6-1-2001

Fastab: Solving the Pitch to Notation Problem

Jeremy I. Robin
Dartmouth College

Fastab: Solving the Pitch-to-Notation Problem

Jeremy Robin
jir@alum.dartmouth.org
Senior Honors Thesis
Dartmouth College Computer Science
Advisor: Scot Drysdale

Technical Report #TR2001-406

1 June, 2001

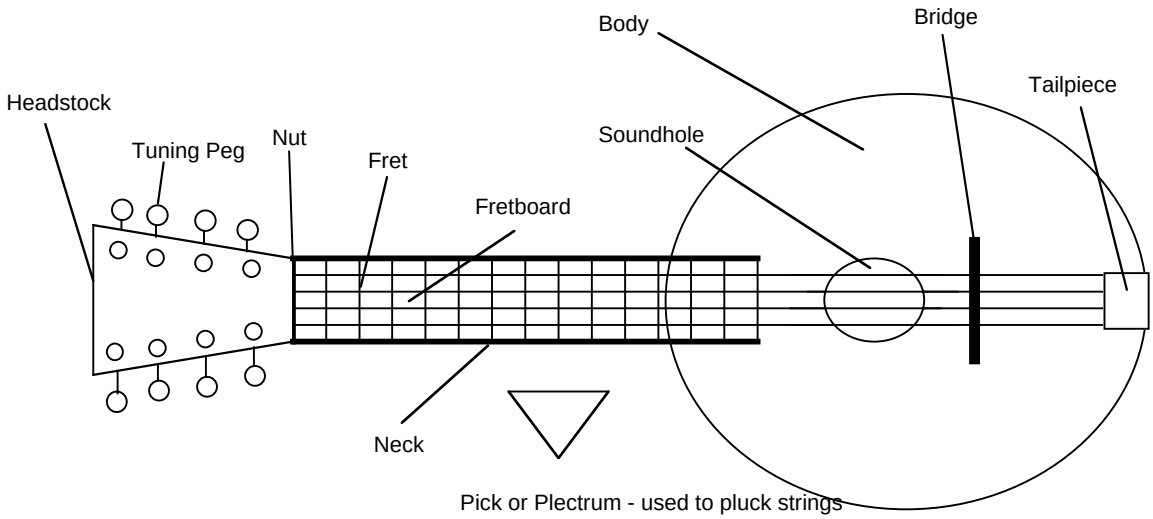
Abstract

I have always been frustrated with the length of time necessary to notate a piece of music. Computers have simplified so many other aspects of our lives, it seems that they should be able to simplify this task as well. In fact, there are already two distinct ways that engineers have attempted to attack this problem. The first analyzes the waveform generated by microphone input and relies on Fourier Analysis and other similar methods. The other examines the analog signal generated by a electric guitar-like pickup placed beneath the strings. The method used by Fastab relies much less on the musical properties of an instrument. Instead, Fastab records where and when the fingers and pick contact the instrument using digital electronics and microprocessor technology. Fastab provides a solution to the pitch to notation problem which is cheaper and more accurate than any other device available today.

Introduction

When I set out to design the system I now call Fastab, I had no idea it would become an honors thesis in computer science. I was a mandolin teacher enrolled in a computer music class at Dartmouth and short on time.

The Mandolin



My students were constantly asking for me to give them music for the tunes we were working on. I had learned most of these songs from other teachers by ear and had no sheet music to provide. I thought it would be easy to write out these simple songs for my students. But I failed to realize that it is very difficult to remember exactly how to play a tune without an instrument in hand. It took me what I considered to be way too much time to write out everything for them. I thought, "wouldn't it be nice if I could just play the tune into the computer and have it record and write it out for me?" Well, the answer is yes. It would be very nice. After doing a bit of research, I learned I could have this technology and a lot more for the price of $700.

The technology that I discovered is the MIDI (Musical Instrumental Digital Interface) pickup system. An electrically sensitive bridge is placed underneath all the strings, replacing the original wooden one. This bridge acts somewhat like an electric guitar pickup: coils of wire underneath each string generate a unique current depending on how fast the metal string oscillates in their magnetic field. These string signals are sent to a digital signal processor for conversion to MIDI. Then the MIDI can be sent to a computer or an amplifier. The MIDI messages generated by that system can be stored in a file to later be interpreted by a standard musical notational program. The major problem with this system is that in order to generate accurate notational data, there needs to be a software system in place to regulate the flow of MIDI messages. This pickup system is designed as a real-time converter. An amplifier plays the notes coming off the instrument in real-time but gives the user the capability to change the sound in literally thousands of ways. The MIDI messages are generated exactly as the player plays; ie, he plays a quarter note a little short and it is notated as an eighth note tied to a 16^{th} note tied

to a 32nd note (for general musical terminology see Appendix B). This is exactly what is needed for the real-time conversion described above but is unacceptable for notational purposes.

An additional problem is that this system is needlessly expensive if notation is all that the user wants. Notation is possible, but there are many other functions. Furthermore, there is no usable alternative between the MIDI pickup and nothing. The only other product that I've seen analyzes the microphone input (using Fourier analysis or filter bank algorithms). Ideally, this is the best solution because it is instrument independent, works equally well in acoustic and electric settings, and requires no modification of the instrument to operate correctly. Unfortunately, the current technology is incapable of supporting this solution. I have experimented with this system and found it to work poorly if single notes are played even at slow speeds and not at all for multiple notes.

Thus, I decided to embark on the journey that would ultimately culminate in my creation of a product called Fastab. It is very easy to use and accurate at medium to fast playing speeds. It does require a few small modifications to the instrument, but relies technology that is readily available and would be able to be produced and sold very easily for an affordable price. The purpose was not to create something that would capture everything perfectly. Instead, I wanted to create a device that would make writing down a piece of music on the guitar or mandolin as simple as playing it through once and then making a few minor alterations.

Design Decisions

Do I want to create something that can stand on it's own or do I want to incorporate the MIDI file protocol to utilize state of the art notational programs? Do I want to try to capture all the exact articulations of the player at the expense of rock-solid notation for less complex pieces? When laying out the design plan for Fastab, I needed to decide what problems I could realistically expect to solve in the given time and what features should be included in the system to make it as useful as possible to someone trying to write out music. I decided early on exactly what Fastab should be able to do and what it should not:

1. Instead of trying to capture someone's playing as accurately as possible all the time, the user should be given control over exactly how much accuracy is needed; if ninety-five percent of the notes in a given piece are quarter notes, there is no real need for the software to worry about detecting smaller note increments. That would only contribute to error.
2. Hammer-ons and pull-offs (see appendix A) are very common techniques on guitar and mandolin. They should be captured accurately.
3. Chords are essential to the sound of all stringed instruments. Chord detection and notation should be standard.
4. The cost of the system should be low.
5. Fastab should incorporate the MIDI protocol.
6. A metronome at a user-selected speed should be sounded whenever Fastab is recording to force the user to play notes of consistent length throughout each

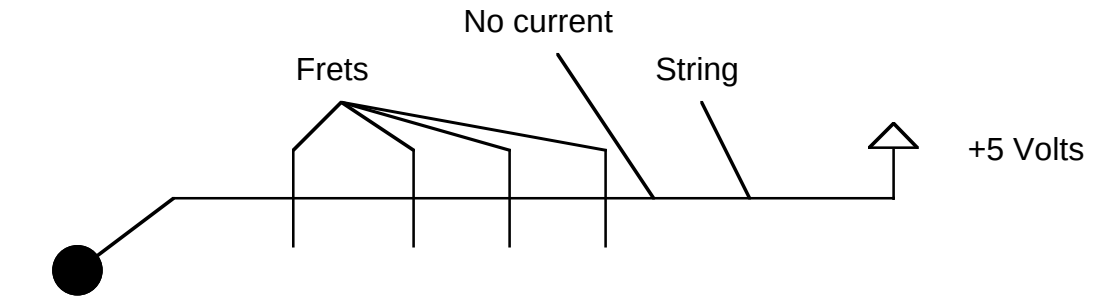
piece.
If there is no metronome, the program is forced to try to analyze notes within a small context instead of by some universal constant. A user will not play notes that they intend to have the same time value (eg quarter note, eighth note, etc,) consistently throughout a piece without a metronome. With a metronome, it is possible to simplify the time analysis and also increase accuracy.

The simplest definition of MIDI is "a series of messages which are designed to allow computers and synthesizers to communicate 'what sound to play' information." The alternative to writing the output of Fastab into a MIDI file was creating a gui to display, playback, and edit notes. The only drawback to this design is that instead of specifying the exact fingering (tablature) for a piece, I can only lay down the notes. The notational software then analyzes the notes and creates the tablature. However, after a little bit of testing, I came to the conclusion that the program (the freeware program TablEdit is the one that I used throughout the development of Fastab) correctly determined the fingerings almost 100 percent of the time. Furthermore, flaws can be quickly and easily corrected by the user. So instead of "reinventing the wheel" and creating my own protocol and program to store and edit musical data, I decided to learn to use the MIDI protocol. The time saved by this decision gave me much more time to focus on accurately capturing notes.

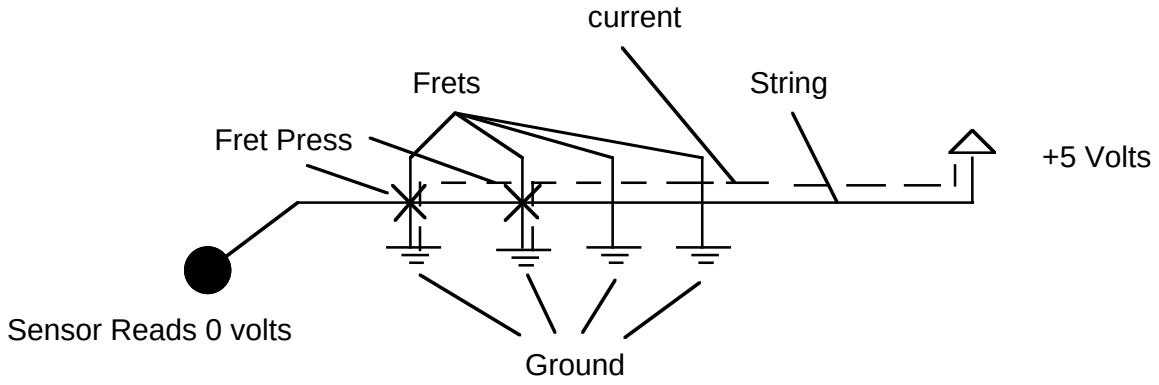
There are a few standard methods of attacking the pitch-to-note problem. One requires analyzing an electric signal, as the MIDI pickup does. Another converts microphone input to note values. A final way attempts to determine the notes from collecting information about the state of the instrument (ie, where the fingers are touching and the fretboard and where the pick is in contact with the strings). I immediately decided against trying to analyze microphone input. As was previously stated, trying to analyze the waveform of a chord is extremely difficult. This fact alone disqualifies that method for Fastab. Analyzing the electric signal with Fourier analysis or some other method I also decided against; that technology already exists in the MIDI pickup and has a high price tag. I chose to attempt to monitor the state of the instrument because this method provides the potential for greater accuracy with orders of magnitude less complexity and cost.

Conceptual Overview

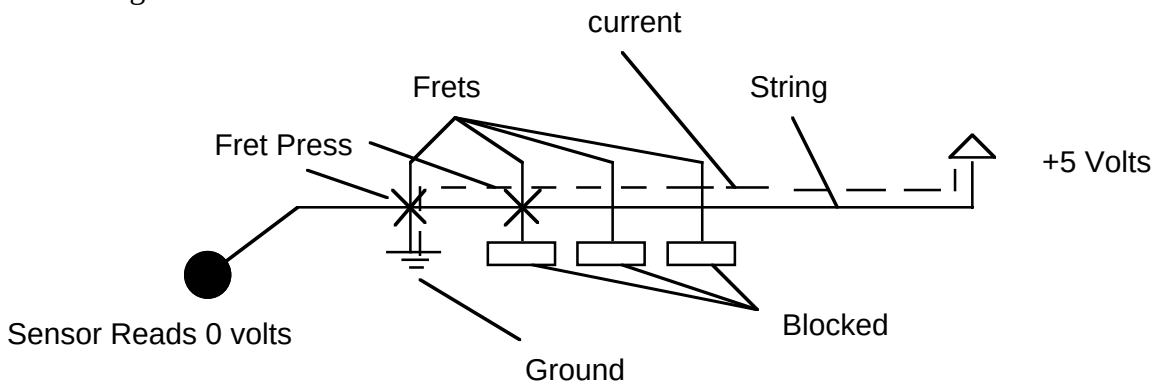
I chose to monitor the state of the instrument using a *digital* (0 or 1) approach. The hardware is designed so that each string is connected to a +5 volt source. Each fret is connected to a current sink, a *grounded* point (yes, there are wires soldered to my mandolin). For the final product, I have devised another solution to this problem (see Future Work and Unresolved Issues section). The electrons that comprise an electric current always want to travel from a high potential to a lower potential. *Ground* is lowest with a potential of 0 volts. This means that if a string comes in contact with a fret, electrons will travel from the +5 volts source directly to *ground*. This fact allows Fastab to monitor the instrument because sensors can be placed on each string. No current will flow unless there is a path from the +5 volts source to a lower potential.



Because the sensor does not provide that lower potential, the +5 volt source holds the string at +5v and there is no current. However, if a fret is connected to that source, ie, if a string touches a fret, the electrons have a path directly to ground. Thus, if the sensor detects no current on the string, no fret is pressed. If there is current then some fret is pressed. The pick is also connected to *ground.* This is perhaps the most annoying aspect of Fastab; the pick must have a wire attached to it. If the pick touches the string, there is current and the sensor detects it.



The way that the system can distinguish which fret is pressed is that it can turn on and off the *grounding* capabilities of each fret. If there is current on a particular string, it is because that string is touching a fret that has been enabled to drain current. So having this ability, the system can examine one fret at a time to determine whether or not a string is touching it.



To detect the *state* of the instrument, fret number one is enabled to drain current. All of

the others are blocked so that they can not drain current. So now if there is current on a particular string, it is because that string is touching fret one. Then we record the data specifying whether each string is in contact with fret one. The software then reconfigures the frets so that fret two can drain off current and none of the other frets can. The data concerning the state of the instrument is then recorded. This check happens for all the frets and then the pick. Then all of that data, the *state* of the instrument, is stored. Then the whole process is repeated again and the old and new *state* values are compared. If they are different, a message is sent to the computer containing the new *state* value. If the values are the same, the *state* is read again and the values are compared.

A circuit specially designed for Fastab handles all of the processing described above. This circuit guarantees that the software on the computer will see a message describing the *state* of the instrument every time it changes. So the task of the computer software is to take this series of *states* and convert them to note values. In order to accomplish this, the Fastab software divides the work into two parts. They can both be thought of as timelines. One records where and when the fret presses occur. The other is a record of when the pick strokes occurred. When a message is received saying that a fret is touching a string, it is recorded in the timeline as the start of a fret event. When a message is received saying that that fret is no longer touching, that event is recorded as the end of a fret event. The time each event starts and ends corresponds to the time when each message was received. When the user decides to stop recording, the Fastab software has to match up pick strokes and fret events. A pair matches if the ending time of the pick stroke is in between the start and ending time of the fret press. If there is a pick stroke on string 3 that ended at time 500 and a fret event on fret 5, string 3 that started at time 490 and ended at time 520, the two parts match. Fastab then records a MIDI note with a pitch value equal to D above middle-C, the 5^{th} fret of the 3^{rd} string.

The previously described model is insufficient for capturing hammer-ons and pull-offs. In order to detect these, a little bit of fine tuning is needed. So when should notes be generated due to hammer-ons and pull-offs?

1. If an open string is picked, any subsequent fret press on that string within a small amount of time should cause a note to be created.
2. If a fretted string is picked, any change in the highest fret pressed before another pick stroke should cause a note to be created.
3. If a fretted string is picked, if that string subsequently becomes open (ie, there are no frets pressed) and if no other pick strokes have been recorded, a note should be created on the open string.

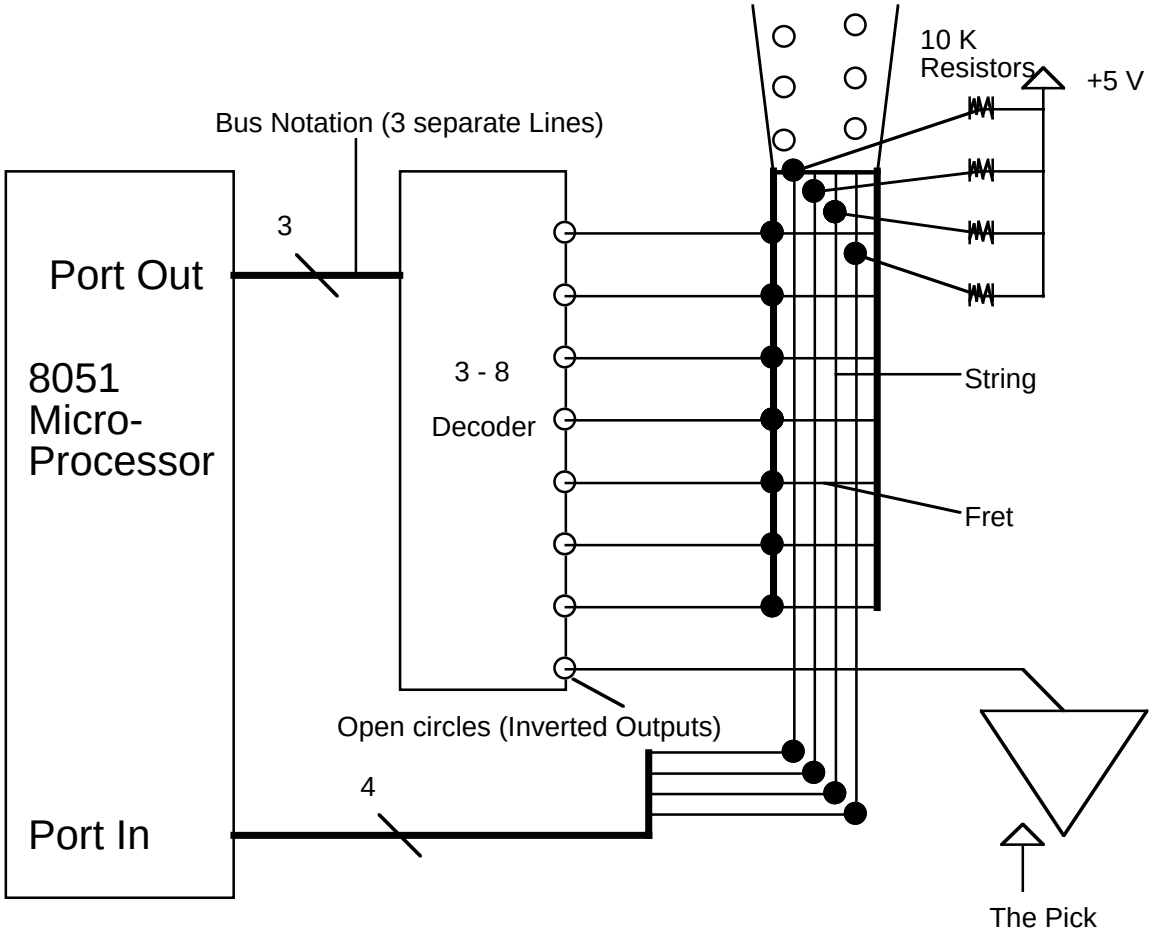
Turning these specific series of events into notes is a little more complicated than the original model provides for and will be discussed in more detail below.

Hardware

A +5 volt source is connected to all of the strings. At the other end, a sensor monitors each string. In between the source and the sensor, 7 frets and the pick are each connected to ground. Thus, if the string touches a fret, the current is drained off and the sensor reads a very low potential (around .08 volts). The frets are grounded one at a time so the exact hand placement can be determined. So if string 3 is touching fret 5 only and

all other strings are untouched, when frets one through four and six through eight are in turn grounded, the sensor will read +5 volts for all strings. However, when fret 5 is grounded, the sensor will read +5 volts for all strings but the third. There, a +.08 volts will be registered. The *state* data described above can be condensed for storage into only 32-bits. For each fret we designate 4 bits; one for each string. A value of 0 (0 volts) means that there was a string press. A value of 1 (+5 volts) means there was no string press.

Here is the circuit diagram (minus one small modification, see below):



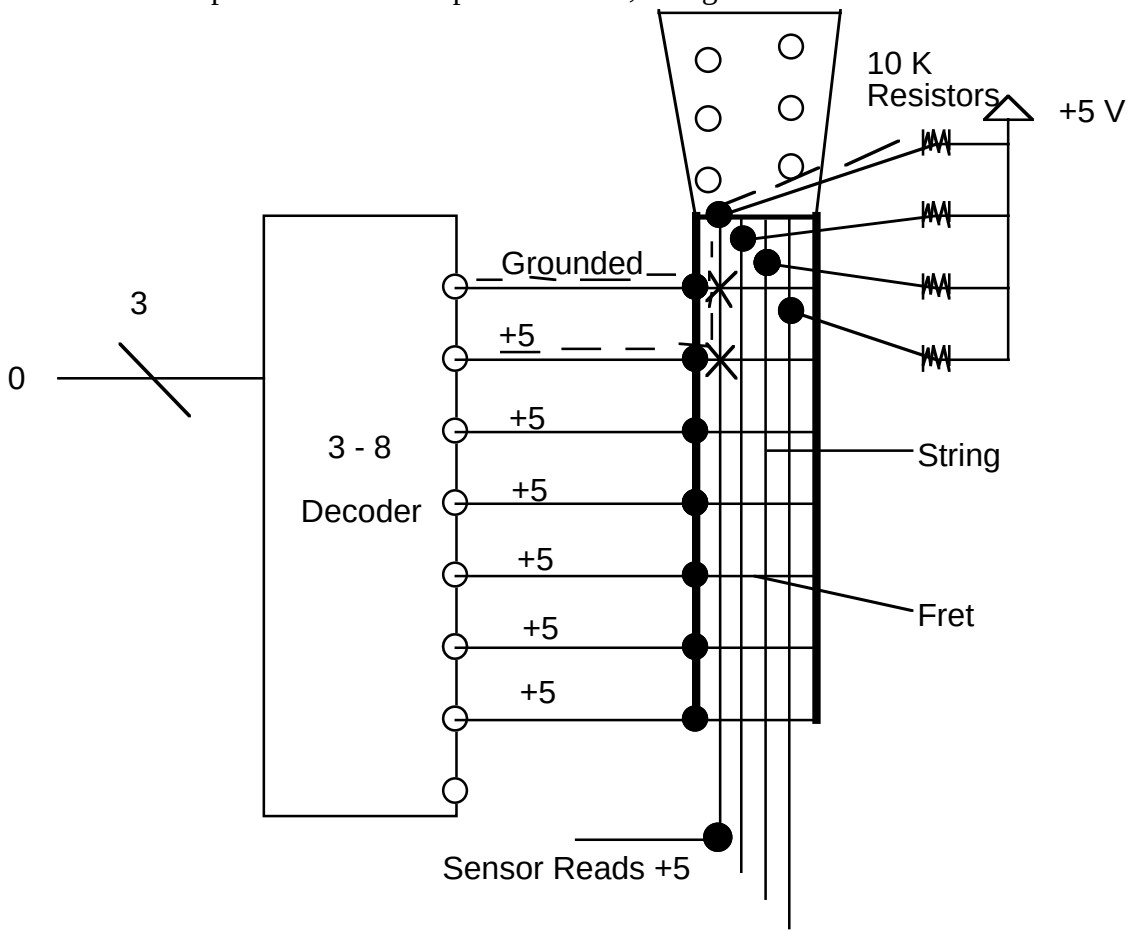
The reality is that one fret can not really be connected to ground by itself. If the user presses the fingerboard at fret 5, that really means he presses in between frets 4 and 5, creating two connections; and because the bridge end of the instrument is physically so much higher, electrical connections can be made at frets one, two, and three also. On the software side, we only care about the highest fret pressed. But on the hardware side, the instrument needs to be monitored as carefully as possible. The theory behind my electrical design is the same as that used in matrix keypad implementation. A requirement of this design is that each string needs to be insulated from all others. As a result, I have placed leather grommets in between the tailpiece of my mandolin and the strings and electrically separated the tuning pegs from one another. In the future this could be easily accomplished with a specially designed string having ends wrapped with

cloth or another non-conductive material.

Selecting one fret to be *grounded* is accomplished by way of a 3-8 decoder with inverted outputs. A 3-8 decoder takes in a 3 bit binary number, 0 to 7, and then outputs +5 volts to the specified output pin (0-7) and 0 volts to the others. With inverted outputs, the specified output pin is grounded and all the others are set at +5 volts. One fret at a time is selected by writing 0 to 7 binary into the decoder. This specification for a chip works perfectly with the model described above for detecting where the string and pick come in contact with the instrument.

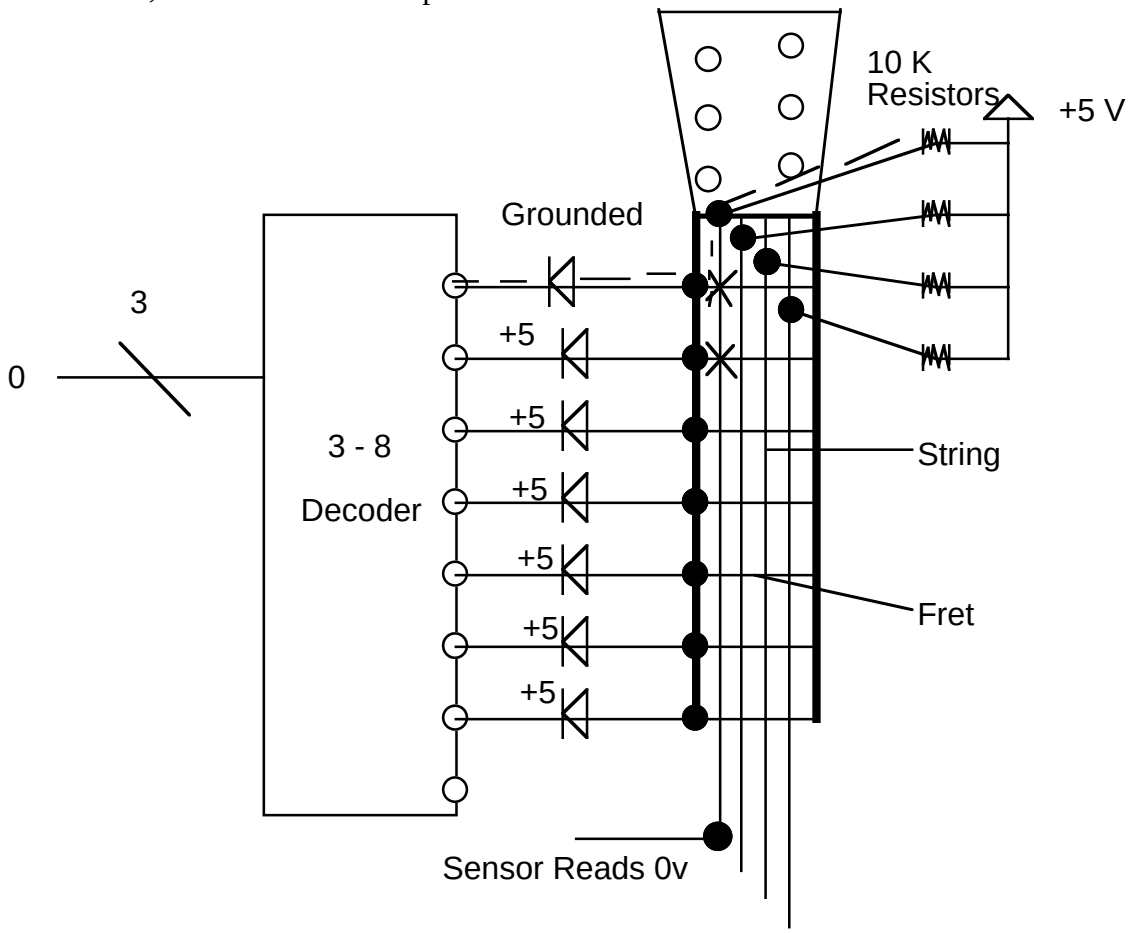
A problem with this initial design is that when the user presses a fret, he actually creates two connections.

Here is an example where the user presses Fret 2, string 4:



With nothing between the decoder and the frets, the +5 volts from the decoder would be placed on the same line as the continuous +5 volt signal. Two +5 volt signals on the same line proved to be too much for the decoder input to ground. As a result, the sensor reads a +5v signal even when the fret press indicates that 0v should have appeared. To combat this problem I placed diodes in between the decoder chip and the fret. A diode is an electrical device that allows current to flow through it only one way. I inserted them so that current could flow only into the decoder and not away from it. This prevents the decoder's +5 volts from getting to the fret. We might also place a 10k resistor in the

decoder output path. However, this would diminish the decoder's ability to ground the frets. Thus, diodes are a better option.



This circuit works perfectly well for monitoring the instrument, but how do we get it to go fast enough to be useful? If we can only cycle through the 8 outputs of the decoder once a second, it is impossible to create a realistic timeline of fret presses and pick strokes. A software or hardware design to quickly cycle through all frets and record the *state data* is needed. My first attempt at Fastab utilized the parallel port for communication with the circuit described above. I had no microprocessor, so all the data collection needed to pass through the i/o port. In order process one 32-bit *state cycle*, it was necessary to select each fret and then read its value: The computer (1) sent a message out the port specifying the fret and then (2) read back through the port the electrical pattern on the strings. This means that the parallel port served as both the sensor for the strings and the selector inputs for the decoder. This is inefficient for a number of reasons. For one, it is ridiculous that the host computer should have to send a separate message every time it wants to examine a new fret. Why not just set a 3-bit counter going and read the 32 bit *state* every time it resets? Once I learned a little more about circuits, I realized that I could have a counter iterate through the 8 different grounding modes and then send the 32-bit value all at once. This is considerably more efficient but much more complicated because storage devices (latches, flip-flops, etc.) are needed to remember the 4 bit state values for each fret.

It is also an inefficient use of i/o to send redundant data. In other words, why send data that does not effect the output of Fastab? Once the data collection speed reaches about 100 cycles per second, there is no reason to send every 32-bit value. Many are exactly the same. The current Fastab implementation cycles about 8000 times per second, and there can be a max of 100 *state* changes per second (that is at least fifty percent higher than I have ever observed). If we create a storage element for the previous *state*, we can compare the previous and current *states* of the instrument and only send a message when the data changes. This would double the space requirements, to 8 bytes, while reducing the data transmission by at least 8000 percent. This small change drastically increased the efficiency of Fastab's data collection scheme.

I was fairly happy with the results of this first implementation. However, because Fastab is meant to be a salable product, I realized that using the parallel port was impossible. The parallel port has become primarily a printer port, and most potential customers would be unwilling to plug and unplug their printer every time they wanted to notate a bit of music. So I decided that the design needed to incorporate serial communication.

Unfortunately, serial communications comes with additional overhead. The parallel port is meant for very short range data communication. Thus, there is no need for a special protocol to diminish bit error. This error is most often caused by signal deterioration. Because the distance is so short, there is almost no danger in a +4.5V signal being mistaken for a +.08V signal, so TTL logic levels (0 to +5V) can be used on the parallel port with no problem. However, the serial port is designed for greater reliability over distance. So for RS-232 (the electrical specification for serial communications) compatibility with the host PC (the computer connected to the circuit), it is necessary to convert voltages to +12V and -12V before sending them. This makes it much less likely that a 1 can be mistaken for a 0 or vice-versa. In addition, because only one bit can be on the line at a time, the flow of data needs to be regulated to ensure accurate transfer; ie, the host computer can not just read 8 separate lines to receive a byte. Utilizing a microprocessor for serial communications simplifies things immensely. There is a specially designated buffer built into the Intel 8051 microprocessor, the SBUF. In order to send a byte serially from the 8051, it is necessary only to place the byte in the SBUF. An internal flag is set whenever all data in the SBUF has been transferred so that a new byte can be sent.

Another benefit of using the 8051 microcontroller is that it has built in data storage. The 8051 can do the 8 fret iterations and then store that *state* in internal RAM. Using the 8051 for this data storage alone reduces the amount of hardware (and also the probability that it will not work) by around sixty percent. A program of about100 lines of Intel 8051 assembly code controls the selecting and recording the instrument *state*. One output port is connected to the decoder inputs, and an input port (the sensors) is connected to each string. The control flow of the microprocessor code is as follows:

-a fret is selected.

-the binary string pattern is read into the input port.

-the value is stored in a dedicated address in memory.

-when all 8 strings have been read, the value is compared to the previously stored values.

-if the new 32 bit value is different, it is sent out the serial port and replaces the

old value in memory.
-if the 32 bit value is the same, nothing happens and the strings are read again.

My first implementation incremented a counter when the same value was recorded and sent a count value along with the string pattern. This design proved to be faulty; as more notes were played, the count for each became less because of the extra processing time involved in sending the data out the serial port - whole notes would be stored at 4000 cycles, half notes would only count 1700 cycles. In the end, it was more efficient (less data transfer) and also more reliable to use the host PC's system clock. Messages are time-stamped and stored for later analysis as they arrive.

Processing Microcontroller Messages

I think the best way to describe how the 32-bit *state* messages are processed by the host PC is to describe, bottom up, how the classes of Fastab are structured. The purpose is to construct something that resembles a timeline to store the fret presses and pick strokes. So if we can create a data structure to represent fret presses and pick strokes in the timeline, that is a good start. We call a recorded fret press a *FretEvent.* A recorded pick stroke is called a *PickEvent.* Because timing is so crucial to this application, it should be made clear from the start that there is a Windows function called GetTickCount() which Fastab uses to generate the system time. When *state* messages are received from the external circuit, they are stamped with a time returned by this function. This function can generate around 1000 evenly spaced clicks per second and is therefore sufficiently accurate for my purposes.

Associated with the *FretEvent* class are a few variables:
-m_sysTime (starting time for event)
-m_duration (how long event lasts)
-m_string (with what string the event is associated)
-m_fret (which fret has been pressed)
-two other variables, m_enabled and m_used (used to detect hammer -ons and pull-offs and will be discussed later)

The variables pertaining to a *PickEvent* are exactly the same except there is no storage of a fret. These two classes are, in fact, both derived from a superclass, *Event*, so that that pointers to both types of events can be held in the same data structure.

The transition from *state message* to e*vents* is not a quick one to one conversion. There are three primary difficulties in going from one to the other:

1. Because the pick is moving across the strings, the electrical connection is constantly being made and broken. Each pick stroke can generate as many as 5 or 6 *state* change messages. (pick touching string, no touch, pick touching string, no touch, etc...)
2. Because the string vibrates when it is plucked, there is a lot of incidental string and fret contact above the highest fret that the user is pressing. There can also be interruptions even in the intended fret's contact with the string.
3. All sorts of messages can be generated as a result of *state* changes behind the highest fret press. These do not effect the pitch being played and should

therefore never create *fretEvents.*

The *eltList* class is used to store these *Events* and do much of the error checking necessary to handle the three problems described above. The *events* are stored in a linked list data structure. The member data for an *eltList* is just a current pointer and a sentinel pointer. It is a circularly linked list with a sentinel with a few conditions allowing or prohibiting *events* from being inserted. There are five important functions that this class provides. The most obvious are the two insert functions. Insert() is overloaded so that one takes a pickEvent and the other takes a fretEvent. The two different types of events are never mixed in the same list, but dealt with by the same data structure. The find() function takes a pickEvent and seaches its list of fretEvents (timeline) to find a match. The function first() returns the first element in the list. Next() takes an element and returns the next element or 0 if the given element is last in the list.

In simplest terms, a note is created when a pickEvent occurs. There are two separate timelines for each string represented by two eltLists of *Events:* one for the fretEvents and one for pickEvents. When the user has finished recording, the Fastab software attempts to match up all the recorded *PickEvents* with corresponding *FretEvents.* This is accomplished by calling the first() and next functions to return successive pickEvents and then the find() function to match them up with fretEvents. If a matching fretEvent is found, then a note on the picked string at that fret is created. If no fretEvent is found, an open string note is recorded.

For the reasons enumerated above, converting *state* messages directly into *Events* will generate ridiculous results. We need some way of distinguishing when *state messages* reference the same pick stroke or fret press and we also need a way to discard *state messages* that reference nothing other than inadvertent electrical connections. To overcome these difficulties, I conceptualized the "one-back" model. In this model, we do not believe that *events* are legitimate until they are followed by another event. Here is a typical series of processes to deal with a pickEvent:

1. A message is received at time 1000 stating that the pick is touching string 3. This data is stored in temporary variables until another *state message* lets the software know that the pick is no longer touching.
2. A message is received at time 1020 stating that the pick is no longer touching string 3.
3. A pickEvent (m_string = 3, m_sysTime = 1000, m_duration = 20) is stored in a the eltList corresponding to string 3.
4. A message is received at time 1030 stating that the pick is touching string 3.
5. A message is received at time 1040 stating that the pick is no longer touching string 3.
6. The previous pickEvents m_duration variable is reset to 40 because the two events are so close in time that they can be assumed to be the same event interrupted.
7. A message is received at time 2050 stating that the pick is touching string 3.
8. A message is received at time 2100 stating that the pick is not touching string 3.
9. A new pickEvent (m_sysTime = 2050, m_duration = 50) is inserted. We now believe that the previous one is legitimate. New events can no longer affect the older one.

If we did not combine *events,* the two *state* messages would have generated two

pickEvents. Here is a case where we actually discard a state message:

-A message is received at time 1030 stating that fret 3 is touching string 3.
-A message is received at time 1035 stating that fret 3 is no longer touching string 3.
-A fretEvent is inserted into the list.
-A message is received at time 2030 stating that fret 3 is touching string 3.
-A message is received at time 2090 stating that fret 3 is no longer touching string 3. A pickEvent is inserted into the list. The previous event is discarded because it is too short to be its own event and too far away on the timeline to be part of the new event.

FretEvents are inserted into their lists the same way as pickEvents, except instead of checking whether or not the pick is touching, we perform a check to determine the highest fret press. *FretEvents* can be combined the same way provided that the m_fret variablse are the same. The logic in the two parts of the program is very simlar. The constants specifying how close *FretEvents* need to be to one another and the minimum time for a legitimate fretEvent are different from those pertaining to *PickEvent.*

When a piece is played which does not includes hammer-ons or pull-offs, the above algorithm is enough. However, in these two special cases, fret presses alone need to generate their own notes. The e*vent* class' member data m_enabled and m_used enumerated above but not described are for use in detecting these more sophisticated instrumental techniques. A fretEvent is m_enabled (= 1) if it can cause a note to be created on its own. The default is that fretEvents can not cause notes to be created (m_enabled = 0). A fretEvent is m_used (= 1) if it has been matched with a pickEvent. When fretEvents are first created, they are set to unused (m_used = 0) except in one case discussed below. When a player hammers-on, he picks a note, and then puts a finger down on that same string to create a second, higher tone. When he pulls-off, he picks a note and then removes his finger to sound a second, lower tone.

The model we use to detect hammer-ons and pull-offs is to enable all *fretEvents* which match a pickEvent. We also insert bogus *FretEvents* with m_fret = 0, m_enabled = m_used = 1 when an open string note is created. This allows fastab easy access within each fret eltList to the times of the open string notes. Because hammer-ons can occur after open notes as well as fretted notes, having enabled *fretEvents* in the list corresponding to open string notes allows us to keep our model consistent for all cases. This enabled and used *fretEvent* corresponds to the initial pick stroke. Then after the recording has finished, we sweep through the fret eltLists to find out whether there are other fretEvents in close enough proximity to these enabled events to be considered as a hammer-on or pull-off. The m_used variable indicates that a m_enabled fretEvent has been played with a pick stroke already and should therefore not create another, identical note. When a fretEvent is enabled, that means it has the potential to create a note without being found by a pickEvent. What specific conditions result in a fretEvent being m_enabled?

1. If it is matched to a pickEvent.
2. If it starts within a small amount of time, FRETENABLETIME in my code, of the termination of a previous fretEvent which was enabled.
3. If it occurs within OPENENABLETIME of a pick stroke on an open string.

These specifications for enabling strings directly mimic the conditions in which a

musician hammers-on or pulls-off. If he picks an open string, any fret pressed on that string within a certain amount of time should sound as a note also. If he hammers-on from a fretted note, there will be only a very small amount of time separating the fretEvent that matches a pick stroke and the hammered-on note. The same is true of pulled-off notes. The important concept to remember here is that all this is accomplished in post-processing. Only the matched *fretEvents* are actually enabled during the recording session.

The next hierarchical class is called sixqueue (named for the six fret queues necessary to store guitar fret information). This class can be regarded as the organizer for the classes described above. All the timelines (linked lists) and generated notes are stored in this class. Each string's eltList is created and named fretQueue[i]; eltList's are also created to store pickEvents. There are NUMSTRINGS (a global variable which holds the number of strings of the instrument we are analyzing) pickQueue[i]s. When a new message arrives from the microcontroller, it is examined by the sixqueue function analyze(). Analyze works on the one-back model. It stores *states* in temporary variables and inserts them after they are finished (ie, the pick is no longer touching the strings or a fret is no longer pressed). Analyze() does the following for each string:

1. Finds the highest fret pressed on the current string by checking the bit patterns. That is the only one we care about, because the others can have no effect on the sound.
    1. If there is a fret press being held in temporary variables and the new highest fret is not the same or if there is no current fret press, a fretEvent is created from the old temporary variables.
    2. If there is not a fret press being held in temporary variables or if the new highest fret is the same as the one stored in temporary variables (just because a message was received does not necessarily mean that any one string has changed its *state*), skips to (2).
2. If there is no current fret press, does nothing.
3. If there is a current fret press, inserts the system time and the fret into the temporary variables.
4. Examines the bit pattern on the current string to determine whether there is a pick stroke.
5. If there is a pick stroke being held in temporary variables and there is currently a pick stroke, or if there is not an old pick stroke and there is not a current pick stroke, examines next string.
6. If there is a pick stroke being held in temporary variables and there is currently not a pick stroke, creates a pickEvent and inserts it based on the temporary variables. Then examines next string.
7. If there is not a pick stroke stored in temporary variables and there is a current pick stroke, stores the time in the temporary variables before examining next string.

Because messages are sent from the microcontroller every time anything changes, the fact that a message has been received does not mean that any specific pending event will be terminated.

Converting to Notation

After the user has stopped recording and all the hammer-on and pull-off post processing has terminated, Fastab's pertinent data is a linked list of notes. Each note has associated with it a pitch value, an absolute time assigned from the starting time of the *pickEvent* which created it, and a duration determined by the length of a corresponding fret press. There are only 3 steps left in the process: (1) assign time values to the notes (eg, quarter note, eighth note) (2) group chords together (3) convert to MIDI for writing to a file. The class *noteList* handles all three of these tasks with the aid of two storage classes.

The most basic note data class is called note. It contains member data m_sysTime, m_value (to hold the MIDI value of the note, eg A440 = 50), m_length (to hold the notated length, eg quarter note = 4), m_string (to hold the string in case a need to write out tablature from inside Fastab ever arises), and m_duration which holds the duration of the note in ticks (from the function GetTickCount()), as assigned by the eltList functions which create notes. The second data structure is called noteVector. This is a standard vector class that represents not individual notes, but chords. It holds a variable sized array of note values (int *m_notes) for a chord, a sysTime for the chord, a duration in cycles, a notated length, and a variable m_size which holds the number of notes in the chord. This class provides basic vector class functionality.

The class that organizes this note data is called noteList. It contains a doubly linked list of notes, a variable m_noteCount to hold the number of notes, and an array of noteVectors of size m_noteCount (to be initialized after the initial linked list has become static when the recording is finished). The important functions are cycToLength(), which turns a number of cycles into a notated length, giveLength(), which takes the linked list of notes, processes them, and creates the array of notevectors, and then the two midi functions writeMIDI(), which takes care of all the midi housekeeping, and writeMIDINotes(), which converts and writes the array of notevectors into the .mid file.

When control is transferred to the class noteList, there is a long list of noteEvents waiting to be processed. We sort them by sysTime to make sure that they are in order. According to the purpose of Fastab, these notes must be processed in such a way that makes editing them to form a cohesive song as easy as possible for the user. Based upon a couple hours of testing with a few different musicians, it seems that allowing each chord or note to ring until the next one is played is good enough in 95 percent of cases. With regards to processing, this means that the m_duration variable we have tried so hard to maintain is absolutely useless. Time values are calculated only based on the differences in starting times. When that sort of basic processing is not sufficient, it takes the user only a moment to insert a rest. The reason that this is preferable to actually analyzing the recorded durations is a bit subtle. It is also why the MIDI software on the market today, even at the price of $700, is insufficient. The designers of that software were more concerned with capturing the playing accurately (as they should be for the real-time sound conversion described above) than with accurately interpreting the notes within the context of strict notational rules. That system was not designed for notation; its notational functionality is a by-product of another goal. If we examine the actual duration of a played note, it will almost never be what, notationally, the musician intended it to be. A note is usually played for around 75 percent of its allotted time. It is

just not possible to play 4 quarter notes which each sound for exactly one-fourth of a four-four measure. So by assuming that the user plays no rests and dividing up the measure by the beginnings of notes, Fastab does a very effective job of predicting the intended lengths.

Taking the above ideas as guidelines, I designed the functions cycToLength() and giveLength(). These two functions use three global variables set by the user to do their processing. BEATSPERMINUTE specifies how fast the metronome beats. A metronome is needed so that the number of cycles corresponding to each note length stays constant through the duration of the recording. TDIVPERMEASURE lets Fastab know how "deep" the user wants it to look. If the user specifies four time divisions per measure (assuming BEATSPERMEASURE is set to the standard four), it is possible to record only quarter notes. If TDIVPERMEASURE is set to 8, eighth notes are examined, with 16, sixteenth notes..... CYCLESPERDIVISION is calculated from averaging the time difference in metronome clicks throughout the entire recording, multiplying by BEATSPERMEASURE and then dividing by TDIVPEARMEASURE.

After sorting the notes, Fastab rounds each one to the nearest time division. A chord is any group of notes which round to the same value. Up to NUMSTRINGS notes can go in one chord. Each set of notes at a certain time is given a noteVector. The duration assigned to each noteVector corresponds to the distance between its start time and the start time of the next noteVector. This value is taken into the function cycToLength(). The function determines how many time divisions correspond to that value and returns it. The array of noteVectors is now ready to be converted to MIDI (see Appendix A for MIDI details).

See Appendix D for complete C++ and assembly code for Fastab.

Windows

Because Fastab is meant to be a commercial project, it was essential that the system be usable on the Windows platform. I chose to do all the development in C++ using Windows Visual C++ to compile my code. With Visual C++, I have access to all of the complicated Win32 architecture with all of the difficult system calls and windowing abstracted away by the MFC. The MFC basically turns all the windows, dialog boxes, and menus of a complicated Windows program into a bunch of C++ objects. A visual editor makes specifying everything about a system's GUI as simple as pointing and clicking the mouse. These visually created windows are accessed through the MFC's message mapping system, where button presses, menu clicks, etc... trigger calls to functions.

Unfortunately, the MFC cannot do everything. I realized this when I decided I wanted to sound a metronome during the recording of the notes. This was necessary because there is no other way to assure that the user keeps the time value of his notes constant throughout the piece. When all but the best musicians play without a metronome, they, over the course of a piece, unintentionally change notes' time values. If I chose not to sound a metronome, a large part of the software would have to analyze each note in context and try to make a best guess at each intended note value. With the

metronome, even if individual notes are sometimes a little bit off, the bar lines stay very constant for the entire duration. This makes analyzing the signals much simpler.

In order to sound this metronome, first it was necessary to find the Windows system call that plays a .wav file. The default Windows call to PlaySound creates a non-overlapped sound event. This is a function call that does not terminate until the sound is finished playing. Obviously this is very inefficient. When a half second sound plays and stalls the rest of the program, a considerable amount of processing power is wasted. Fortunately, there is a parameter, SND_ASYNC that can be passed to the function to force the call to terminate immediately.

The serial port i/o also posed a difficult problem. The CreateFile() function call creates a connection to the serial port and returns a handle that allows a program to read and write to a buffer. This function, like most other Windows system calls, defaults to non-overlapped mode. Non-overlapped is a synonym for synchronous; the function does not return until it has completely its operation. Unfortunately, there is a lot of other stuff to take care of while the serial port waits on the next 32-bit pattern. Even though the interrupt function which handles playing the metronome sound can interrupt anything (even a while(1) loop), the GUI will go dead if ReadFile() is called in non-overlapped mode. There are two solutions to this problem. The first seems to be simpler. Because the port is simply waiting for a certain number of bits to arrive, it should have been easy to specify an interrupt function to call when the buffer fills up. There is a function OnCompletion() which, according to all the Windows documentation, should be called when the serial port is opened in overlapped (asynchronous) mode. Predictably, I could not get this to work. There was small mention in the documentation that perhaps overlapped mode works only on NT, but because it seems like such essential functionality to have, I kept trying. In the end, I gave up because a better solution presented itself.

Using Visual C++, if multithreading is enabled in the project settings, a function called CreateThread can be used to create multiple threads. So, in the end, the GUI and metronome are contained in one thread and the serial port and message processing are contained in another. State variables for each side are maintained to prevent the two from inadvertently corrupting each other's data.

User's Manual

Note – this user's describes the Fastab product in its final form, not as it is implements now.

Before you do any thing else, make sure your computer is running Windows 98 or 2000. The Fastab software will not work with NT, Windows 95, Linux, Unix, or Macintosh. To set your instrument up for use with Fastab do the following:

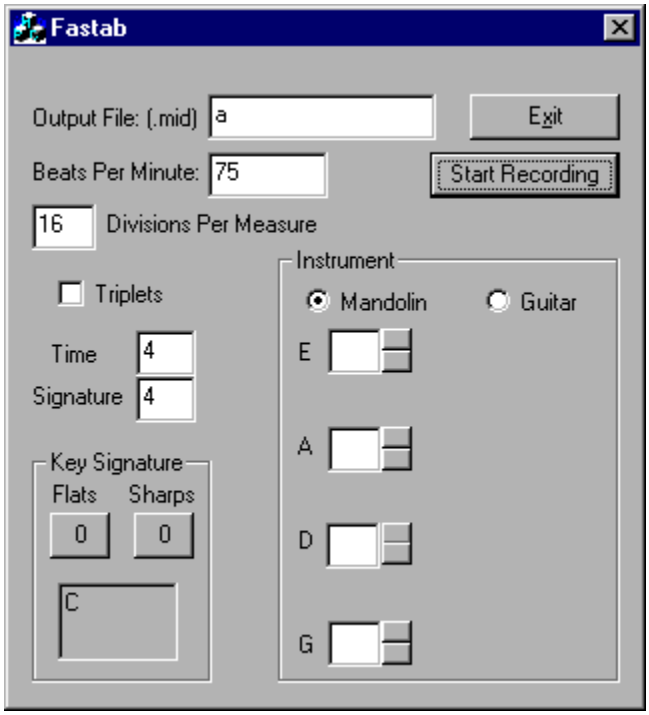
1. Install strings provided in the Fastab Installation Pack.
2. Attach provided Fastab Strip along the edge of the frets at the low G-string side of the fretboard as shown in Appendix C – Parts List. Make sure that the metal contact tabs are touching the frets.
3. Clip Fastab Adapter to headstock.

4. Plug chord from Fastab Strip into the Fastab Adapter.
5. Install 9V battery in the Fastab Adapter.
6. Plug hexagonal cord into the Fastab Adapter and the serial port in the back of computer. The Hexagonal Fastab cord has 9 pins and will only fit into one of the slots on the back of your PC.
7. Turn Fastab Adapter on. A red light will indicate that the unit is functioning correctly.
8. Attach Fastab Pick's cord to Fastab Adapter. You must play with this pick and plug it in for Fastab to function properly.

To start recording with Fastab, you will need a notational software program. We at Fastab recommend TablEdit, available for free download at www.TablEdit.com, though any software able to read MIDI files will work just as well. This program will allow you to view and edit the songs generated by Fastab.

Now you are ready to start Tabbing! Create a folder called Fastab where you would like to store your MIDI files. Put the Fastab.exe program in this directory. Make sure that the click.wav file is in the same folder or you will hear an annoying system beep instead of a metronome click. All the MIDI files generated by Fastab will now go in this folder.

Double clicking on the Fastab.exe icon will start the program. A dialog box that looks like this will appear on your screen.



(1) The "Output file: (.mid)" text insertion box allows you to specify the name of the MIDI file you would like to output. There is no need to add the .mid filename

extension. It will be added for you after the program is run.

(2) The "Beats Per Minute" box allows you to enter the metronome speed you would like to play at.

(3) The "Divisions Per Measure" box specifies the smallest note value for Fastab to search. 8 in the box means 8^{th} notes, 16 means 16^{th} notes, etc. Be careful: do not enter a higher number than you need. It will decrease the accuracy of your results.

(4) The Triplets box can be checked if you are playing triplets in your piece. Caution: do not check this box if you do not intend to play triplets in your piece. It will decrease the accuracy of your results.

(5) The two Time Signature boxes allow you to enter the time signature of the piece.

(6) The Key Signature boxes are for selecting the key signature. The lower box shows the current key. Hitting the Flats button will increase the number of Flats or decrease the number of sharps. Hitting the Sharps button will do the opposite.

(7) In the Instrument section, you select the instrument to which Fastab is attached. You also must specify a tuning change by hitting the buttons corresponding to each string if you are not playing the standard E-A-D-G mandolin tuning or E-B-G-D-A-E tuning of a guitar.

Hit the button "Start Recording" to start recording. You should hear the metronome beating at the specified speed. Hit "Stop Recording" to cease recording. You can record as many pieces as you want without restarting the Fastab program but be sure to change the output file name because your pieces will be overwritten.

Each run of the Fastab program will generate a MIDI file which can opened, viewed, and edited by your notational program. For documentation, refer to the company's web site.

Any questions can be mailed to jir@dartmouth.alum.org.

Future Work and Unresolved Issues

The assembly code described in the hardware section above is not actually burned into the program ROM of a microprocessor; and the circuit that accompanies that code is not stand-alone. The Fastab circuit should be powered by a battery and the microprocessor should begin the *state* data collection loop immediately upon power up. Instead, the code burned into the microprocessor is called the M/DP (Monitor/Debugging Program), written by Doug Fraser. This code allows the user to load 8051 assembly code into RAM after power on using the serial line. The problem is that the code is lost when the power is shut off. So in order to use Fastab, the user has to load code into the microprocessor every time it is reconnected to power. In addition, the Fastab circuit

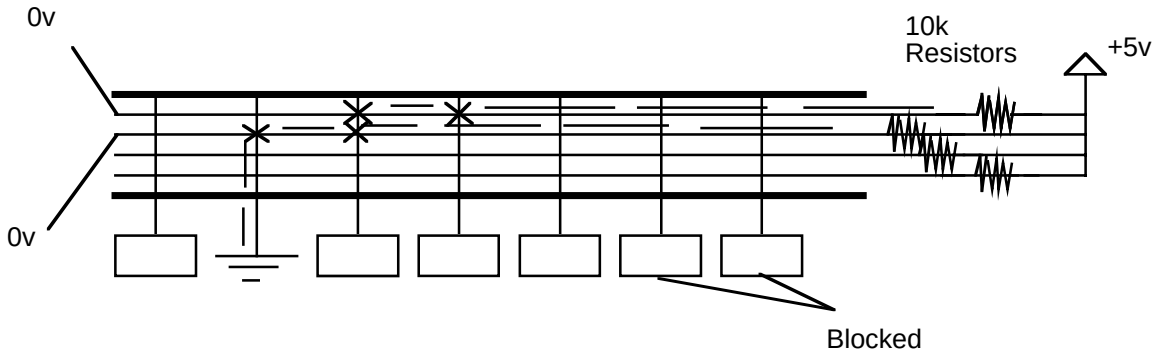
resides on a different board than the microprocessor. A host board provides a chip slot that gives the user access to the ports, the address bus, the data bus, and many control lines. My circuit is constructed on an external perf board (a plastic board with many small holes to allow chip pins to slip through and be connected on the back) and connected to the processor through the chip slot. It makes for a design that is much larger than necessary.

The remedy for this situation is to simply construct my own custom board. One of difficulties is creating the serial connection. Because RS-232 and TTL voltages are different, a few extra chips are needed to aid in this conversion. The most important is the MAX-232. It has an ingenious switching mechanism that allows generation of a 10V to -10V signal with only a +5V source. It does this by charging two capacitors in parallel and then reconfiguring the hardware to discharge them in series. Providing power to the new board is the other major difficulty. That can be accomplished by putting a voltage converter between a 9V battery and the rest of the circuit.

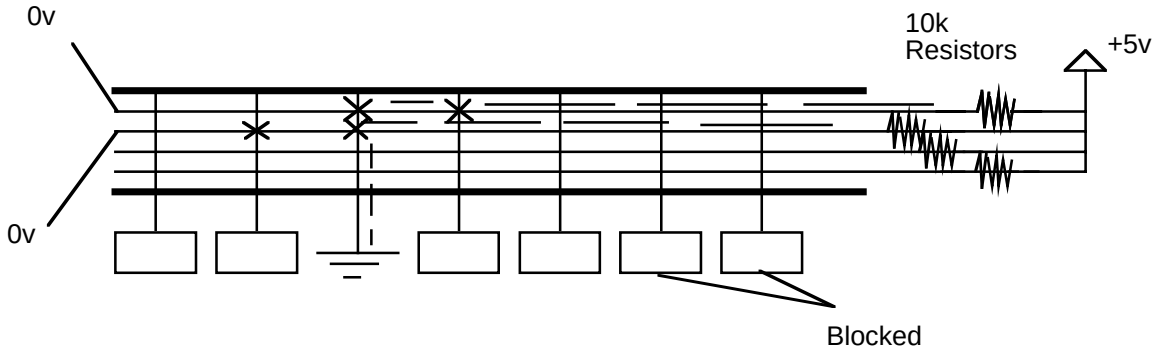
A few functions that appear in the dialog box do not work yet. I have simply not had time yet to implement triplet functionality. The algorithm for checking triplets would be to examine the eltList before rounding to find successive notes that appear to be separated by two-thirds of a normal time division. Triplets are almost always found in groups of three. They are three notes that take the same time value as two quarter notes, eighth notes, etc… The dialog box would also seem to offer the user the option of using a guitar. Unfortunately, I never found a guitar that I liked little enough to destroy with solder. However, Fastab is designed to be very mutable. All the data structures (the arrays of eltLists representing each string, in particular) are dynamically allocated and are based on constants set by the user at run-time. Furthermore, there are no constants in the code and the MIDI note values assigned to each string can also be changed at run-time. As a result, I am confident that once I receive the Fastab Strip (Appendix C has the parts list for the final product) and test the system with a guitar, a few hours should be sufficient to assure full functionality.

The only major issue that this project has left unresolved is chording. In basic terms, the current design does not support accurate notation of certain chord shapes. With the current electrical configuration, this flaw is impossible to correct. It occurs when the user forms a chord shape that allows one string to ground another through the connection in the fret. (see example diagram below). The solution to this problem is to electrically separate the areas of the fret that touch the string, though the connection to the 3-8 decoder must be maintained to all. I believe, however that this correction is not essential to proper operation. In spite of this impediment, correct chord fingerings can still be recorded much of the time. And Fastab still accurately determines the strings plucked and the duration of the notes. So, unless the played piece consists entirely of chords, Fastab still increases the user's speed of notation.

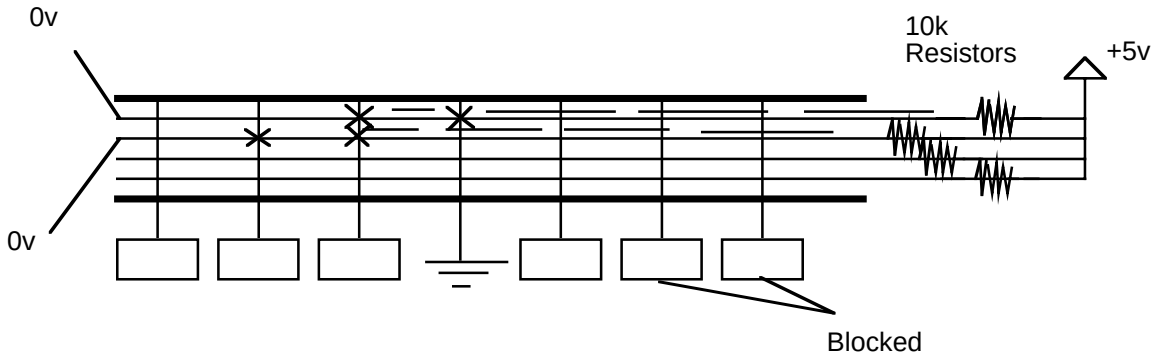
Here the decoder has selected fret 2 while the user is pressing fret 3, string 3 and fret 4, string 4. The sensor reads that both string 3 and 4 are depressed at fret 2. This is incorrect.



Here the decoder has selected fret 3 while the user is pressing fret 3, string 3 and fret 4, string 4. The sensor reads that both string 3 and 4 are depressed at fret 3. This is correct.

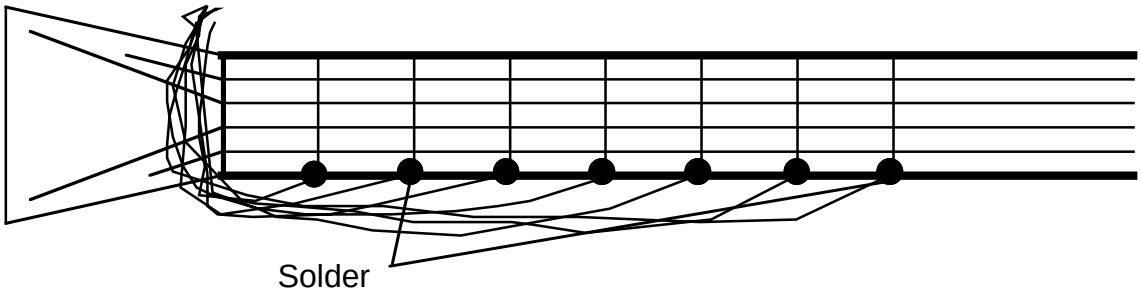


Here the decoder has selected fret 4 while the user is pressing fret 3, string 3 and fret 4, string 4. The sensor reads that both string 3 and 4 are depressed at fret 4. This is incorrect.

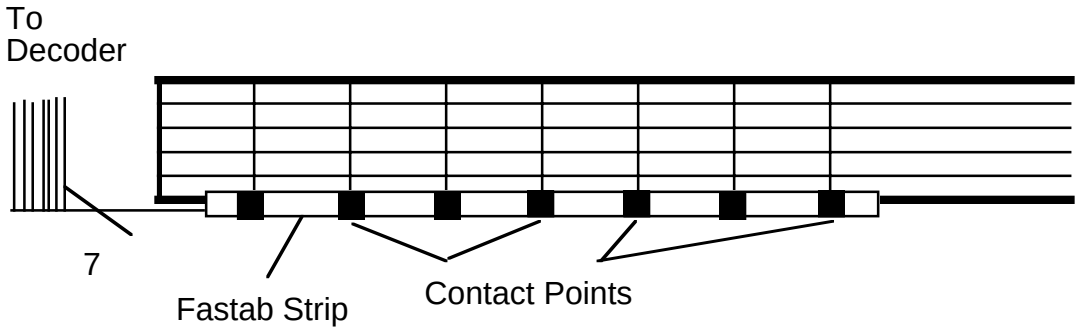


The final issue is one of esthetics and engineering. The fact is that no one will purchase Fastab if they must solder wires to the frets of their instrument. I believe that a simple plastic strip with an 8 pin connector on one end and 8 connection points along its length would solve this problem. I have contacted the companies Texas Instruments and Fairchild Semiconductor and have been assured that this part is a viable option for production.

This messy-looking setup is how my mandolin is wired now:



This is what it will look like after the Fastab Strip is fabricated:



Appendix A – MIDI

MIDI is the industry standard protocol for communicating sounds between computers. It was designed specifically to support data streaming. Generally, it supports 16 channels, each representing a different instrument or voice. Multiple notes can be played on each channel, and each can be controlled separately. The MIDI protocol has 5 different types of messages:

Channel Voice Messages - Stop, start, or alter a sound being played on one channel.

Channel Mode Messages - Messages that affect the entire channel (ie, turn off all notes)

System Real-Time Messages - Used by devices to regulate and synchronize timing.

There are two other types of messages, System Common Messages and System Exclusive messages. They are not used by Fastab. They concern the functioning of specific MIDI devices with which I do not deal.

Because Fastab is not a data streaming system or a real-time conversion program, it do not actually generate MIDI messages. Instead, they are written to a file separated by delta time values as specified by the MIDI file format. MIDI files are separated into chunks. Each chunk consists of a 4 byte type, a 4 byte *length,* and then *length* data bytes. There are two types of chunks. One, the header chunk, has type Mthd (Midi Track HeaDer). The four bytes are actually those 4 characters. The length of a header chunk is always 6: 2 bytes each for the *format* of the file, how many *tracks* are contained, and *divisions,* which defines the default unit of delta time for the MIDI file. I specify format 0 because I only need one instrument track. Format 1 is a MIDI file where multiple tracks are played simultaneously and Format 2 specifies a file where multiple tracks are played independently of one another. The next two bytes contain 1 because I have one track chunk in the file. The last byte, in my code specified as 0xC0, specify how many MIDI ticks per quarter note.

Now that our file is setup, it is necessary only to specify the sounds to come out of our one track. The type of the track chunk is Mtrk (Midi TRacK). The *length* is calculated based on the number of notes that need to be notated, and then the *length* date bytes follow with all the sound information. Because MIDI was designed as for real-time streaming of data, the track's data bytes are written as follows: <delta-time> <event>. It is as if the file recreates the sending of events by separating them exactly as they would be in real time. There are a bunch of standard things that need to be done in the track data before any notes can be written. Things such as setting the tempo and the time signature are taken care of here.

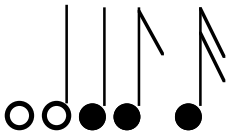
After that housekeeping is dealt with, we are finally ready to specify the notes. We iterate through each chord, turning each note of a chord on with a <delta-time> of 0 and a standard note-on velocity of 0x5F. Velocity is always a parameter for a note-on or note-off event and specifies how abruptly the sound terminates. Lower velocities translate to longer finishing times. Based on the duration of that chord, we have the first note-off event for the chord with a <delta-time> corresponding to the chord's length. Each successive note-off for that chord has a delta time of 0 and a standard note-off velocity of 0x64. The end of the file is specified by 0xFF2F00 and is always included.

Appendix B – Musical Terminology

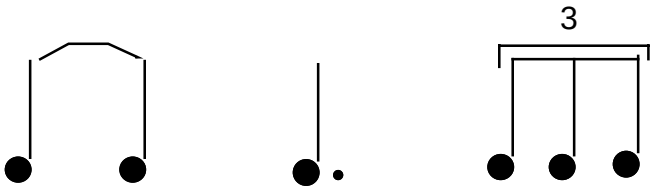
Time Signature: X or X/Y
Y

X = how many beats per measure.
Y = what note gets the beat.

Key Signature: The default sharped or flatted notes specified at the beginning of a piece of music.



Whole Note: 4 beats of 4/4.
Half Note: 2 beats of 4/4.
Quarter Note: 1 beat of 4/4.
Eighth Note: half a beat of 4/4.
16th Note: 1/16th of a measure of 4/4.



Tied Note - Held for both note durations
Dotted Note - Held for one and a half times the specified duration.
Triplets - Three notes which take the same amount of time as two standard notes. There can be quarter note triplets, eighth note triplets, etc…. Above are eighth note triplets.

Hammer-On - A note sounded by playing a note and then placing a finger higher on the fretboard to sound a higher tone.

Pull-Off - A note sounded by playing a note and then removing the finger to sound a lower tone.

Metronome - A device that sounds a constant click to regulate a musician's playing speed.

Tablature - A way of writing out music that displays where a musician places his fingers on the fretboard instead of the actual notes.

Appendix C – User's Manual Parts List

- (1) Hexagonal Serial Cable for connection to PC.
- (1) Fastab Strip
- (1) Fastab Adapter
- (1) set Fastab Strings
- (1) Fastab Pick