7-2008

Real-time MIDI Performance Evaluation for Beginning Piano Students

David Duvall
Clemson University, ivoaebo@yahoo.com

REAL-TIME MIDI PERFORMANCE EVALUATION FOR BEGINNING PIANO STUDENTS

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Science

---

by
David Chris Duvall
August 2008

Accepted by:
Dr. David Jacobs, Committee Chair
Dr. Andrew Duchowski
Dr. D. E. Stevenson
—

Abstract

MIDI is a standard digital protocol for the communication of musical events. In this paper, we examine the construction of a complex system to validate musical performance characteristics without compromising musical interpretation through the use and evaluation of MIDI messages. Beginning music students often have a difficult time translating written, musical characteristics to the correlating sound that they imply. Even though a teacher can effectively help a student through this learning process, the process can typically be slow, as evaluation of musical performances happens only once a week during a thirty minute lesson. Prior research has shown that a model, such as the proposed system, increases the pace of learning. While some commercial, Windows-based applications do exist, we propose a solution for evaluation in real-time giving the student immediate feedback rather than at the end of a performance. We take a detailed look at the development process of our application, Blunote, in a Linux environment built on ALSA (Advanced Linux Sound Architecture) for the beginning piano student.

Acknowledgments

We like to acknowledge Christopher Rath for allowing us the use of various images of musical notes and symbols both within this document and the Blunote application.

Table of Contents

List of Figures

Figure Page

Chapter 1

INTRODUCTION

Blunote is a computer application designed to aid beginning piano students in learning how to play written music. Beginning students have a difficult time translating written notes into the corresponding sound because of the initial complexity of learning music and the relatively short time spent with a teacher. In order to play a musical piece, a performer must exhibit proper technique in order to successfully play sequences of notes and play the appropriate pitches for the notes at the correct volume for the correct durations of time. For piano, variations of pitch are not a concern as it would be for other types of instruments such as a trumpet. A note struck on a piano has the same pitch every time as opposed to other instruments where a range of values is considered the same pitch. Correct volume level as well as duration of notes is arbitrary within the proper range. These variations, or musical interpretations, can make learning initially more difficult, as no human can play any musical piece the same way twice. In a way, musical scores are merely suggestions rather than an accurate description. The result is that a student can have a hard time noticing if their variations really do fall within the acceptable range. Bad habits can form quickly. It can take longer to learn from a teacher when evaluation occurs only once a week.

How can we accurately evaluate piano performances through application software to provide a mechanism for students performances to be assessed without the presence of a teacher? Part of the answer to this problem lies in the larger issue of correctly interpreting musical events given the nature of a performer's musical interpretations. As we will see, the evaluations of musical events to musical scores are easier than creating musical scores from performances in this regard; however, the major problem of evaluating musical performances lies in the computational speed of the application. There are a few commercial products and various academic research that attempt to implement or propose a solution to

the problem of music evaluation, but they vary in the speed in which feedback is given.

The key feature that Blunote provides is giving feedback to the student during the performance as opposed to only generating feedback at the end of the performance. As the student reads the musical score from the terminal and inputs musical events through a MIDI controller, the program determines the accuracy of the pitch and timing of the musical event. Correctly performed musical events change the color of notes in the musical score to blue while incorrectly performed musical events change the color of the notes represented in the musical score to red. When a musician makes a mistake during a performance, he must make an adjustment to perform the next sequence of events correctly. This mechanism of instant feedback provides a student with an opportunity to realize that such an adjustment needs to take place. This is signifigant as beginning students are sometimes unaware that a mistake has taken place. Learning how to make adjustments during the course of a performance is a part of learning how to play piano.

Blunote offers several additional features. Commercial products restrict the availability of musical scores in order to generate income from users purchasing more songs from them; however, some of these products do allow a user to record their own song for performance evaluation. Blunote primarily uses the latter method, giving a teacher the resource to record songs for a student to practice, but as an additional feature, any MIDI file (of format zero) may be downloaded, converted into a musical score, and used with the performance evaluator. Thus, an endless resource of songs exists for teachers to give their students or for students wishing to learn additional songs without the aid of a teacher. Student performances can also be converted into MIDI files which can then be played on any audio player that recognizes MIDI files. This preserves a student's interpretation of a musical piece, encouraging the student as well as providing long distance learning as a teacher could evaluate the student's interpretation through the recorded performance. Lastly, an inherent aspect of a performance evaluator is the ability for the student to hear the musical piece as recorded before attempting to peform the song.

The music education application software industry offers a few, comparable products in this area, such as "Children's Music Journey" and "Piano Tutor", yet academic research in music education software application, particularly in the area of performance evaluation,

is sparse and progressing at a rate far less than the progress of the computational power of computers, providing many areas in need of research. As a result, most of the research does not focus solely on performance evaluation. Either the research is focused on a broader aspect of music education and sparingly discusses the complexities of a performance evaluator, or the research is more concerned with studying the benefits of performance evaluators. In addition, research centers on questions of why music evaluation applications are important or what they are as opposed to how to implement one. The few commercial products are all developed on popular platforms such as Windows and Mac, but there is seemingly no music evaluator written in the Linux environment. The goal of this paper and the Blunote project is to provide an open source, music evaluation application that provides feedback immediately to the student during performance, and to provide a detailed look at the software development life cycle of building such an application.

Because of the complexity of building such an application, Blunote is a simplified model of a solution. The major parts of the solution are implemented and discussed while the parts not implemented are easily extendable. Blunote evaluates correctness of pitch and timing, but not velocity. While variable time signatures are not a feature of the application, the most used time signature, 4/4 time, is realized in such a way that extending functionality to variable time signatures is trivial. Durations of notes are only subdivided to eigth notes. While this does make the application very basic in nature, it allows us to concentrate on the larger details and avoid the repetitive nature of a lengthier implementation. In the same manner, MIDI files of format zero are the only format recognized by the application. While only one key signature will be implemented, the key of C, the ability to alter beats per minute is an integral part of the solution and will be implemented to specification.

1.1 Definition of Terms

Before continuing it will be helpful to clarify some of the terminology that will be utilized in subsequent chapters. As we will see, in the digital world, a piano produces the same pitch value every time the same key on a digital piano is pressed as opposed to other instruments whose pitch value may vary within a specific range. Therefore, when pitch is referred to

in this discussion, it is referring to one, specific frequency of a note. It does not refer to variations within a given range. The velocity of a note refers to the volume level associated with that note or how hard a key is pressed, and how long a note is held will be referred to as the duration of the note. Since these two qualities of a note can vary within an acceptable range, we define quantization as the process of aligning these qualities to precise values. Standard musical terms are defined in Chapter 3.

1.2 Description of Remaining Chapters

In the remaining chapters we will discuss the surrounding issues involved in implementing a real-time MIDI evaluator step-by-step through the software development life cycle. We will begin with a look at relevant work in Chapter 2. Chapter 3 is a tutorial on basic music terminology as well as MIDI, the standard digital communication protocol for communicating musical events. We will start with a basic introduction to music for non-musicians. Then the tutorial will give an overview of the MIDI protocol, the basic structure of MIDI messages, and the relevant information needed for MIDI file formats. In Chapter 4, the software development approach is discussed. We will approach the problem from the perspective of the student and formulate the necessary requirements and specifications of Blunote. One of the more important design decisions, ALSA, will be discussed in detail, and then an overview of the general design of the project will be presented. Chapter 5 will detail the implementation of Blunote in five phases: sound generation, timing information, the sequencer, the graphical user interface, and the performance evaluator. We examine the mechanisms of the sequencer responsible for recording, playback, memory management of musical events, and parsing of MIDI file formats. Because musical performances inherently rely on timing characteristics, the graphical user interface will be discussed in relation to the timing constraints of the program.

Chapter 2

BACKGROUND

There has been a plethora of academic research and computer products developed in the area of performance evaluation. Performance evaluation evolved from score following systems which are systems that monitor live performances for the purpose of providing automated accompaniment [Tekin et al. 2005]. Because of the constraints of computational speed during early development, real-time performance evaluation could not be attained. Focus shifted in subsequent development to the precision of evaluation as opposed to the speed of giving feedback to the user. In the last decade, several commercial products have emerged utilizing performance evaluation with varying speeds in which feedback is provided.

Danneberg et al. [1990] developed one of the first systems to incorporate performance evaluation called "Piano Tutor"; however, the goal of the project was not performance evaluation but to devise an intelligent based teaching system. Much of the focus of the project is based on the development of an expert system to coordinate the order in which lessons are given to the student based on the student's progress. A student performs a given set of exercises, and based on the student's performance the necessary lessons or exercises to strengthen the student's ability are then presented. During the course of the performance, the system is able to make some basic decisions in real-time, but Piano Tutor is not able to give feedback in real-time. Evaluation only occurs at the end of the performance. According to the authors, "one of the key constraints on the design of the Piano Tutor has been real-time performance. It is essential that the system be able to interact rapidly with the student."

During a comparison of another intelligent tutoring system, the LISP tutor which was created out of cognitive research by Anderson, Danneberg et al. [1990] point out that rele-

vant aspects of cognitive theory in the system are immediacy of feedback and the assumption that knowledge is proceduralized. The point is that we learn how to perform complex tasks through first learning a detailed description of how to do something and then by actually doing the event itself. Naturally, it follows that it would be beneficial to receive immediate feedback during the performance of the event. As stated earlier, Piano Tutor could not achieve immediate feedback, and the cost of a single system to use Piano Tutor was around ten thousand dollars when it was developed.

Seemingly because of the timing constraints and the state of technology of the time as well as the need to explore more precise evaluation methods, subsequent performance evaluation applications were developed shortly after Piano Tutor that focused on the quality of feedback given presumably to the student at the end of a performance. Smoliar, Waterworth, and Kellock [1995] developed "pianoFORTE" in attempt to make it easier for a teacher to communicate to the student "the distinction between the art of playing piano and the technique of playing the correct notes." In their discussion, they point out that Piano Tutor was designed for beginning students who needed to learn the skill of notation literacy, but this skill was not the goal of piano education. By recording a student's performance and concentrating on four distinct characteristics: dynamics, tempo, articulation, and synchronization, a teacher could then use the graphical displays of pianoFORTE to assist in the critique of the student's interpretation of the piece. The focus of pianoFORTE is therefore on assisting the teacher in student assessment as opposed to providing immediate feedback for the student.

Other notable programs that have been developed include the "MIDIator" which is a software tool designed to analyze student piano performances [Shirmohammadi et al. 2006]. MIDIator allows an instructor to analyze a student's performance in real-time or at a later point through a series of graphs depicting the volume levels of the notes played and the pitch durations of the notes. A student or teacher can also examine the performance through the comparison to other performances such as one previously performed, one performed by another musician, or a performance that exemplified how the music would sound if it were played exactly and devoid of expression. While a teacher could receive immediate feedback from the performance facilitating long-distance interaction, immediate student feedback is

not given.

Heijink et al.[2000] survey a variety of approaches to score performance matching which they define as "the procedure which relates events in a performance to the corresponding events in a score." Algorithms are placed into two categories: focusing on real-time matching and non-real-time analyses. Although humans may translate from music scores easily, digital score matching is complicated by performer errors, the use of expressive timings, and under specified musical scores.

From an educational perspective, McKinnon [2005] describes the benefits of music education to the development of young children. McKinnon claims that "a research team exploring the link between music and intelligence reports that music training - specifically piano instruction - is far superior to computer instruction in dramatically enhancing children's abstract reasoning skills that are necessary for learning math and science." The result is that regardless of social or economic backgrounds that children who are actively involved in music education receive higher marks on standardized tests than those children who are not involved.

After exploring the benefits of music education, McKinnon [2005] details the commercial product, "Children's Music Journey", developed by Adventus Interactive. The product is a three year course of study through an animated interactive music learning program for children ages four to eight, and the total cost of implementation of this program for twenty-five workstations including all hardware and software needed as well as support would be around five thousand dollars. McKinnon claims the product gives immediate feedback on student performances.

After the development of Piano Tutor, most of the research seems dedicated to further defining the communication of musical expression for the purpose of music education as opposed to an aid during performance. The complexity of music forces the resulting analyses of music performance to also be complex. Thus, the resulting feedback provided is a detailed description of how music is played, yet little or no feedback is given while the performance is taking place, arguably the most important time to receive feedback.

If one were just learning how to ride a bicycle, one would receive instructions and then try to enact those instructions by actually riding a bicycle. After a failed attempt, a technical

representation of what you did wrong may help, but short, meaningful assistance verbally during your attempt would be much more helpful. After all, riding a bicycle is more about the "feel" of riding the bicycle. Short verbal assistance helps one to define the parameters of the "feel" while it is occurring. The same type of learning occurs while performing music. According to Smoliar, Waterworth, and Kellock [1995], "any music technology which does not account for listening runs the risk of short-changing its users. The ultimate goal of pianoFORTE is to make us all better listeners." While becoming better listeners is a key step in learning how to play music, listening should not be the goal. The goal should be to assist students in becoming better players through learning the "feel" of performing music which can only happen through short, meaningful, and immediate feedback.

Chapter 3

MIDI TUTORIAL

Before we can begin looking at the requirements of building a performance evaluator, we need to establish a firm understanding of basic, standardized musical principles as well as digital communication protocols in order to possess the background needed for the construction of our application. The focus here is discussing how musical events are communicated digitally as well as how musical events are communicated from musical notation to the performer. The latter is inherently crucial to our topic as the goal of a performance evaluator is to assist a performer in learning how to interpret musical events from musical notation. Therefore, we will begin with a short introduction to music from the perspective of the communication of musical events through music notation. After examining this interaction, we can then look at MIDI, the standardized digital communication protocol for musical events, the MIDI messaging system, and how musical events are stored on a hard drive through MIDI file formats.

3.1 Basic Music Terminology

As we begin our discussion on music terminology, it is important to understand that music performance is nothing more than a sequence of musical events. Music notation is the process of defining these events in such a way that a performer can recreate those events. For a musician, one musical event is labeled a note, and a collection of musical events is labeled a song or score. For novices, music notation can initially be intimidating. Figure 3.1 shows the musical notation for a typical song or score. The notation system can be subdivided into three categories according to the information that it conveys: pitch, timing, and velocity. For each category, certain symbols provide a framework for the collection of all musical

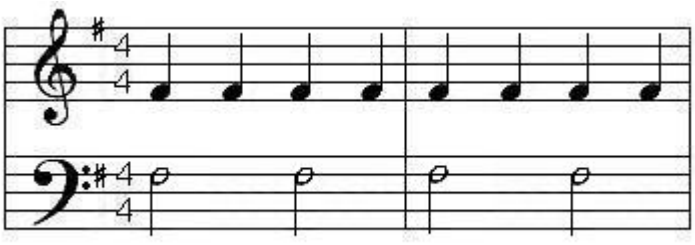

Figure 3.1: Basic music notation example

events, while other symbols describe one of these three qualities for a specific musical event. We will deconstruct each category so that we may fully understand the music notation system. Note that, for our purposes, we are limiting the discussion to the basics of musical notation and the relevant material that we will need in our project, but in some cases we will elaborate for the purposes of providing background. Therefore, instead of covering every musical notation symbol, the discussion will only center around the notation as depicted in Figure 3.1.

The basic framework for denoting pitch is centered around the use of a five line staff (Figure 3.2). A clef is used to indicate the range of pitches for a given staff. For piano, two clefs, treble and bass (Figure 3.2), form the basic range of pitches that a piano can produce. The placement of a note on a staff determines a note's pitch. The lower the note is located on the staff, the lower the pitch, and the higher a note is on the staff, the higher the pitch. However, a five line staff can only represent twenty pitches without more advanced notation. With two staffs, currently we can depict forty pitches, yet a piano can produce eighty-eight pitches. Therefore, notes that occur outside of the scope of the staff system can be represented with the use of ledger lines which provide a single note with additional staff lines. An example of ledger lines is given in Figure 3.3. The second quarter note depicted, middle C, is the pitch that separates treble and bass clefs. Although there are exceptions to this rule, it can generally be perceived that notes lower in pitch than middle C are placed in relation to the bass clef and are performed with the left hand while notes higher in pitch are placed in relation to the treble clef and performed with the right hand.

Now that we have discussed the framework for defining pitch, we will discuss the specifics of mapping a note for a specific pitch within the framework. There are only twelve

(a) Five line staff (b) Treble clef (c) Bass clef

Figure 3.2: Staff basics

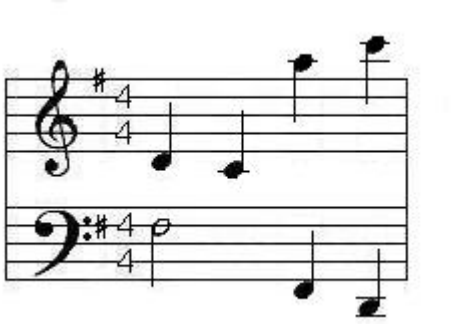

Figure 3.3: ledger lines

basic pitches in music; however, these pitches may be played at different frequencies. So, the same pitch can be played higher or lower in relation to itself. In order to simplify our discussion, we have been using the definition of pitch to encompass frequency so that different frequencies of the same pitch are considered different pitches, but we will have to make a distinction in this section only. If you examine a piano from left to right, the first twelve notes or pitches are repeated seven times at different frequencies for a total of eighty-four notes. (There is an incomplete repetition at the farthest right of the piano comprising four notes.) The seven white keys of the first twelve pitches are labeled in order alphabetically A through G (Figure 3.4). The five black keys can be labeled in two different ways, either in relation to the white key directly before it or the white key directly after it. Therefore, the first black key can either be defined as A sharp or B flat, where sharp is defined as raising the pitch one interval higher and flat is defined as lowering the pitch by one. In a similar manner and without the use of key signatures, a note placed either directly on a staff line or in the space between two lines, denotes one of the seven white keys. In order to denote a black key, the note must have either a sharp or a flat symbol preceding it (Figure 3.5); however any note may be made sharp or flat, meaning that either the note should be raised or lowered one interval respectively. In Figure 3.5, the first two notes are the same note, F,

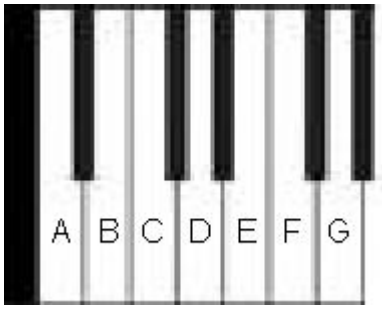

Figure 3.4: Twelve basic pitches

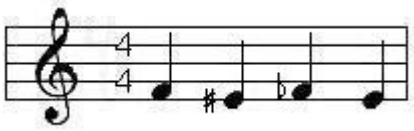

Figure 3.5: Sharps and flats

and the last two notes are the same note, E, as there isn't a black key between the notes.

Given the ambiguity of our current model of labeling certain pitches, music notation utilizes key signatures to provide a more accurate representation and for music theory not covered in this document. A key signature specifies which notes are commonly sharps or flats in a musical score removing the need of placing a sharp or flat before certain notes every time in a musical score. There are twelve different key signatures, one for every unique pitch. Each key denotes either how many pitches should be sharp, or how many pitches should be flat. In Figure 3.1, directly after the treble clef, the key signature is given. In this case, the key denotes that every F note should be sharp. Therefore, this implies to the performer that when any F note is encountered regardless of frequency, that an F sharp should be played instead. If a composer wanted to make an exception such as playing an F for one specific musical event even though the key signature denotes an F sharp, the composer may enter a natural before the note to signify that the unaltered version of the note should be played instead. Figure 3.6 displays three F sharps and one note as a natural F, the last one.

The basic timing framework can be subdivided into two categories: time signatures and tempo. In Figure 3.1, directly to the right of the key signature is the time signature which denotes how many beats are in each measure (the top number) and what constitutes one

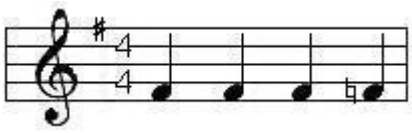

Figure 3.6: Natural

beat (the bottom number). The time signature as well as the key signature is denoted twice, once for each clef. A measure or a bar constitutes a division of time within the score. In Figure 3.1, the vertical line in the middle of the image denotes the end of one measure and the beginning of another. The top number of the time signature in Figure 3.1 denotes that there are four beats in a measure and the bottom number of the time signature in this example, also a four, denotes that a quarter note represents one beat. Therefore, Figure 3.1 represents a musical score with four quarter note beats in a measure. Tempo, or beats per minute (bpm) defines how quickly to perform these notes and is usually assigned a value by the performer. Bpm is not usually denoted within a musical score, but a typical tempo value is one hundred beats per minute. Therefore, in our example with this tempo, the performer would play the equivalent of one hundred quarter notes per minute.

Within the timing framework, a musical event is given a timing characteristic. This timing characteristic represents the length of a musical event, or in relation to a piano, how long a key a pressed and held. The time signature in our example is the most common time signature. Popular music, for example, almost exclusively uses 4/4 time. Therefore, in relation to our time signature, a note that is held for an entire measure or four quarter note beats is called a whole note. As depicted in Figure 3.7[1], a whole note can be subdivided according to duration. The second half of Figure 3.7 denotes the musical notation of rests according to their duration. A rest simply indicates the absence of musical events for a given period of time. At the bottom of our image, there are note values with dots beside them. These values indicate one and a half of the current value. Ties are used mostly to represent a timing value that continues into the next measure. This is the method for insuring that there are not more beats labeled in one measure than there should be. For example, if on the fourth beat of a measure, a musical event of a half note duration needed to be executed,

[1] http://cache.eb.com/eb/image?id=6238&rendTypeId=4 accessed 7-23-08

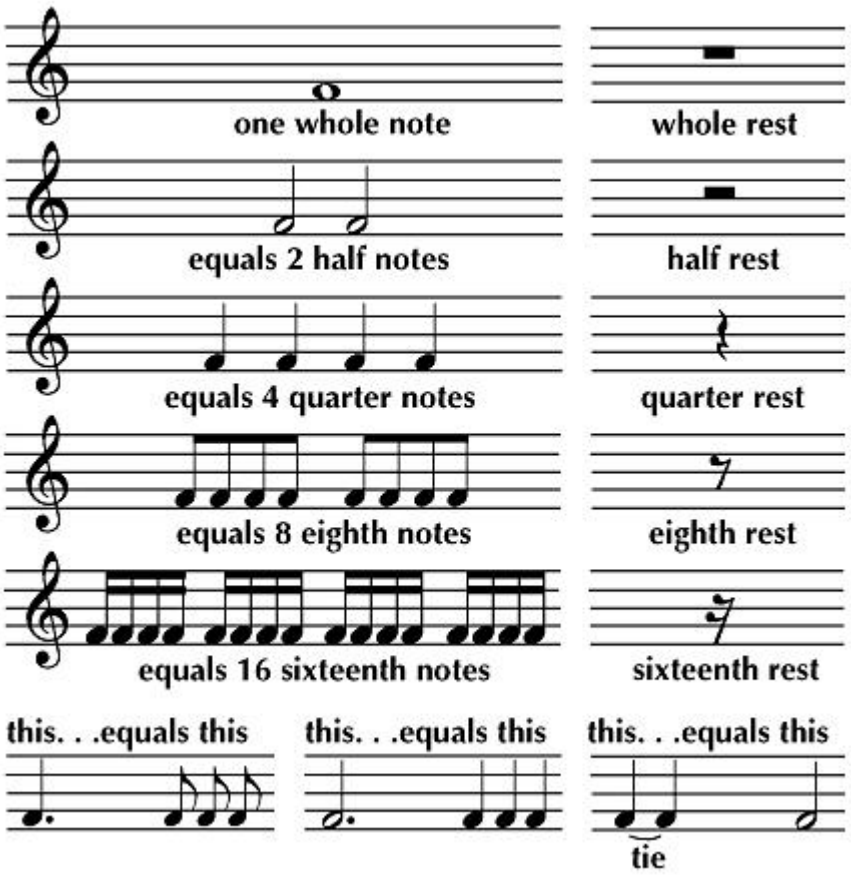

Figure 3.7: Basic subdivision of timing values

a quarter note would be denoted in this measure tied to a quarter note in the next measure.

While the discussion has centered around key background information needed to understand the construction of Blunote, it is important to understand that there is more to music notation than we have covered. We have defined that a musical event can be segmented into three main categories: pitch, timing, and velocity. While the construction of Blunote will center on evaluating pitch and timing, velocity should be mentioned. Sections of a musical score are given a specific range of volume for all musical events. Volume or velocity can then be altered within the framework through use of symbols to denote musical ideas as crescendos or accents. However, just as other musical notation examples such as double sharps and flats will be left out or further subdivisions of timing values of notes, the focus of Blunote is on how to build a basic performance evaluator as opposed to building an all encompassing performance evaluator.

3.2 Overview of MIDI and MIDI Messages

Now that we've discussed what type of information needs to be communicated, we will look at how musical events are communicated by computers. MIDI or Musical Instrument Digital Interface, is an industry standardized digital protocol facilitating the communication and synchronization of electronic musical instruments and computers. This protocol "has been widely accepted and utilized by musicians and composers since its conception in the early 1980's [MIDI Manufacturers Association 1996]". Since then, MIDI has remained virtually unchanged and has been successful enough to be regarded as the "lowest common denominator in the world of musical information [Selfridge-Field 1997]". Other protocols have been developed, but MIDI remains the leading, standardized protocol in musical digital communication.

MIDI connects a MIDI controller, such as an electronic keyboard, to a computer or other modules. A MIDI controller is played like a musical instrument, and it translates the musical events into a MIDI data stream. Traditionally, the MIDI interface uses a MIDI cable, to facilitate a connection between two, five pin MIDI ports, but recent advances in technology have resulted in the use of USB and Firewire connections as well. MIDI is a digital communication protocol, so it does not transmit an audio signal. Instead, it transmits musical event information through the use of serially transmitted "MIDI messages". Because this information contains measurable characteristics such as pitch, timing, and velocity, an application or module, called a synthesizer, can then reconstruct an audio signal from this information. To avoid confusion, an electronic keyboard is commonly referred to as a "synthesizer", but it is called such because the audio reconstruction capability is a built in feature. In other words, the resulting sound is digitally created even when producing sound without the aid of a computer.

Synthesizers only produce an audio signal from digital data. A sequencer, another application or module, is designed to manage a "sequence" of musical events, and facilitates recording, playback, and editing. Many MIDI messages include a four bit channel number to identify a logical division of the physical channel. In this way, a sequencer can support up to sixteen different sequences of musical events simultaneously. For example, a performer

could play up to sixteen different parts individually and then have the sequencer play back all of the parts at the same time. Therefore, when a key is pressed on a MIDI controller, a MIDI message is sent to the sequencer on the appropriate MIDI channel, processed, and then, if necessary, forwarded to a synthesizer that produces an audio signal. As we will see, because MIDI is standardized and because MIDI makes allowances for vendor specific information transfer, portability is not problematic.

MIDI messages require several layers to provide semantic meaning. From the lowest level viewpoint, the MIDI data stream is unidirectional and asynchronous with ten bits transmitted per byte, consisting of a start bit, eight data bits, and one stop bit [MIDI Manufacturers Association 1996]. The start bit is always zero and the stop bit is always one. After these bits are stripped away, the middle eight bits of the ten bit word can be classified as either a status or data byte. A status byte signifies the type of MIDI event taking place, and depending upon what MIDI event that is, a number of data bytes, generally up to three, may follow describing that event or none at all. The most significant bit of the byte determines if it is a status or data byte. If the most significant bit is a one, then it a status byte. A data byte is indicated by a zero value for the most significant bit.

From an abstract viewpoint, the MIDI events described in MIDI messages can be categorized as either system or channel messages. A system message is a message that is not channel specific, and therefore contains no channel number in its status byte. System messages affect the parameters of the environment and are not used in transmitting information about specific music events. System messages can be further classified either as a System Common Message, a System Real Time Message, or System Exclusive Message. System Common Messages are used for advanced sequencers and drum machines to initiate such events as song selection or tuning. System Real Time Messages are used to synchronize all timing events within a collection of modules, and System Exclusive Messages define the transfer of vendor specific information. A channel message may be classified as either Channel Voice Messages or Channel Mode Messages. Much like a system message, Channel Mode Messages differ in that they affect the parameters of a specific channel instead of the overall system. Channel Voice Messages are used to send information about specific musical events, and therefore they are the most utilized.

The most commonly used Channel Voice Messages are the MIDI events `Note On` and `Note Off`. When a user presses a key on a MIDI controller, a `Note On` event is sent to the receiving application. The event consists of the status byte signaling the event, a data byte for the note's pitch, and a data byte for how hard the note was struck (velocity). Subsequently, when the note is released, a `Note Off` event is sent to the receiving application. Note that no timing information is sent as it is the sequencer's responsibility to interpret timing characteristics from the difference in the two events if necessary. Some systems send a `Note On` event with velocity of zero, to indicate a `Note Off` event in order to take advantage of the concept of running status.

Running status optimizes a string of consecutive messages of the same event type. For example, a `Note On` event is always three bytes, the first being the status byte, indicating a `Note On` event. After sending a `Note On` event, a MIDI controller may omit sending the status byte of the next event if the next event is the same as the one just sent or in this case, another `Note On` event. This becomes particulary important when the performer hits multiple notes at the same time. Because MIDI is transmitted serially, this could potentially cause a problem. The standard transmission rate of a traditional MIDI cable is 31.25 Kbit/s meaning that a three byte `Note On` message takes less than one millisecond to be sent [MIDI Manufacturers Association 1996]. A three note chord takes around three milliseconds to send without running status. While this delay is imperceptible to the human ear, a perceptible difference could be discertained from a sequencer attempting to play back a number of individually recorded parts incorporated with a large number of simultaneous events. A human ear, at best, can only perceive the difference in arrival times of two sounds, in one ear, of about twenty milliseconds [Cox 1984].

In this section describing MIDI and MIDI messages, we have looked at the overall system for communicating musical performance characteristics in real time. MIDI can be used for a plethora of other type events such as pitch bend, sustain, and modulation, but our focus is on performance and events depicted in musical scores. Because MIDI is digital, every event has a code associated with it, and these codes are listed in the Complete MIDI 1.0 Detailed Specification handbook [MIDI Manufacturers Association 1996]. Since we may want to store songs, the last thing we will need to look in this tutorial is how MIDI is

stored on the hard drive.

3.3 General MIDI File Formats

SMF, or Standard MIDI Files, provide a specification to store MIDI messages in a standardized file format. These files use a ".mid" extension and can be played with any General MIDI player. Standard Midi Files are maintained by the MIDI Manufacturers Association and were designed to "provide a way of interchanging time-stamped MIDI data between different programs on the same or different computers" [MIDI Manufacturers Association 1996]. On the lowest level, SMF are binary files containing a series of eight bit bytes that are generally viewable by converting to hexadecimal to ASCII. A key difference with real time MIDI messages is that SMF store MIDI messages with a series of bytes defining the timing of individual musical events. In addition, Standard Midi Files store the overall information of a song such as time signature, key signature, and tempo. Our general focus, however, will be on how sequences of musical events are stored.

There are three formats or categories of Standard MIDI Files. As explained in the previous section, a sequencer can be used to record or play back multiple individual parts simultaneously. Each of these parts is called a track which represents a specific sequence of musical events. The particular use of tracks defines the difference between the MIDI file formats. Format zero, the most basic format, is generally used for a single track performance. This format is particularly useful if the entire performance can be held in main memory. If a song requires the use of multiple tracks, either format one or format two is used. Format one represents multiple tracks of one song, while format two is utilized if a single track may represent a whole song. Of these formats, format two is the least used as there are many sequencer applications that will not even recognize this format. Conversely, it is inherently possible to convert multiple tracks in format one into one long track in format zero. A single, live performance will be utilized in Blunote and will be best suited to format zero.

Before analyzing the specifications of a MIDI file, we need to examine the use of a specific convention, variable length quantities. Since one of the goals of MIDI files is

Number(hex)	Representation(hex)
00000000	00
00000040	40
0000007F	7F
00000080	81 00
00002000	C0 00
00003FFF	FF 7F
00004000	81 80 00
00100000	C0 80 00
001FFFFF	FF FF F7
00200000	81 80 80 00
08000000	C0 80 80 00
0FFFFFFF	FF FF FF 7F

Figure 3.8: MIDI file examples of variable-length quantities
[MIDI Manufacturers Association 1996]

compactness, particular parameters described in SMF, do not have a set length. Therefore, the most significant bit is used as an indicator for length. All bytes except for the last one have their most significant bit set. The last one will have its most significant bit clear, so if the value can be expressed as number between zero and one hundred twenty-seven, then it can be represented as one byte. Figure 3.8 shows some examples of numbers represented as variable length quantities.

As an example of variable length quantities, timing characteristics are represented as delta time which describes the length of time since the last event. Naturally, the first event would have a delta time of zero, as there is no event preceding it. Delta times precede all events and can be assigned anywhere from the hex values of 00 to FF FF FF F7, so that they will fit into a thirty two bit representation. Larger number are possible, but $2 * 10^8$ 96ths of a beat at a fast tempo of 500 beats per minute is four days which is certainly long enough for any delta time, especially considering that most experts would have considerable problems playing a series of notes that subdivided at half that speed [MIDI Manufacturers Association 1996].

MIDI files are subdivided into chunks. There are basically two types of chunks, a header chunk and one or more track chunks. Each chunk begins with a four character identifier labeling the chunk and a thirty-two bit length describing the amount of data to follow, not including the eight bytes for the identifier and the length. The data section that

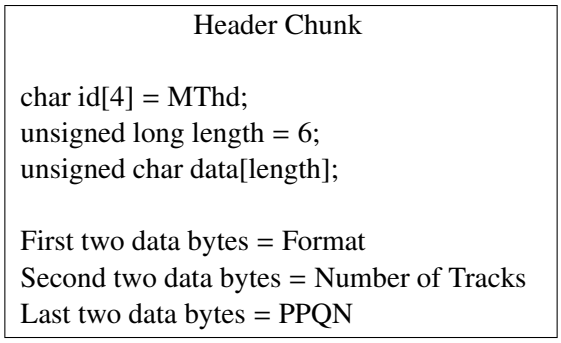

Figure 3.9: Format of a MIDI file header chunk

follows is then a series of bytes storing MIDI events. Allowing for future or vendor specific development, the standard for reading in MIDI files is to ignore any foreign information.

A MIDI file always begins with a header chunk, as shown in Figure 3.9, and has the identifier "MThd". It includes file information such as the format, the number of tracks in a file, and pulses per quarter note or PPQN. Basically, PPQN, defines how to interpret the delta times for musical events. For example, if PPQN had a value of ninety six, then we could interpret a delta time between two events in a file as: 384 = whole note, 192 = half note, 96 = quarter note, 48 = eighth note.

There may be multiple track chunks in a file where each track chunk describes a single track in a song. In a MIDI file of format zero, there is only one track defined in a track chunk, as shown in Figure 3.10, that has the identifier of "MTrk". This section of the file describes the sequence of musical events. An event such as a `Note On` event is ordered by its variable length delta time, then its status byte along with the required number of data bytes describing the event. Running status is used in conjunction with files as well. While this describes a typical MIDI event in a given MIDI file, system messages and non-MIDI events are described in track chunks as well. These non-MIDI events are called Meta-Events and describe general track information or events such as time signature, key signature, and end of track. This is also the section where the opportunity is given for vendor specific information.

As a result of discussing background concepts needed for the construction of a performance application, the beginning stages of design can now be investigated. We have seen

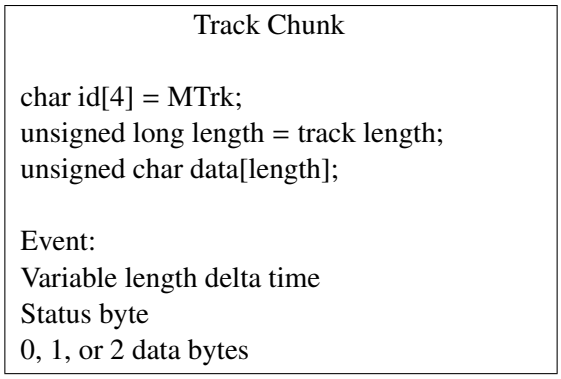

Figure 3.10: Format of a MIDI file track chunk

how basic musical concepts are communicated to a performer as well as how MIDI communicates musical events to a computer. Because a performance evaluator assesses a single performance, format zero is a suitable MIDI file format for our purposes. The next step will be to explore how the parts of the digital communication process such as a MIDI controller, a sequencer, and a synthesizer can work together to provide performance evaluation.

Chapter 4

SOFTWARE DESIGN

Now that we have discussed the basic understanding of musical events, how musical events are digitally communicated, and how these events are stored in MIDI files, we can develop a cohesive plan for developing our project, Blunote. By providing an overall context of the project, we can then narrow our focus on implementation details in the next chapter. Section 4.1 details what we would like the project to do, what we would like to accomplish in the development of Blunote, the constraints and limitations of the project as well as hardware and software specifications. Section 4.2 provides background information on the utilization of ALSA (Advanced Linux Sound Architecture) and section 4.3 pulls all the details together, providing an overview of the design.

4.1 Requirements and Specifications

A short description of Blunote is a software application to evaluate student performances on the basis of pitch and timing in comparison to musical events depicted in a musical score. From looking at this basic description, we can see that we will require a MIDI controller to input events from the student through a MIDI connection to a computer running our application. We will need the ability to collect data from the MIDI controller as well as the ability to generate sounds from the transmitted MIDI messages. In order to evaluate performance, we will need musical scores stored in MIDI files, so we will require the ability to read in MIDI files into our application, data structures to hold this information in memory, as well as a way to display this information to the student in the appropriate musical score format. Optimally, we would like some way for a teacher to input their own songs for the student to practice, so we will need to develop an approach of building MIDI files through

a fully defined recording process. Finally, we need to define the performance evaluation process as well as define the type of feedback presented to the student.

Ultimately, the feedback given by the application is the goal of the project. Danneberg et al. [1990] describes the difference between qualitative versus quantitative feedback. An example of qualitive feedback could be "you are slowing down" given during the performance while an example of quantitative could be more precise information in the form of a graph of tempo versus time. As we have seen, ample research has been done in fully developing quantitative feedback [Shirmohammadi et al. 2006] [Smoliar et al. 1995], but detailed, quantitative feedback ends up only being a useful aid to a teacher. This bring up the question, does the teacher even need this aid? Smoliar et al. [1995] states the ultimate goal of their system is to make us better listeners, but presumably teachers have already acquired this ability and can accurately evaluate performances without the need of technical analysis. A further point was made that quantitative feedback is useful to a teacher because once the note is played, it exists from that point on only in memory. First, anyone that has ever taken a piano lesson knows that an instructor predominately does not make corrections after the piece is played but during the performance, often stopping the student during the performance or asking for segments to be played again. Second, if long distance teaching is a goal and a teacher has the ability to view technical analysis of a performance, he would also have the ability to re-play the performance along with the normal functions of pausing, and rewinding. Finally, from the student's perspective, cognitive based learning dictates that corrections are made during the performance as opposed to trying to equate technical data with memories of playing a sequence of notes.

Most beginning students are under the age of ten, and the project is focused on aiding the student rather than the teacher. Extensive quantitative feedback is not going to mean very much to them. However, short, meaningful, and immediate feedback aids beginning students to acquire notation literacy and attain the "feel" of music on a level that they can understand quickly in a form of qualitative feedback. Therefore, the criteria that we will establish is a simple feedback mechanism in which notes that are not played correctly according to pitch and timing change color immediately. Also, we will establish that any quantitative feedback given at the end of the performance will be customized to students

under the age of ten.

The limitations of the project involve only evaluating basic musical characteristics while the constraints of the project involve timing characterstics. Because of the focus of our project is short, qualitative feedback, performance evaluation will center on the comparison of pitch and timing only. Since this project is also meant as an introduction to performance evaluation, we will only explore the implementation of basic musical ideas. Basic musical notation will be limited to treble and bass clefs, staffs, measure lines, valid piano pitches, the key signature of C, 4/4 timing signature, and subdivisions of individual musical events only involving whole notes, half notes, quarter notes and eigth notes. The principle constraint of the project involves the real-time aspect of performance evaluation. As we have defined, a human ear, at best, can only perceive the difference in arrival times of two sounds, in one ear, of about twenty-five milliseconds [Cox 1984]. Therefore, in order to replicate the sound of the performance given, the entire audio processing for an individual musical event must execute within this time frame. We will establish that there must be a separation between graphical aspects of the application, including comparison analysis, versus audio aspects of the application, including timestamping the event as well as playing the associated sound of the event. Thus, a higher priority is given to the audio aspects of the application.

We need to establish some hardware and software specifications for building our application. Although we will make a requirement that any MIDI to USB connection will suffice provided that the appropriate software drivers are installed to recognize the connection, the project will be implemented utilizing a M-Audio KeyRig 49 USB Piano Keyboard and a standard MIDI to USB cable. The application is implemented on a standard T41 IBM laptop with no modifications to the hardware in a SUSE Linux operating system environment using C++. In conjunction with providing Blunote as an open source application, we will also be incorporating other open source applications to contribute to the project. The application, Kaconnect, written by Matthias Nagorni with some code segments written by Takashi Iwai, facilitates message communication between various programs, namely by binding ports between the USB port and our application and binding our outgoing application port to a synthesizer program port. The synthesizer application that we will use

is ZynAddSubFX, written by Nasca O. Paul[1], although any synthesizer application would suffice.

In this section we have determined that immediate feedback during performance is the goal of Blunote as well as examined the limitations and constraints of the project and detailed the individual specifications that are utilized to build such an application. We will discuss in the next section the design decision of which architecture has been chosen for the project. Afterwards, we will bring all of the design details together from an abstract viewpoint.

4.2 ALSA

In approaching the design of this project several key issues needed addressing. First, how can we abstract the MIDI port so that our application works with any MIDI connection the user chooses such as traditional MIDI, USB, or Firewire? Also, what method of data collection needs to be utilized? Secondly, how do we establish a connection and route MIDI messages to our synthesizer application in such a way that it is convenient to the user to use the synthesizer application of their choice? Lastly, how can we synchronize our timing mechanism? The implications of a given architecture can provide us a strategic advantage so that implementation can concentrate on higher level details such as recording and performance evaluation while abstracting lower level details such as port management and hardware timers. Thus, a design decision to utilize ALSA was made for these reasons. In this section we will look at what ALSA is, and why we would want to use this architecture; however, we will look at how ALSA works in the context of our implementation in chapter 5.

ALSA or Advanced Linux Sound Architecture is an open source project originally developed in the C programming language by Jaroslav Kysela, Abramo Bagnara, Takashi Iwai, and Frank van de Pol, and it is now further developed, maintained, and updated by developers in their spare time[2]. "ALSA provides audio and MIDI functionality to the Linux

[1] http://lac.zkm.de/2005/slides/paul_octavian_nasca_slides.pdf accessed 5-25-08
[2] http://www.alsa-project.org/alsa-doc/alsa-lib/ accessed 5-25-08

operating system[3]." Among its many capabilities, ALSA supports audio interfaces, fully modularizes sound drivers, and provides SMP and thread-safe design as well as supplying a user space library to simplify audio programming and provide higher level functionality.

As we will see, ALSA specifically answers all of the questions initially brought up in the beginning of this section. Building on this architecture simplifies the creation and maintenance of ports by facilitating interchangeable physical medium connections as well as providing seamless communication between external MIDI applications such as a synthesizer application. ALSA also abstracts and manages the interaction between user and kernel memory allowing us to synchronize timing mechanisms through the use of hardware timers. From a higher level viewpoint, ALSA extracts MIDI information through a polling mechanism and then supplies this information through predefined data structures to the application. Thus, MIDI messaging functionality is provided through a useful framework for MIDI application programming.

In this section, we have described the Advanced Linux Sound Architecture and explained the design decision to utilize it in this project. ALSA abstracts the lower level details providing an ease of usuability in focusing our efforts on the construction of a performance evaluator. During our discussion of implementation in chapter 5, we will look at the specifics of how ALSA provides this functionality. It should be noted that only the aspects of ALSA specific to this project will be discussed. Before discussing the implementation of Blunote, we will look at the design of the higher level components of the project.

4.3 Overview of Design Details

As we have seen from section 4.1, there are several areas of functionality that this project needs to provide. We can divide this functionality into five main subsystems: sound generation, recording, playback, graphical user interface interaction, and performance evaluation. Each subsystem is made up of a collection of software components and several of these components may be shared by different subsystems. We will examine each subsystem and

[3]http://www.alsa-project.org/main/index.php/Main_Page accessed 5-25-08

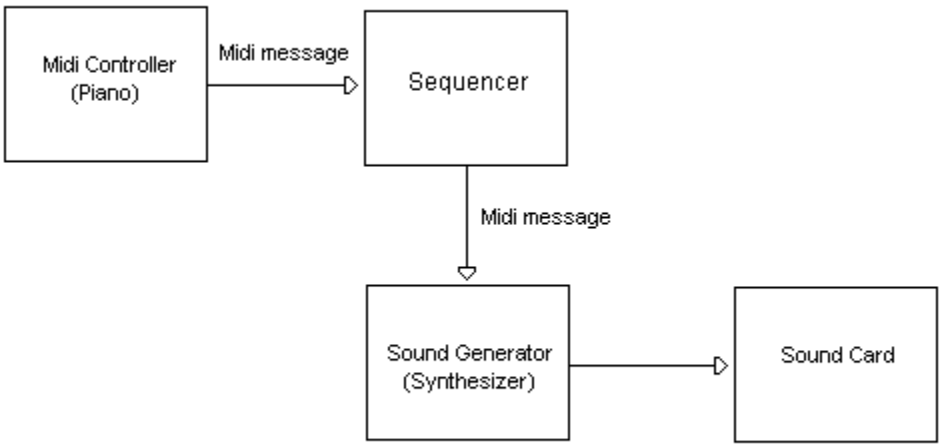

Figure 4.1: Sound generation overview

its relationship to the overall design of the project. During the course of this section, each figure represents a subsystem while each box in a figure represents a component.

The first subsystem, sound generation, is the process of generating the appropriate sound from user input, and it is visualized in Figure 4.1. This subsystem is utilized in some form by all other subsystems since we would like the user to be able to hear the musical notes generated by the MIDI controller. When the user presses a key on the MIDI controller, a MIDI message describing that musical event is sent to our application. Our application then routes the message to a synthesizer application. The synthesizer is configured at this point with customizable parameters describing how to construct waveform information from the MIDI information. For example, these parameters may describe how to construct a piano sound or a hammond organ sound. The waveform information is then sent to a sound card which generates the sound through the computer system's speakers.

Figure 4.2 and Figure 4.3 describe the recording and playback subsystems. Note that the sound generation subsystem is incorporated in the recording subsystem. In the recording subsystem, we record a sequence of musical events inputted by a user from a MIDI controller. A message is sent from the MIDI controller to our application. Our application stores the event in memory through a sequencer while simultaneously forwarding the event

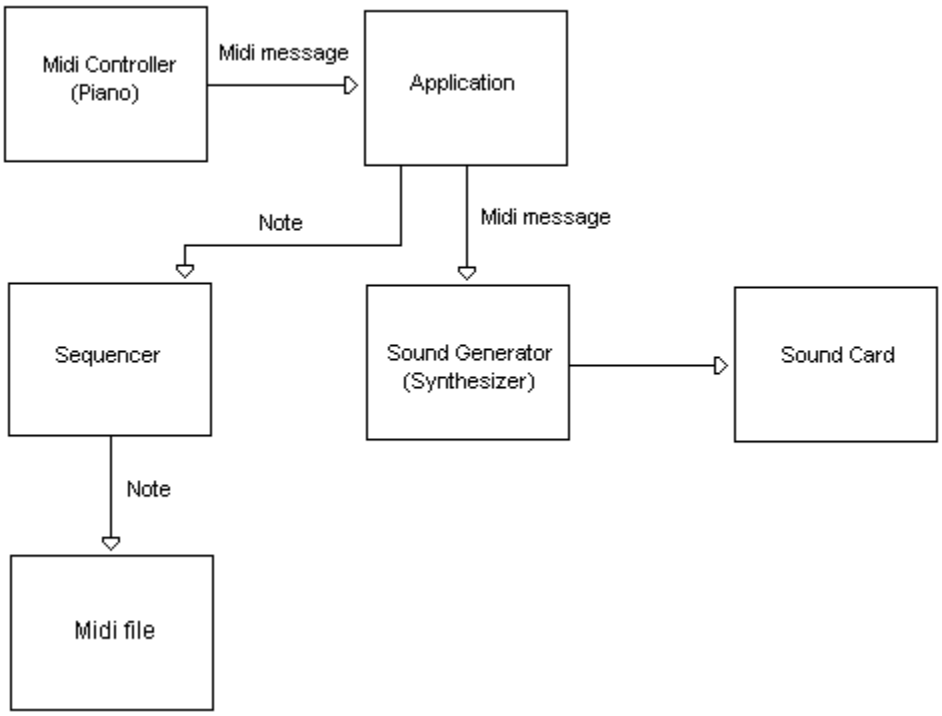

Figure 4.2: MIDI recording overview

to a synthesizer that generates the sound of the event. When the user has finished inputting musical events, the sequencer generates a MIDI file describing this sequence of events. In the playback subsystem, the sequencer mimics the role of a MIDI controller by converting a MIDI file into memory and firing off musical events at the appropriate time to the application to generate the appropriate sounds. The signifigance of these subsystems is that it provides the user a way of recording songs to be used for comparison to performance in the performance evaluation subsystem as well as listening to a recorded song for review by the user recording the song or the user previewing how a performance should sound before attempting to perform the song.

Although the GUI subsystem can be viewed as one complete system providing user interaction, there are a couple of distinguishable parts to the subsystem. Figure 4.4 describes how a user navigates through Blunote. There are four screens: main menu, the recording and playback screen, the performance evaluation screen, and the configuration screen. The user can access the latter screens from the main menu and access the main menu at any

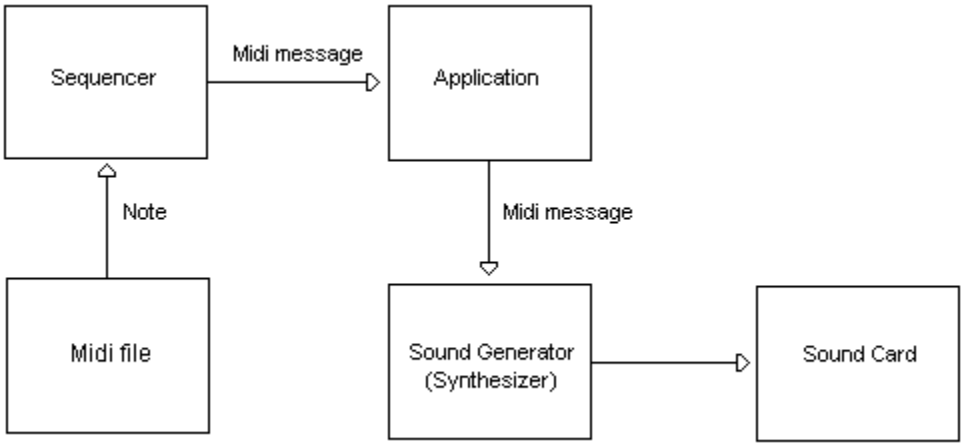

Figure 4.3: MIDI Playback Overview

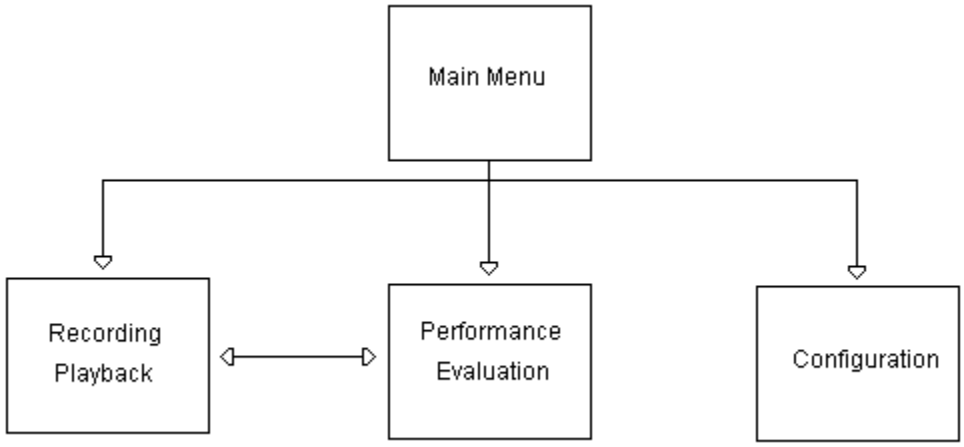

Figure 4.4: Graphical user interface overview

time. In addition the user may access the recording and playback screen or the performance evaluation screen from either screen. The sequencer component provides different functionality according to the screen being viewed and the user input on the respective screen. Therefore, as Figure 4.5 describes, the graphical user interface must communicate with the sequencer to coordinate the employment of the respective functionality.

Finally, the performance evaluation subsystem is described in Figure 4.6. During initialization, the subsystem loads the sequence of musical events into memory to be utilized by the sequencer as well as interpreting this sequence of events into a musical score displayed by the GUI. The subsystem coordinates the beginning of the song. As musical events

Figure 4.5: GUI and sequencer synchronization

are inputted by the user, the application sends out three messages: a MIDI message to the synthesizer to produce a sound, the performance note to the evaluator, and a timing message to the sequencer. Based on the timing message, the sequencer can extract the musical event that is supposed to occur at that time and relays this information to the evaluator. The evaluator can then analyze the two musical events and send the result to the GUI if necessary.

In this section, we examined the design of the five subsystems that provide the overall functionality of Blunote. We have seen that some components are shared by different subsystems, and we will examine these relationships from a component's viewpoint in chapter 5 during our discussion of implementation. The significance of the order in which the subsystems were presented is that it correlates to the order in which these subsystems should be implemented. We now have enough information to turn our discussion to implementation.

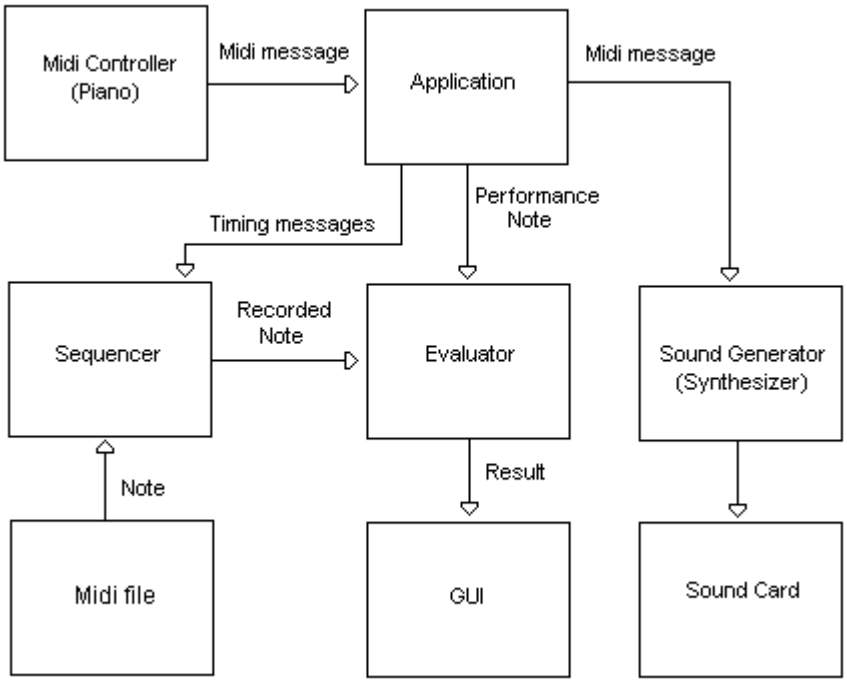

Figure 4.6: Performance evaluation overview

Chapter 5

SOFTWARE IMPLEMENTATION

Since the implementation progression follows the design progression, we will refer to the figures in chapter 4 for reference. Section 5.1 describes how to generate sound using the Advanced Linux Sound Architecture (ALSA). Section 5.2 details the sequencer development through the construction of delta times while section 5.3 describes the implementation of the recording and playback subsystems. Since the user interface is an important part of the project, section 5.4 will discuss the construction of the graphical user interface, the need for separation of these parts, as well as the resulting need for communication of both. Lastly, section 5.5 details the development of the performance evaluator.

5.1 MIDI Controller Recognition and Sound Generation

The first subsystem to implement in this performance evaluator is sound generation, depicted in Figure 4.1. Since the synthesizer will communicate with the sound card independently, there are three main components that we will investigate: the MIDI controller, our application, and the synthesizer. What we are interested in at this point is the connections of these components. Basically, we want to route information from the controller through our application to the synthesizer. The goal is not only to be able to sound a note by hitting a key on the MIDI controller, but to capture the events and validate the extracted information from the messages being sent.

As we discussed in section 4.2, ALSA can abstract ports so that we can use any MIDI controller and any synthesizer in conjunction with our application. MIDI ports are unidirectional. There is one port associated with reads and another port associated with writes. Therefore, our application needs to create two ports: one to read from the MIDI controller

and another port to write to the synthesizer to generate sound. After the ports are created, we will then need to bind our ports to the MIDI controller port and the synthesizer to generate sound. From an abstract viewpoint, ALSA uses a mechanism called "subscription" to achieve this functionality. Ports subscribe to other MIDI ports to automatically receive or transmit a MIDI stream in a way similar to piping a file on a Unix system allows an application to receive or transmit data via stdin and stdout automatically. The ALSA server actually owns the connection and then forwards the stream to the applications that are subscribing to a particular port. The significance is that since multiple applications can subscribe to the same port, we would be able to set up connections from the MIDI controller to our application as well as from the MIDI controller directly to the synthesizer. However, there are multiple reasons why this is not advantageous as we will see, especially if we want to add a metronome to the application.

From a concrete viewpoint, there are a couple of steps to implement this process. Namely, we need to create both a read and a write port through calls to the ALSA library, and then we need our ports to subscribe to the appropriate port for communication. We could explicitly subscribe to these ports or have the user input the necessary port numbers in order to make a connection through our application. However, Kaconnect, an open source program, is a convenient tool to create subscriptions. This program lists all available MIDI ports including the ones that we have just created in a graphical display that separates read ports from write ports. It then easily facilitates subscription by allowing you to connect ports by selecting one in the read list and one in the write list. Because our ports will have to be created before the connections can be made, our application will have to begin before running Kaconnect.

Now that we have established how connections are made, we can turn our attention to how our application operates in general to extract information from the messages being sent. When Blunote first starts, the sequencer component creates our MIDI ports. This component then uses a polling mechanism to obtain any information being sent from the ALSA server. It is important to note that our program is a client application of the ALSA server. If there is a MIDI event, the ALSA server sends this message in a predetermined format. Our application can then take the appropriate action based on the event described in

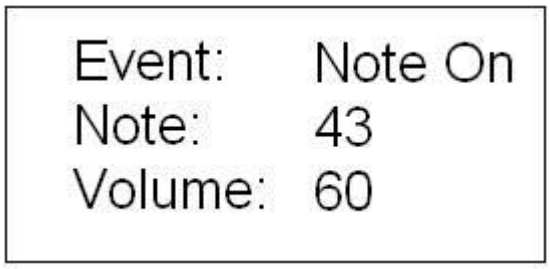

Figure 5.1: Screen output

the message. Blunote directly outputs this event to the synthesizer before taking any action based on the event type. At this point, our application has the information of the musical event and can output this information to the screen as seen in Figure 5.1.

Now, we can route messages through our application and produce sound while simultaneously extracting information from MIDI messages. This information includes the type of event taking place as well as pitch and volume values. However, one key piece of information is missing: the event's duration. Because of the nature of MIDI messages and the need for a controller to tell a sound generator to start sounding a note before knowing when it will end, our sequencer will have to derive meaning from when the message arrives. However, the duration of the note will not become important until we want to display the note visually. We now have a skeleton of a sequencer constructed, and in the next section, we will look at the timing information we need for the sequencer. We will begin to look at how we can further improve upon our sequencer into something useful by recording and playing back sequences of musical events in section 5.3.

5.2 Deriving Delta

At this point, our messages are giving us pitch and volume values for events, but we are lacking a key piece of information: timing. MIDI messages sent from a controller do not provide timing information because there are a lot of variants that go into deriving context. Therefore, it is left up to the application or sequencer to decide what contextual information is needed in this area. So, before we can look at the recording and playback subsystems, we will need to investigate what timing information we do need in this section.

There are three types of event timing information that we will eventually need: duration, delta time, and timestamps. To help illustrate the difference, look at a stream of events such as the one below in which each event is represented by a pair of values representing a certain pitch and event type.

(middle C, `Note On`)-(D, `Note On`)-(E, `Note On`)-(D, `Note Off`)-(middle C, `Note Off`)

Duration describes the length of time from a `Note On` event of a certain pitch to the `Note Off` event of that same pitch. In this example, middle C is the first message received telling the sound generator to sound that pitch. The last message tells the sound generator to turn that pitch off. It should be easily recognizable that there may be a lot of event messages in between the messages marking the beginning and the end of a note; however, once a `Note On` event has taken place, the only acceptable event for that given pitch is a `Note Off` event. This makes sense because a note must end before the same note can begin again. Delta time represents the time that has passed between the last event and the current event. In the same sequence of events, the delta time for the D `Note On` event is the time that has passed from the last event, middle C `Note On`, and this event, D `Note On`. Timestamps are the summation of delta times of the current event with every event that has already occurred since the beginning of the song.

As it turns out, we only need delta time or timestamps for recording and playback. While notes are the construction of two particular events (according to one pitch), MIDI files and MIDI streams are constructed from a series of sequential events. The playback system emulates a real-time performance. In order to do so, the only timing information required is when and how to perform the next action in a sequence. When we are routing messages from a performance, our application can not wait until a note ends before beginning to generate sound for that pitch. Therefore, there is no need to calculate duration in order to playback the sequence of events, and it stands then that for recording all we need is to calculate the delta times for each event. This is different from musical scores which need the semantics of timing information to construct musical notation. When we build a musical score for our user interface, we will then need the duration of the notes represented from the series of events.

We need delta times for each event to construct MIDI files. Certainly the first thing that comes to mind is that we could take a traditional timestamp for an event when an event is recognized and subtract the preceding event's timestamp to derive delta. Theoretically, this is all we need to record and playback a series of events; however, there are some problems. The first problem is that this is invariably going to be a very small time difference, often in microseconds, yet the possibilities that this time difference could yield increase the needed capacity of data storage. Therefore, how small or large of a measurement do we need to make? We need some way of establishing the level of measurement we would like to make or provide a time division system. Also, even though we could accurately record and then playback a performance, we would have no context from which we could either speed up or slow down the performance, which is what musicians refer to as tempo. Therefore, our delta times and timestamp representations need to reflect both the tempo and the time division value that we will establish.

A time division system essentially is how we determine the relational qualities of the measurements needed to be taken. This is an abstract viewpoint of the timing system or how measurements are related, not how they are absolutely measured. From a data storage standpoint, we would like to express our delta times in integers, so with that in mind, we would like to build a time division system in which we can easily express significant intervals. Our significant intervals include the set of subdivided timing values of notes that we would like to represent. For example, in our application, we would like to subdivide beats as small as an eighth note. So, our relations include a whole note, that is equal to two half notes, which is equal to four quarter notes, and which is equal to eight eighth notes (refer to Figure 3.7). Generally, time divisions are expressed in PPQN or pulses per quarter note. A pulse or tick, represents the smallest amount of time that we will call significant. For example, we can arbitrarily pick a time division of ninety-six ticks to represent a quarter note. Using that as a base, it would then follow that an eighth note would be forty-eight ticks, and a half note would be one hundred ninety-two ticks. Notice that a relational model such as one tick representing an eighth note and two ticks for a quarter note was not used. We need to allow for some variance in the timing values to allow for artistic expression as well as other issues of the subdivision timing values which is beyond the scope of this

project. However, a time division system using ninety-six ticks to represent a quarter note is widely used as an adequate basis for defining a time division system.

Having defined a time division system, we can turn our attention back to tempo. As defined in section 3.1, tempo is beats per minute, or bpm. This definition refers to the amount of quarter notes in a minute using a time signature of 4/4 time. However, this is opposite of the way that we would like to look at tempo in our application. It is advantageous for our calculations if we determine how much time elapses in microseconds for the duration of a quarter note.

$$tempo\ (microseconds\ per\ quarter\ note) = 60,000,000/bpm \tag{5.1}$$

In the formula 5.1, one microsecond is the same as one one-millionth of a second. A bpm of sixty represents sixty steady beats per minute or one beat second. In this example we would have a million microseconds per beat. Thinking about tempo in this fashion allows us to have a higher resolution without dealing in fractions. This becomes very noticeable when tempos measured in bpm's are a fraction. Also, in formula 5.2 we can derive microseconds per tick as a useful aid when trying to understand the relational model of PPQN.

$$microseconds\ per\ tick = 60,000,000/(bpm * PPQN) \tag{5.2}$$

Now, we have enough information in order to construct delta times and timestamps. The tempo specifies the time in ticks of a quarter note, and PPQN describes the relational qualities of ticks. Delta time then expresses the difference in ticks from the current and previous events. Timestamps are the summation of delta events from the beginning of the song. Having pitch, volume, and timing values allows us to begin building our recording, playback, and performance evaluation subsystems defined in the subsequent sections.

5.3 Recording and Playback Subsystems

Our sequencer is now operational, but it does little more than observe the events being passed in the message stream. In order to accomplish performance evaluation, we need

something in memory that we can compare to the performance. Our options are to use pre-existing MIDI files of piano performances or allowing teachers to record specific pieces for the student to learn. We will do both, and in addition, we would like the student to have the ability to hear the song that they need to perform. Therefore in this section, we will look at how to implement recording and playback systems (refer to Figures 4.2 and 4.3).

The recording process involves setting up a timer to timestamp events, a mechanism for supplying the passage of time to a user, a mechanism for storing the incoming events into memory, and then a mechanism for converting this data into MIDI files. Since we have outlined how MIDI files work in section 3.3, we will not go into a lengthy discussion on how to construct or parse MIDI files. While this is not trivial, the necessary information has already been outlined, and further discussion has no bearing on performance evaluation except as it relates to delta times. Therefore, we will concentrate on recording events into memory.

We can achieve the synchronization of MIDI events through the creation of a reliable timing mechanism to coordinate the starting and stopping of the recording process and the extraction of reliable delta times and timestamps for MIDI events. Fortunately, ALSA provides abstractions to utilize a hardware timer in the form of a queue container. By supplying the ALSA timer, or queue, with tempo and PPQN, we can start the timer when the user is ready to record, and then query the timer as events are passed to the application from the controller to obtain a timestamp or the total number of ticks that has passed since the timer was started. We can then store the event's characteristics in a container such as a list while still routing the event to the sound generator. Recording a list of events is depicted in Figure 5.2. It is worth noting that the timer resides in kernel space. After the recording process, the user can then save the song, which will initiate a data dump into a MIDI file governed by MIDI file specifications.

Although we have discussed how to synchronize a timing mechanism internally within the application, we also need to represent tempo to the user in the form of a metronome. The abstraction of the timer is a queue, specifically for the purpose of being able to add events to the timer. Events supplied to the ALSA queue, become the ALSA queue's responsibility. The ALSA queue will then fire off the events to a designated port at the scheduled time.

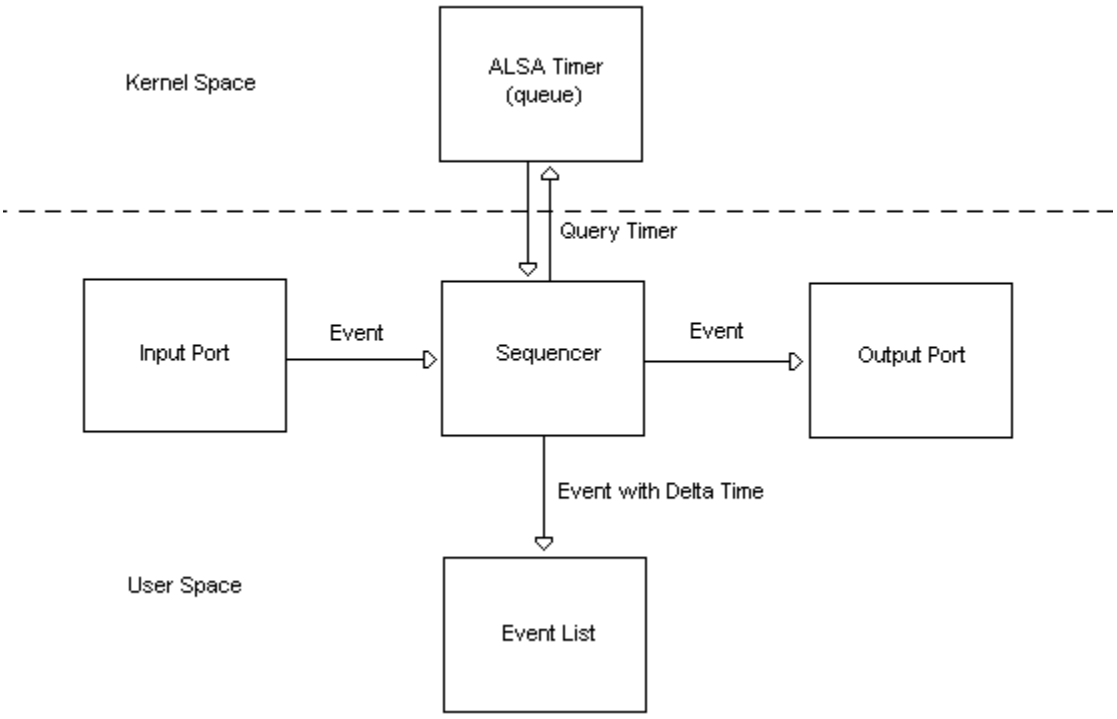

Figure 5.2: Recording a list of events

Up to this point during the recording process, the ALSA queue has remained empty as we have only been utilizing it for its timer capabilities. Now, we would like to add events that represent metronome clicks. A metronome plays a sound or a click at every quarter note placement in a song and represents tempo. Each click lasts for an eighth note in duration. It is easy enough to schedule a number of clicks on the queue, but since the user controls when the process starts and stops, we have no way of knowing how many clicks are needed. Figure 5.3 describes a way of looping the metronome events without recording the clicks in the process. The sequencer initially begins the loop by adding the number of clicks in a bar of music, and designating its' own input port as the destination. These events are sent with an event type of echo. After the ALSA queue fires the event off at the appropriate time, the sequencer can then distinguish the event by the event type, output the click to the sound generator, and then reschedule the next click by taking the timestamp of the current click and adding the number of quarter notes in a bar of music. Because the sequencer can distinguish between the click events and regular events being recorded, simple logic keeps

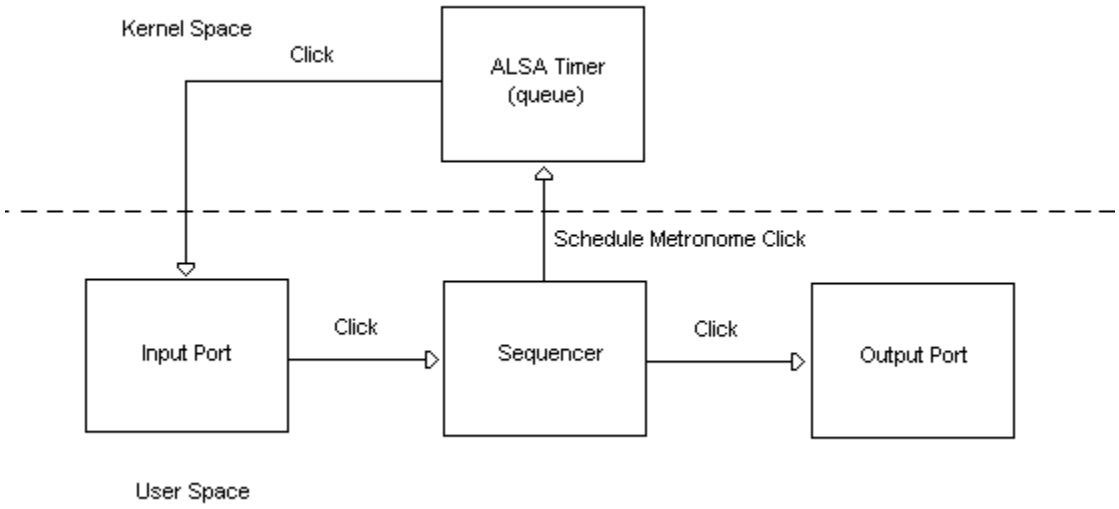

Figure 5.3: Metronome scheduling

the metronome clicks from being added to the event list. In this way, we can achieve a self maintaining loop of metronome clicks.

The playback process involves setting up the ALSA timer to fire off events that are stored in the event list at the appropriate time. At this point, we must assert that the sequencer has a sequence of events loaded into memory either through the recording process, or through the parsing process of a MIDI file. Although the metronome is important for recording, it is not needed for the playback subsystem. However, we will still need the ALSA queue. The playback process is depicted in Figure 5.4. The sequencer loads events from the event list into the ALSA queue. After playback is initiated by the user, the ALSA queue dispatches the events at the appropriate time to the applications subscribed to the output port or, in this case, the sound generator. Since the ALSA queue resides in kernel memory, ALSA only allocates so much space for a client in order to utilize the timer. Each event is allocated as a cell, and the default number of cells a client can use at any one time, distinguished as pool size, is five hundred cells or events. We can maximize the pool size up to two-thousand cells, but the problem remains if the list of events in a song is longer than two-thousand. Also, if we use the maximum space allocated, ALSA will force our application to sleep until a predetermined number of cells have been released. This is not an

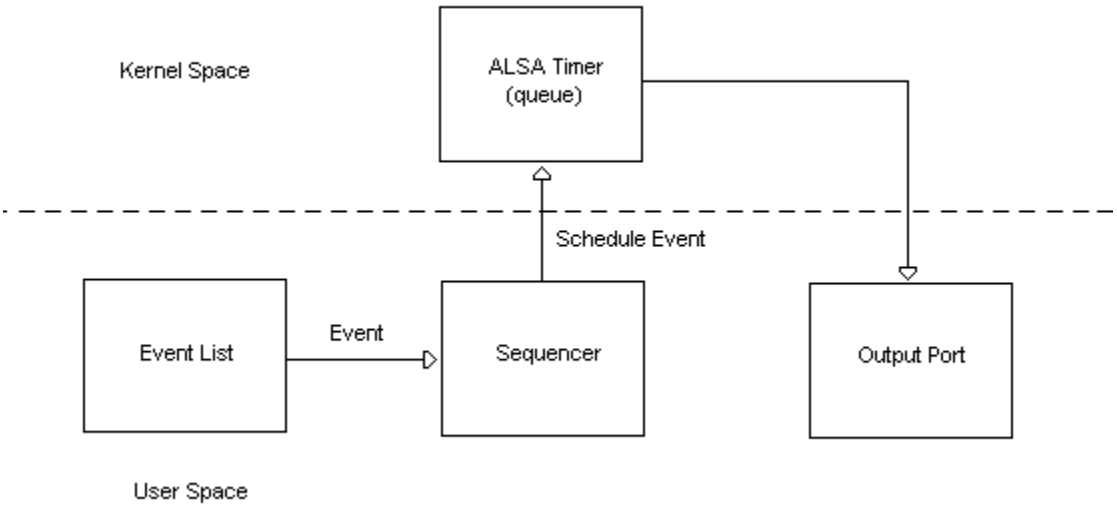

Figure 5.4: Playing a list of events

ideal situation as we would then not be able to offer the user any control of the application until the song had finished playing. Therefore, some control logic has been implemented to add portions of the event list at the appropriate times without interfering with the playback process. This problem is solved in the manner of the earlier problem of adding metronome clicks by scheduling a control message to loop back through our input port to alert our sequencer when to add the next portion of the event list.

At this point in implementation, we can offer the user audio recording and playback of original or previously recorded pieces of music as well as the ability to load and save this music in the general MIDI file format. In the process of developing these subsystems, we have developed modules such as the event list and the ALSA timer that will be needed in performance evaluation. The next implementation stage, however, will be developing a graphic user interface that will represent to the user the list of events that we have stored in memory.

5.4 Advanced GUI Implementation

Up to this point, there has been little discussion about the user interface; however, in this section, we will address the significant issues of integrating a user interface with an audio

component without becoming too involved in the purely visual details of constructing a user interface. We will begin with a quick reference to the particular graphical user interface library chosen for our application, and then discuss the need for separate audio and visual components. These components will then require communication and control logic during the course of implementation. Finally, we will look at how to calculate duration so that we can display a note, how we can display the note within the context of the song, and how to keep up with the portions of the song that need to be displayed at the correct times. After this section, we will have built all of the subcomponents that we will need in order to achieve performance evaluation.

Although any graphical user interface library could have been chosen to implement the interface, SDL was chosen for this project. SDL, or Simple DirectMedia Layer [1], is "designed to provide low level access to audio, keyboard, mouse, joystick, 3D hardware via OpenGL, and 2D video framebuffer." SDL is a popular choice for designing video games, and the feature that we are most interested in is the ability to redraw images easily and quickly; however, there are a couple of downsides to using SDL. Implementing abstractions in the ALSA library will conflict with any sound implementation instantiated from the SDL library, so we will just avoid this situation by not using any sound abstractions offered by the SDL library. Also, the library is limited in offering typical generic user interface abstractions of menus or controls other than a keyboard and a mouse, so we will implement control features through the creation of buttons and a command prompt.

There is a prevalent need to separate the visual and audio components into different threads with a higher priority given to the audio thread. While our goal is to provide immediate feedback, as a precaution, we want to insure that if our application lags for any reason, that it will do so only in the visual thread. Discrepancies in user input of musical notes can be much smaller than the visual recognition of feedback, and if the performance input is not timestamped correctly, then the visual display will be invalid. As a result of dividing the workload, the threads will have to communicate with each other to coordinate user events. Briefly, Blunote achieves control and communication through shared memory and the use of states. User requests in the user interface thread signal a change in state in

[1] http://www.libsdl.org accessed 6-18-08

the audio thread through the use of a transition matrix.

Of particular interest to this project involving the graphical user interface, is how to calculate the duration of a note needed for displaying the representation of a note. The duration of a note, as we have discussed, is the combination of two events with a specific pitch value: one `Note On` event to mark the beginning of the note, and one `Note Off` event to mark the end. Currently, our list of sequential events contains both type of events with timestamps, so one option is to implement a search algorithm that finds the next `Note Off` event in the list for a given pitch to obtain the pair of events for a given note during a conversion process after recording. An elegant search algorithm is to use an array representing the keys of MIDI controller. As a `Note On` event is discovered, a pointer to that event is inserted into the array at the location directly correlated to the pitch value of that event. When a `Note Off` event is encountered, the `Note On` event is extracted from the array, their differences in time calculated to derive duration, and the array location is reset to the default value. Since only a `Note Off` event can follow a `Note On` event for a given pitch, we are guaranteed of a direct match.

The calculations of the difference in timestamps require that each timestamp be normalized before the actual subtraction takes place. Each subdivided beat, such as a quarter note or a half note, recognized by the application has an absolute value in ticks. A note that has been recorded by a user will very rarely have an absolute value timestamp but rather a timestamp that exists in an accepted range. Our ranges are determined by the set of subdivided beats in our PPQN relational model. For each absolute value of a subdivided beat in this model, an acceptable range for an input value would be the absolute value plus or minus half of the smallest subdivided beat. For example, if we were constructing a note on the first beat of the first measure with PPQN set at ninety-six, then the `Note On` event would have a timestamp between zero and twenty-four ticks. (Twenty-four is half of the smallest subdivided beat in this PPQN model.) If the `Note Off` event has a timestamp of ninety-six ticks, then our note would be defined as a quarter note. A timestamp of forty-eight ticks would denote an eighth note. However, suppose that our timestamp for the `Note Off` event is seventy-three. The closest absolute value to our timestamp among recognized subdivided beats is a quarter note, and therefore we would treat the note as a quarter note.

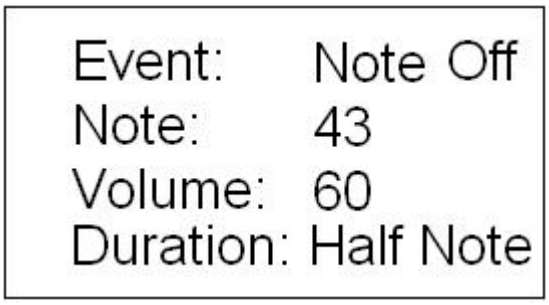

Figure 5.5: Screen output with duration

The reason to normalize before the subtraction is that if these two events occurred later in the song, an early `Note On` event combined with a late `Note Off` event could produce a time difference greater than the difference of absolute values, thereby altering the intended duration of the note. Figure 5.5 depicts the screen output of a deconstructed note. From this information, we can easily create a note image based on duration, and adjust that image on a staff according to pitch.

Now that we know how to construct a note, we need a way of putting this note within the context of a song. Our main concern is when the note takes place. As discussed in section 3.1, a bar of music is defined by the number of quarter notes in a measure. As we are only implementing four-four time, we will have a total of four quarter notes in a bar. The lowest subdivision of beats our application is allowing is eighth notes. Therefore, we have eight possible beat locations within a measure of music as depicted in Figure 5.6. Figure 5.7 illustrates that the most notes that a bar of music in our application can contain is eight eighth notes. However, it should be noted that these beat locations are merely placeholders in which any note timing value can reside as represented in Figure 5.8. Given the timestamp of any note, we can calculate both the bar number and the beat number of any note within a song. If we construct a bar object that contains an array of the number of beat locations in a measure, then we can create a display list of bars for our graphical interface in order to provide a musical score for the user. Since a song may be longer than the space we have to display a musical score, control logic in the user interface is implemented to scroll through the bars at the appropriate times. Finally, one problem remains: our interface component needs to communicate with the metronome in the audio thread to synchronize

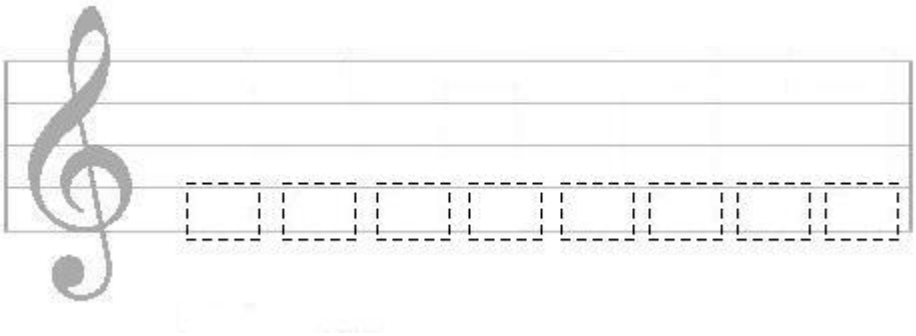

Figure 5.6: Empty beat locations

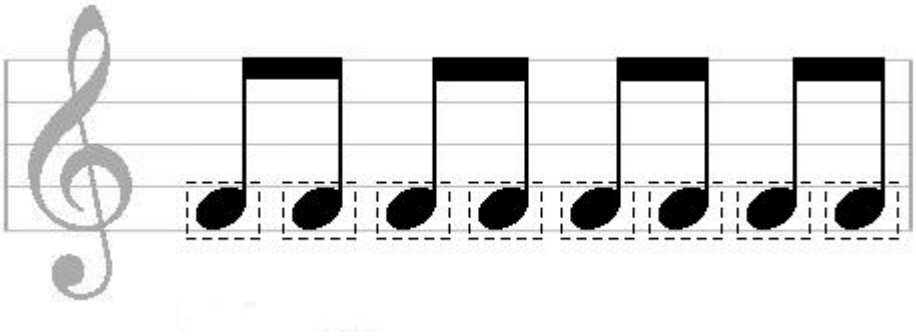

Figure 5.7: Beat locations with eighth notes

bar replacement. This synchronization is achieved through updating current bar and beat values in shared memory.

In this section, the discussion was limited to background information and parts of the graphical user interface implementation that were specific towards reaching the goal of performance evaluation. At this juncture, we are able to display a musical score to represent the musical events to a user as well as keep track of the current place within the song in the user interface. Now, we can shift our attention to our objective and look at how to compare a performance from a user to the musical score we have stored in our display list.

5.5 Performance Evaluation

As it turns out, a lot of the implementation constructed up to this point, is reusable during the performance evaluation process; however, we need to define exactly what it is that we would like to evaluate. It should be obvious by now that the most defining characteristic

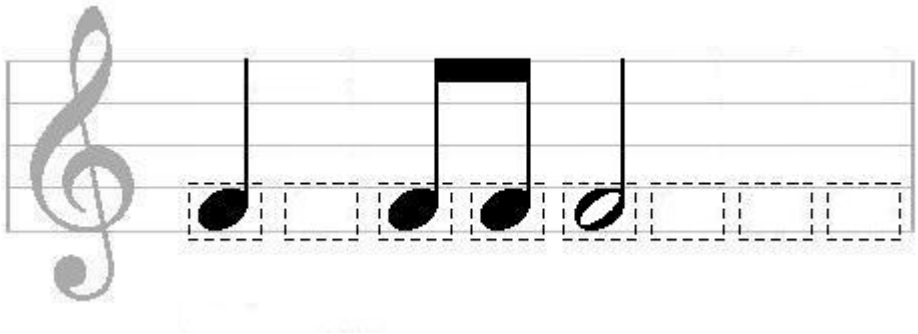

Figure 5.8: Beat locations with mixed notes

of an event is its timing information. The timing information of a musical event describes the location of the event within a sequence of events. If we use the timing information of a musical event that has been entered from a user, we can then locate the corresponding event within the song and compare the pitches of the two events to determine whether that event matches what we have stored in memory. Since we are interested in real-time feedback, our evaluation will be based on both defining events of when a note begins and when it ends. In this sense, as a performance is given, notes depicted in a musical score will turn from black to blue as soon as a match is made from an incoming `Note On` event. A succesful `Note Off` event will leave the note blue while an unsuccessful `Note Off` event will turn the note from blue to red. Therefore, by the end of the song all notes in the musical score will have changed color to either blue or red with successful notes represented by blue notes.

Although we now have a simplified explanation of what we would like to do, there are some problems. What happens if a user enters a note that is not in our display list? How do we turn a note red, if the user doesn't enter any note at that time? What if the user enters a correct `Note Off` event but not a correct `Note On` event? Also, up until this point we have only stored single events in our display list, so how do we deal with chords? Finally, how does the comparison process take place.

Over the course of this section, we will look at each question in turn, but for now we will just discuss single events stored in the subdivided beat locations of a bar. All of our notes within a song are stored within the display list in the user interface component. If we use the array alogrithm described in section 5.4 to convert two events into duration, we

will be able to do our comparisons quickly. As a `Note On` event is entered by the user, we update the array with the `Note On` timestamp. We then pack up the event and send a message to the user interface component. The user interface component can then use the message to calculate the bar and the beat in the display list to find a corresponding match. Since our interface component is advancing itself through metronome updates and the beat locations are in an array within a bar object, both of our lookup times are O(1). If a match can be made, the note image is changed to blue. If it can not be made, nothing is done. As a `Note Off` event is entered by the user, a msg is formed with both the `Note Off` event and the `Note On` event stored in the array. The array location is set back to a default value and the message is sent. The interface component then looks for a match by searching the display list for the `Note On` timestamp, and then compares pitches and durations. If the pitches do not match, nothing is done, and if the durations do not match, the note is turned to red. Therefore, only two situations can change the color of a note in the current algorithm: a note will change to blue only if it has a `Note On` event match, and only a blue note will change to red if the subsequent `Note Off` events do not match. Everything else is disregarded.

To solve the problem of turning notes red that never change to blue first, we implement control logic associated with the algorithm that updates the current position of the display list. While the ALSA timer is updating our current bar and beat values in shared memory, our interface component is simultaneously reading those values. When those values change, the interface component examines the beat locations between the previous and current beat positions. If any black notes reside in those beat locations, then we know that the user has not correctly entered those events, and we can turn them red. Notice that it does not matter at this point what durations are expressed in the beat locations of the bar. The user has missed the `Note On` event associated with this note, and therefore has missed the note. In the same manner, `Note Off` events without a correct correlating `Note On` event will just be ignored.

Finally, our algorithms handle single events in beat locations effectively, but they will need to be altered slightly to handle chords. Chords are multiple notes that occupy a single beat location within a bar. Similarly, notes played with the left hand often appear in the

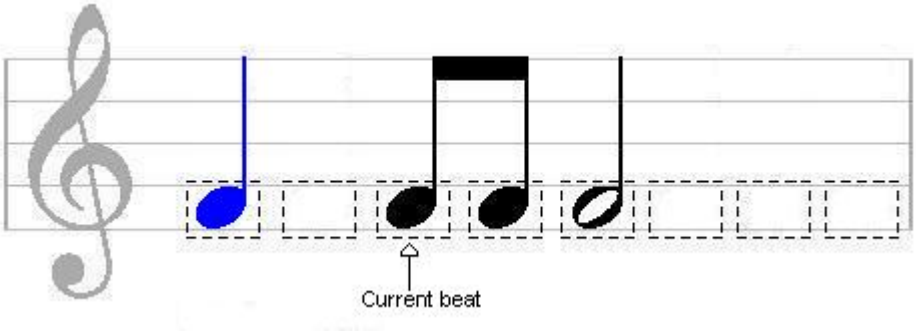

Figure 5.9: Secondary evaluation progression 1

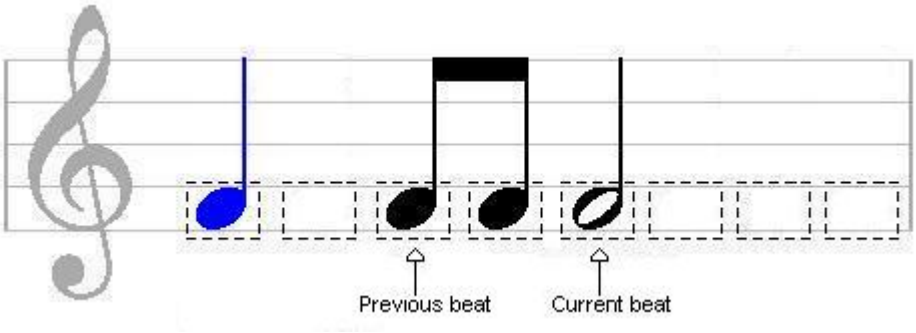

Figure 5.10: Secondary evaluation progression 2

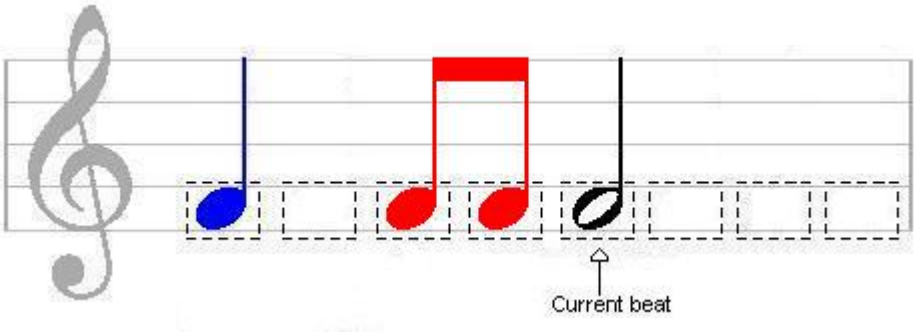

Figure 5.11: Secondary evaluation progression 3

same beat location with notes played in the right hand. There are two problems associated here. First, we have no way of knowing how many notes will need to occupy the same spot. Secondly, even if a user hits two or more notes at exactly the same time, there is no guaranteed order in which the MIDI controller will send the messages. For example a C chord composed of the notes, C-E-G, might be sent as C-E-G, or E-G-C, or any possible combination of the sequence of the three notes. As a result, the bar objects are modified to contain an array of lists, and our search times theoretically increase to O(1) for the bar, O(1) for the beat location within the array, and O(N) for the correct pitch within the list at that location; however, for all intensive purposes our lists will never be that large, therefore our search time will behave closer to O(1).

At the end of the performance, the number of blue notes in the musical score divided by the total notes in the piece will yield the percentage of notes the user entered correctly. Also, the advantage of keeping the three dimensional display list in memory is the ability to go back and look at the notes the user missed or correctly entered. Although this information is supplied to the user at the end of the performance, the main focus of this application is in aiding the musician by supplying feedback while the user is performing. In this section, we covered the performance evaluation process which included algorithms implemented in previous sections.

Chapter 6

CONCLUSION

Blunote is an introductory discussion of how to build a performance evaluator on the Advanced Linux Sound Architecture. This project seeks to offer students a chance at receiving automated, immediate feedback. As part of that feedback system, students are not just given feedback after notes are entered. They are given feedback while they are playing the notes. By breaking a perceived single event according to duration into the original construction blocks of Note On and Note Off events, students are shown immediately when they are missing a note either because a key was not pressed in time or because the key was not released at the right time. The algorithm involved in performance evaluation would have been simpler had wrong notes not changed to a red color; however, displaying red notes helps a user get a sense of the passage of time during performance. If a student loses the feel of the timing of the song's events, the steady visual change of the the color of the notes in the song should provide the student with a reinforced sense of tempo. Although Blunote evaluates performances according to pitch and timing, the immediate feedback system aids more in providing a student with support in learning the timing intricacies of music. Learning finger patterns, hand positions, and hitting the right pitch are all a part of the logic facility. Peforming these types of techniques within the proper time, however, remains a part of being able to "feel" the beat. There are plenty of exercises to help gain an intelligent understanding of tempo and the timing of musical scores, but this application is designed to help the user "feel" the passage of time while he is performing the song.

There are a couple of technical capabilities that could be added to this project. More advanced songs contain more advanced timing schemes incorporating sixteenth notes as well as triplets. In the latter case, the array representing beat locations within a bar would need considerable work to allow a significantly different timing representation. Some songs

subdivide beats into thirty-second and sometimes sixty-fourth intervals. Although used very little, this would be an interesting challenge during the recording process as it would be very hard for performers to retain the intended value of notes while offering artistic expression at the same time. Rests and tied notes, key and time signature modifications would offer more flexibility of material offered in this application without changing the core logic of the algorithms Blunote implements. From a larger and more abstract perspective, future viable implementations would include velocity evaluations, looping of segments or portions of songs, an editing capability after the recording process, quantization of notes, as well a parallax scrolling mechanism option for displaying sheet music in SDL.

Blunote was developed for beginning piano students as well as developers looking to become more familiar with MIDI technology and performance evaluators in a Linux environment. The open source environment allows future developers to modify or add to these abstractions. Our hope is to help students become better players through the emphasis of the "feel" or the incorporation of the timing characteristics of musical events.

Bibliography

Cox, J. 1984. The minimum detectable delay of speech and music. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '84*. International Jensen Incorporated, Schiller Park, Illinois, 136–139.

Danneberg, R. B., Sanchez, M., Joseph, A., Capell, P., Joseph, R., and Saul, R. 1990. A computer-based multi-media tutor for beginning piano students. *Journal of New Music Research 19,* 2–3, 155–173.

Heijink, H., Desain, P., Honing, H., and Windsor, L. 2000. Make me a match: An evaluation of different approaches to score performance matching. *Comput. Music J. 24,* 1, 43–56.

McKinnon, I. 2005. Children's music journey: the development of an interactive software solution for early childhood music education. *Comput. Entertain. 3,* 4, 1–10.

MIDI Manufacturers Association. 1996. *Complete MIDI 1.0 Detailed Specification*. La Habra, CA.

Selfridge-Field, E., Ed. 1997. *Beyond MIDI: the handbook of musical codes*. MIT Press, Cambridge, MA, USA.

Shirmohammadi, S., Comeau, G., and Khanafar, A. 2006. Midiator: A tool for score analysis in musical performances. *Recherche en Éducation Musicale 24*.

Smoliar, S. W., Waterworth, J. A., and Kellock, P. R. 1995. pianoforte: a system for piano education beyond notation literacy. In *MULTIMEDIA '95: Proceedings of the third ACM international conference on Multimedia*. ACM, New York, NY, USA, 457–465.

Tekin, M. E., Anagnostopoulou, C., and Tomita, Y. 2005. Towards an intelligent score following system: handling of mistakes and jumps encountered during piano practicing. In *Computer Music Modeling and Retrieval*. Springer, 211–219.