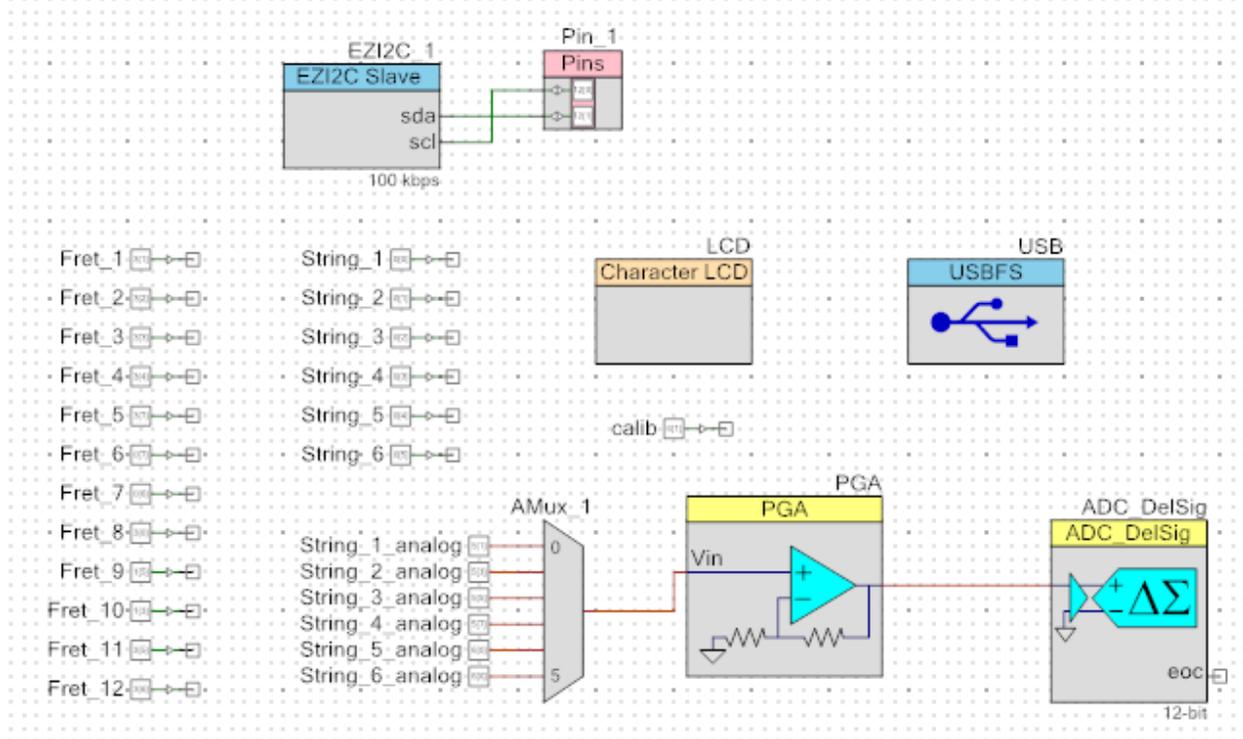
is approximately \$11, saving \$39 per unit. Thus, the final approximate cost of the system is \$66, including guitar. This can be compared to leading industry solutions, such as the Roland GK-3. The GK-3 retails for \$220 and provides polyphonic MIDI functionality, but no direct tablature functionality. In addition, \$220 is the cost of the pickup only, not including guitar. It is easily seen that the implementation described above improves functionality while drastically reducing cost.

Unfortunately, much time and effort was spent in the bring-up phase, grappling with the Cypress software and APIs and trying to get modules working such as the ADC, USB UART and analog mux. This was a byproduct of the general lack of support community surrounding the PSoC chip, complexity of PSoC designer software, and scarcity of example projects and documentation. As a result, much effort and time that should have gone to the final design instead went to dealing with PSoC related issues.

Despite PSoC issues, much was learned about the PSoC, how it works, and its enormous potential and power. In addition, much was learned about the MIDI and USB specifications. Although like any product it could certainly be improved upon, a system was created that may genuinely fill in niche in the market. Even regardless of business viability, this system could open musical doors, add convenience, and improve upon the lives of musicians everywhere. All source code, block diagrams, circuit diagrams, PSoC files, and documentation, including this report, will be published online.

8 Appendix

8.1 PSoC Hardware



8.2 PSoC Code

```

/*
Nathan Monroe
5/10/2013
6.UAP Final Project
MIDI Guitar
main.c
*/
#include <device.h>
#include "Key.h"

#define MAX_LCD_COL          16
#define ON (0x01u) //MIDI on and off
#define OFF (0x00u)

/* Identity Reply message */
const uint8 CYCODE MIDI_IDENTITY_REPLY[] = {
    0xF0u,      /* SysEx */
    0x7Eu,      /* Non-Realtime */
    0x7Fu,      /* ID of target device (7F - "All Call") */
    0x06u,      /* Sub-ID#1 - General Information */
    0x02u,      /* Sub-ID#2 - Identity Reply */
    0x7Du,      /* Manufacturer's ID: 7D - Educational Use */
    0xB4u, 0x04u, /* Family code */
    0x32u, 0xD2u, /* Model number */
    0x01u, 0x00u, 0x00u, 0x00u, /* Version number */
    /*0xF7      End of SysEx automatically appended */
};

```

```

/* Need for Identity Reply message */
extern volatile uint8 USB_MIDI1_InqFlags;
extern volatile uint8 USB_MIDI2_InqFlags;

volatile uint8 usbActivityCounter = 0u;
uint8 midiMsg[4]; //Midi message to be sent to PC
int16 string_absmax[6]; // absolute maximum value for strings. String [0]-[5] maps to
string 1-6.
uint16 string_absmin[6] = {1000u, 1000u, 1000u, 1000u, 1000u, 1000u}; //Absolute
minimum value for strings. Used to get rid of steady state error from ADC.

uint16 string_samples[6][100]; //current string value stores previous 100 samples
(~1.5 periods at 5.8KHz sampling, 82.4Hz minimum freq.
uint16 this_sample; //Current Sample from ADC
uint8 string_tocheck; //Next string to be read from ADC
uint8 sample_number = 0u; //Number of sample to be read into string_samples
uint16 curr_max; //current maximum value of string amplitude
uint8 string_active[6]; //tells if the current string is playing
uint8 fret_active[6]; //fret number of active string
uint8 thismidinote; //MIDI note to be sent out
uint8 checked_strings[6] = {0, 0, 0, 0, 0, 0}; //strings that are being checked for
play activity
uint8 t = 0; //for iterating
uint8 num_high_strings = 0; //number of strings with a high voltage
uint8 returned_string;
uint8 checked_string;
uint8 display_note; //note to display on LCD
uint16 off_thres; //Threshold for MIDI off events
uint8 delay[6]; //used for delaying analog measurements
uint8 midi_off_note; //used to prevent same-string collisions
extern uint8 const CYCODE LCD_customFonts[];

uint8 Check_Fret() { //Returns Fret number of what's being played. Higher frets have
priority, corresponding to guitar physical design.
if (Fret_12_Read()) return 12;
else {
    if (Fret_11_Read()) return 11;
    else {
        if (Fret_10_Read()) return 10;
        else {
            if (Fret_9_Read()) return 9;
            else {
                if (Fret_8_Read()) return 8;
                else {
                    if (Fret_7_Read()) return 7;
                    else {
                        if (Fret_6_Read()) return 6;
                        else {
                            if (Fret_5_Read()) return 5;
                            else {
                                if (Fret_4_Read()) return 4;
                                else {
                                    if (Fret_3_Read()) return 3;
                                    else {

```



```

        string_absmax[5] = 10u;
    }
    else{
        string_samples[string_tocheck][sample_number] = this_sample; //put sample
into array
        if ((this_sample < string_absmin[string_tocheck]))
string_absmin[string_tocheck] = this_sample - 10u; //If it overflowed negatively,
increase the string minimum so it doesn't overflow. This can really mess up note
detection.
        //if ((string_active[string_tocheck] == 0x0) & ((this_sample -
string_absmin[string_tocheck]) > 15u)) string_absmin[string_tocheck] = this_sample -
10u;

        if (string_samples[string_tocheck][sample_number] -
string_absmin[string_tocheck] > string_absmax[string_tocheck])
string_absmax[string_tocheck] = string_samples[string_tocheck][sample_number] -
string_absmin[string_tocheck]; //If a new absolute maximum sample has been found,
update the maximum array accordingly.
    }

    string_tocheck = string_tocheck + 1; //Go to the next string
    if (string_tocheck > 5u) { //Mod 5
        string_tocheck = 0u;
        sample_number = sample_number + 1; //Go to the next sample
        if (sample_number > 99u) sample_number = 0; //Mod 100
    }
    adc_change_string(string_tocheck);
}
}
uint8 Check_String() { //Returns string number of what's being played. returns 1-6
corresponding to standard string number convention.
    checked_strings[0] = 0u; //Reset values
    checked_strings[1] = 0u;
    checked_strings[2] = 0u;
    checked_strings[3] = 0u;
    checked_strings[4] = 0u;
    checked_strings[5] = 0u;
    num_high_strings = 0;
    returned_string = 0;
    if (String_1_Read()) checked_strings[0] = 0x1;
    if (String_2_Read()) checked_strings[1] = 0x1;
    if (String_3_Read()) checked_strings[2] = 0x1;
    if (String_4_Read()) checked_strings[3] = 0x1;
    if (String_5_Read()) checked_strings[4] = 0x1;
    if (String_6_Read()) checked_strings[5] = 0x1;

    for (t=0; t<6; t++) { //iterate over strings that are high
        if (checked_strings[t] == 0x1) num_high_strings = num_high_strings + 1;
    }
    if (num_high_strings == 0u) returned_string = 0; //no strings scenario
    else if (num_high_strings == 1u) { //1 string scenario
        for (t=0; t<6; t++) {
            if (checked_strings[t] == 0x1) returned_string = t+1;
        }
    }
}

```

```

else { //multiple strings scenario, collision detection and decision making
    CyDelay(5); //5ms ~half period
    read_next_string(); //Sample all 6 strings once
    CyDelayUs(200);
    read_next_string(); //2
    CyDelayUs(200);
    read_next_string(); //3
    CyDelayUs(200);
    read_next_string(); //4
    CyDelayUs(200);
    read_next_string(); //5
    CyDelayUs(200);
    read_next_string(); //6
    CyDelayUs(200);
    if (String_6_Read()){
        returned_string = 6;
    }

    //Return the string with highest normalized reading
    //Divide absmx over curr_max to save having to deal with floating point
calculations
    if (String_5_Read() & ((string_absmax[4]/curr_stringmax(4)) <
(string_absmax[returned_string-1]/curr_stringmax(returned_string-1))))
returned_string = 5;
    if (String_4_Read() & ((string_absmax[3]/curr_stringmax(3)) <
(string_absmax[returned_string-1]/curr_stringmax(returned_string-1))))
returned_string = 4;
    if (String_3_Read() & ((string_absmax[2]/curr_stringmax(2)) <
(string_absmax[returned_string-1]/curr_stringmax(returned_string-1))))
returned_string = 3;
    if (String_2_Read() & ((string_absmax[1]/curr_stringmax(1)) <
(string_absmax[returned_string-1]/curr_stringmax(returned_string-1))))
returned_string = 2;
    if (String_1_Read() & ((string_absmax[0]/curr_stringmax(0)) <
(string_absmax[returned_string-1]/curr_stringmax(returned_string-1))))
returned_string = 1;

}
return returned_string;
}

void sendMIDI(uint8 note, uint8 on) {
    if(USB_GetConfiguration() != 0u) // Service USB MIDI when device configured
    {
        midiMsg[0] = USB_MIDI_NOTE_ON;
        if (on == ON) {
            midiMsg[2] = 100u; //MidiMsg[2] is note velocity.
        }
        else {
            midiMsg[2] = 0u; //Velocity of 0 is equivalent to note turning off.
        }
        midiMsg[1] = note; //midiMsg[1] is the code of the note that's being changed.
        USB_PutUsbMidiIn(3u, midiMsg, USB_MIDI_CABLE_00); //Send the message to the
MIDI output buffer
    }
}

```

```

        USB_MIDI_IN_Service(); //Service the buffer.
    }
}

void main()
{
    int y; //for iteration
    uint8 prevbuffer2;
    uint8 buffer[3]; //I2C Buffer
                    //buffer[0] is string,
                    //buffer[1] is fret,
                    //buffer[2] is string being picked
    buffer[0] = 0; //Initialize buffer to 0
    buffer[1] = 0;
    buffer[2] = 0;
    buffer[3] = 0;

    LCD_Start(); //Initialize LCD

    AMux_1_Init(); //Initialize Mux
    AMux_1_FastSelect(0);

    PGA_Init(); //Initialize Gain amplifier
    PGA_Start();

    EZI2C_1_Start(); //Initialize I2C Slave
    EZI2C_1_SetBuffer1(3,1,(void*)(buffer));

    ADC_DelSig_Start(); //Initialize ADC and start first conversion
    ADC_DelSig_StartConvert();

    CYGlobalIntEnable; //Enable Interrupts
    USB_Start(0u, USB_DWR_VDDD_OPERATION); //Start MIDI USB Slave

    while(1)
    {
        //////////////////////////////////////////////////I2C Handling
        read_next_string();
        checked_string = Check_String(); //Do this just once per cycle to improve
performance during multi-string scenarios
        buffer[0] = checked_string; //Put String and Fret information into I2C
buffer
        buffer[1] = Check_Fret();
        if (checked_string != 0u) buffer[2] = 0x1; //Put note on information into
I2C buffer
        else {
            buffer[2] = 0x0;
        }
        //////////////////////////////////////////////////MIDI Note off Events
        for (y=1; y<7; y++){ //Go through all the strings
            off_thres = 0; //Dynamically update threshold for 'off' event based on
fret status
            if (string_active[y-1]==0x1) {
                if (fret_active > 0) off_thres = (string_absmax[y-1] >> 1);
            }
        }
    }
}

```

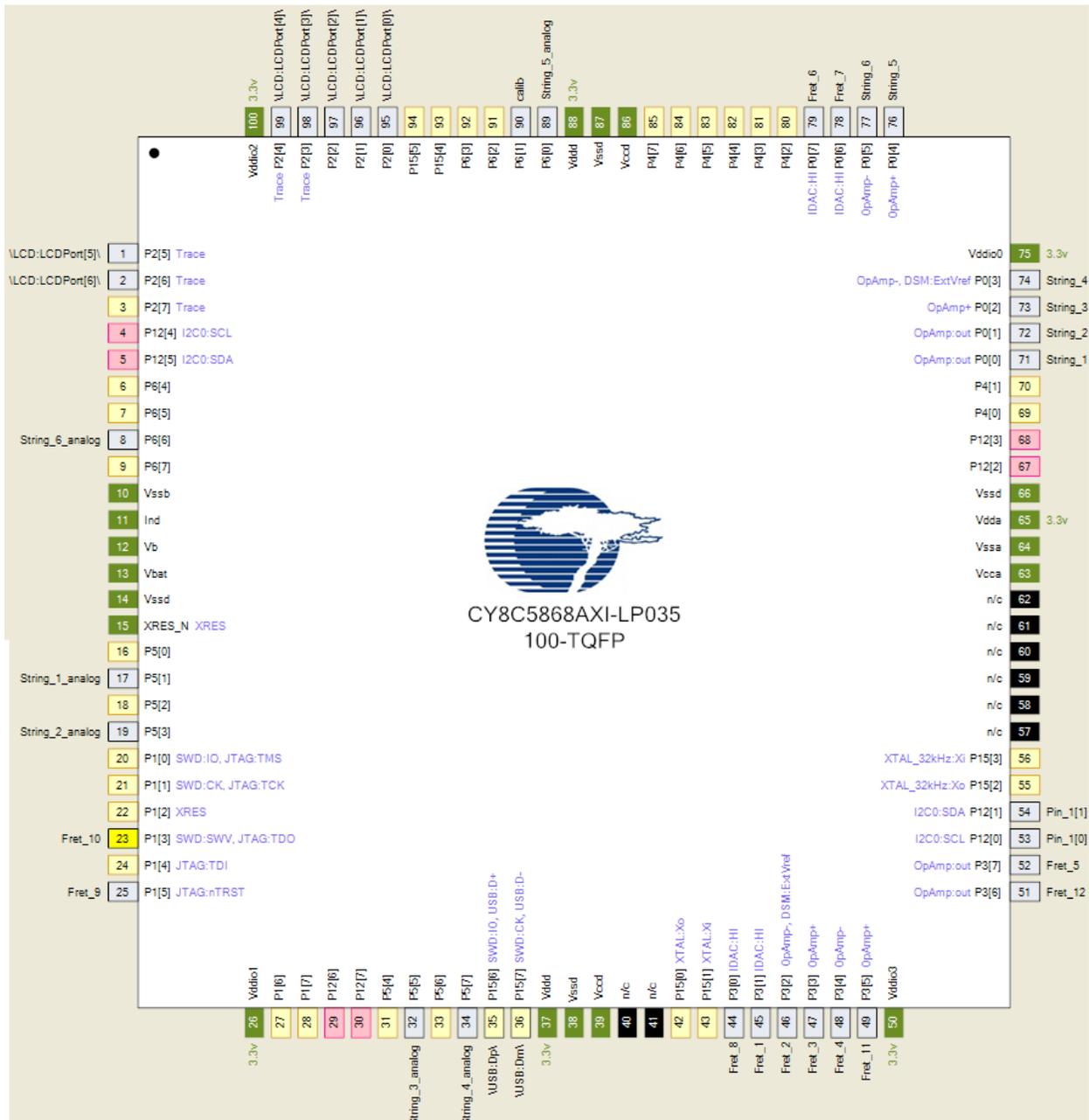


```
LCD_PrintInt8(display_note);
LCD_PrintString(": ");
LCD_PrintInt16(curr_stringmax(display_note-1));

if(USB_IsConfigurationChanged() != 0u) /* Host could send double SET_INTERFACE
request */
{
    if(USB_GetConfiguration() != 0u) /* Init IN endpoints when device
configured */
    {
        /* Enable the output endpoint */
        USB_MIDI_EP_Init();
    }
}

}
}
/* [] END OF FILE */
```

8.3 PSoC Pinout



Name	Port	Pin
\LCD:LCDPort[6:0]\	P2[6:0] Trace, Trace, Trace, Trace	95..99,1..2
\Pin_1[1:0]\	P12[1:0] I2C0:SDA, I2C0:SCL	53..54
\USB:Dm\	P15[7] SWD:CK, USB:D-	36
\USB:Dp\	P15[6] SWD:IO, USB:D+	35
calib	P6[1]	90
Fret_1	P3[1] IDAC:HI	45
Fret_2	P3[2] OpAmp-, DSM:ExtVref	46
Fret_3	P3[3] OpAmp+	47
Fret_4	P3[4] OpAmp-	48
Fret_5	P3[7] OpAmp:out	52
Fret_6	P0[7] IDAC:HI	79
Fret_7	P0[6] IDAC:HI	78
Fret_8	P3[0] IDAC:HI	44
Fret_9	P1[5] JTAG:nTRST	25
Fret_10	P1[3] SWD:SWV, JTAG:TDO	23
Fret_11	P3[5] OpAmp+	49
Fret_12	P3[6] OpAmp:out	51
String_1	P0[0] OpAmp:out	71
String_1_analog	P5[1]	17
String_2	P0[1] OpAmp:out	72
String_2_analog	P5[3]	19
String_3	P0[2] OpAmp+	73
String_3_analog	P5[5]	32
String_4	P0[3] OpAmp-, DSM:ExtVref	74
String_4_analog	P5[7]	34
String_5	P0[4] OpAmp+	76
String_5_analog	P6[0]	89
String_6	P0[5] OpAmp-	77
String_6_analog	P6[6]	8

8.4 Arduino Code

```

#include <Wire.h>
#define address 0x36 //PSoC I2C Address

void setup()
{
  Wire.begin();
  Serial.begin(57600); //57.6 KBaud
}

void loop()
{
  Wire.requestFrom(address, 3); //Read 3 Bytes from 0x36 = PSoC I2C Address
  byte string = Wire.read();
  byte fret = Wire.read();
  byte on = Wire.read();
  Serial.print("f"); //Send information over serial to PC
  Serial.println(fret);
  Serial.print("s");
  Serial.println(string);
  Serial.print("n");
  Serial.println(on);
  delay(5); //Wait 5ms to prevent buffer overflow on PC side
}

```

8.5 Python Code

```

import serial
import time
import msvcrt

ser = serial.Serial("COM7", 57600) #Serial Port on COM3, 57600 Baud

strings = ['0', 'E|', 'B|', 'G|', 'D|', 'A|', 'E|'] #the 0 to simplify zero-indexing
                                                #Each value in array is a line of

text
last_state = 0 #Previous states
last_fret = 0
last_string = 0
last_time = time.clock() #Time of the last note played
curr_time = time.clock() #Time of the current note played

while True:
    if len(strings[1]) > 165:

```

```

strings = ['0', 'E|', 'B|', 'G|', 'D|', 'A|', 'E|']
ser.flushInput() #Clear the Buffer to prevent frame shift errors

val1 = ser.readline() #3 serial bytes
val2 = ser.readline()
val3 = ser.readline()
for value in [val1, val2, val3]:
    if value[0] == 's': #S Indicates String Information
        curr_string = int(value[1])
    elif value[0] == 'f': #F indicates fret information
        if value[1] != "1":
            curr_fret = int(value[1])
        else:
            if len(value) > 4: #Deal with 2 digit fret numbers
                curr_fret = (10*int(value[1])) + int(value[2])
            else:
                curr_fret = int(value[1])

    elif value[0] == 'n': #If note is being turned on or not
(state)
        curr_state = int(value[1])
        curr_time = time.clock() #Record time of note turning on
    else:
        curr_state = 0
if curr_state == 1:
    if curr_fret > 9:
        appender = '---' #two digit fret, add extra dash to other strings
    else: appender = '--'

    if not ((curr_string == last_string) and (curr_fret == last_fret)): #add
either dashes or fret number
        print (curr_time - last_time)
        for i in range(1,7):

            if (curr_time - last_time) > 0.2: #Not a chord (0.2 seconds is
threshold for chord)
                if i == curr_string:
                    strings[i]+=str(curr_fret) + "-" #Add dashes or proper fret
name
                else: strings[i]+=appender
            else:
                if i == curr_string: #For a chord, add it at the same time
                    if curr_fret > 9:
                        strings[i] = strings[i][0:-3] #Take out last 3 characters
and replace them with fret information
                        strings[i] += str(curr_fret) + "-"
                    else:
                        strings[i] = strings[i][0:-2] #Only take out 2 characters
if it's a single digit fret
                        strings[i] += str(curr_fret) + "-"

                print strings[i]
                last_time = curr_time
                print '' #Add in an extra line for clarity

```

```
last_string = curr_string
last_fret = curr_fret

if msvcrt.kbhit(): #kit any key to stop
    break

ser.close() #make sure to close the serial port
```