

INTELLIGENT MIDI SEQUENCING with HAMSTER CONTROL

A Design Project Report

Presented to the Engineering Division of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering (Electrical)

by

Levy Marcel Ingles Lorenzo, Jr.

Project Advisor: Dr. Bruce R. Land

Degree Date: August 2003

Abstract

Master of Electrical Engineering Program
Cornell University
Design Project Report

Project Title: Intelligent MIDI Sequencing with Hamster Control

Author: Levy Marcel Ingles Lorenzo, Jr.

Abstract: This project is a fusion of musical concepts and electrical engineering techniques through the use of MIDI. It provides an elegant outlet for my desire to blend my musical passion with what I have learned over the years as an electrical engineering student. A MIDI sequencer is implemented using an Atmel Mega32 microcontroller. Input control to the sequencer is performed by the movement of hamsters within a sensing unit. The hamsters control parameters for a musically intelligent MIDI output. This musical intelligence is based on newly developed musical theories to calculate future notes and rhythmic choices. Implemented using embedded C programming, these algorithms use Markov chains to create sensible, yet unpredictable melodies. This design is elaborated over three melodic lines to produce a musical output of three-voice polyphony. The final product is a MIDI sequencer that successfully generates intelligent melodies based on the movement of six hamsters.

Report Approved by

Project Advisor: _____ Date: _____

Executive Summary

This project was initially fueled by the desire to explore the MIDI protocol. It was decided that this would be accomplished by building a MIDI device. I also aimed to make something novel that had never been done before. But to balance out the unusual nature of its design, I wanted to also to create something that was very musical.

After much consideration of different technical design aspects and contemplating various musical ideas, I was able to arrive at a project that would fulfill all of my musical and engineering goals.

An intelligent MIDI sequencer was designed with hamster control. The MIDI sequencer intelligently produced melodies by manipulating the musical elements of rhythm and note-choice. Guided by inputs based on hamster movements, Markov chains were used to perform such beat and note computations. In culmination, 3 simultaneous voices were produced spanning 3 octaves and 3 rhythmic tiers. Each voice was controlled by two hamsters: one that was responsible for adjusting the rhythmic qualities of the melody and another that modified the note sequence. With all of these elements in combination, an output was produced with very musical qualities.

All of this was implemented using an Atmel Mega32 microcontroller, distance sensors, a HamsterMIDI Controller, and 6 hamsters. Embedded C programming implemented the algorithms and computations within the sequencer.

Overall, this project was successful. The control between the hamsters and the musical intelligence turned out very well. The music sounds as good as I imagined, and I am very satisfied with the outcome of my design experience.

Digital video and audio file examples of the output can be found at the project website:

<http://www.nbb.cornell.edu/neurobio/land/STUDENTPROJ/2002to2003/lil2/>

TABLE OF CONTENTS

Intelligent MIDI Sequencing with Hamster Control	
Abstract.....	2
Executive Summary.....	3
Introduction	5
Design Requirements	5
Background	6
Range of Solutions	7
Sequencer Hardware.....	7
MIDI Controller/Input Devices	7
Musical Design.....	8
Note-choice	9
Rhythm	11
Making Music	11
Project Ideas.....	12
Final Project Definition	13
Design and Implementation	14
Sequencer INPUT Design: Hamster MIDI Controller.....	14
Sequencer OUTPUT Design: Intelligent Melodic Synthesis	14
Note-choice Design.....	15
Rhythmic Design.....	16
Creating Intelligent Melodies	21
Hardware Implementation.....	24
Software Implementation	25
C Code.....	25
Functions.....	25
Overall operation	26
Debugging.....	26
Results.....	27
Conclusion	29
Acknowledgements.....	30
References.....	31
Appendix A – Glossary of Musical Terms Used.....	32
Appendix B – Hardware Schematics and Diagrams.....	34
Appendix C – Hamster MIDI Controller Photographs.....	36
Appendix D – Markov Transition Matrices.....	40
Appendix E – Source Code – hamsterMIDI.c.....	45

INTRODUCTION

I aimed to complete a project where I could combine my interests as a musician and as an electrical engineering. Through MIDI, I would be able to implement an interesting musical design by means of engineering techniques. By creating a MIDI sequencer I fully learned about the MIDI protocol. In order to make this sequencer unique, I implemented an interesting input device to control a musical output. As a musician, I designed algorithms to produce a sensible, unpredictable output, and as an engineer, it was my job to achieve such designs. Although difficult to quantify, a requirement for a melodic output was tested and verified by simply listening to the music.

DESIGN REQUIREMENTS

There were three main objectives that inspired me to arrive at my final project design. The first and most basic objective was to create something that merged my creativity as a musician with my skills as an electrical engineer. MIDI has proven to be quite significant in digital and computer music and would be the perfect medium to accomplish my first goal. Thus, the second objective was to learn about the MIDI protocol by creating a MIDI device. Thirdly, I wanted to create something unique by implementing a MIDI design that was innovative on both the physical and musical level.

To create a novel MIDI device, I will use a unique input device to control a creative musical output scheme.

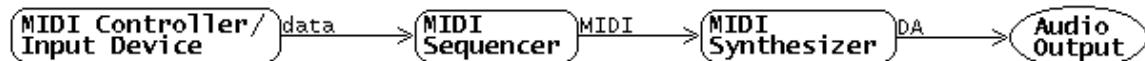
The requirements for this project are summarized:

- Learn about MIDI by designing a MIDI device
- Implement a unique, creative input device
- Design an output of the sequencer that sounds musical

BACKGROUND

What is MIDI?

MIDI – the *Musical Instrument Digital Interface* is a standardized communication protocol used to send and receive musical data between parts of digital music system. A block diagram of a standard MIDI system is shown below:



The primary blocks are the MIDI Sequencer and the MIDI Synthesizer. The sequencer creates MIDI commands and outputs them via a MIDI stream. The Synthesizer takes in this MIDI stream and decodes the MIDI commands.

For example, a simple electronic piano keyboard may act as a MIDI controller. If the user presses the note middle C (C4), the sequencer is notified that this note has been pressed and the sequencer creates the appropriate MIDI command. It is then sent to the synthesizer, which then outputs a digital audio signal with the appropriate frequency.

The overall operation of a MIDI system is as follows. A controller manipulates the sequencer to produce MIDI events in as desired way. The sequencer then sends these MIDI commands to the Synthesizer. The commands are processed by the synthesizer, which accordingly produces digital audio (DA) to be sent to an audio output.

MIDI data is a unidirectional asynchronous bit stream, transmitted at 31.25 kbaud. Ten bits are transmitted per byte – one start bit, eight data bits and one stop bit. MIDI commands typically come in the format of a status byte followed by one or two data bytes. Using the previous example, the MIDI messages that would be sent given that middle C was pressed at a volume of 100 are as follows:

- First, the status byte *0x90* (note-on command),
- Second, data byte 1 - *0x3C* (indicating which note: C4),
- Third, data byte 2 - *0x64* (indicating a velocity of 100).

When C4 is released, a note-off status byte (*0x80*) is sent, followed by the appropriate data bytes indicating that C4 is to be turned off.

More information on the MIDI standard and complete MIDI specifications can be found at <http://www.midi.org>.

RANGE OF SOLUTIONS

From the beginning, I decided that the project would be to create some sort of MIDI sequencer. I would be responsible for creating and transmitting raw MIDI data, allowing me to fully learn and exercise the MIDI protocol. Beyond this, several design decisions needed to be considered for the complete implementation of the sequencer. First, the hardware that would implement the sequencer itself needed to be determined. Next, unique input devices had to be chosen. Thirdly, the type of MIDI output stream had to be musically designed. Finally, all aspects of the sequencer needed to combine to create a unique and interesting MIDI device.

Sequencer Hardware

Already having a particular interest in microcontroller programming, I realized that I would be able to output a MIDI stream via the UART feature on the Atmel 8515 and Mega163 chips that I had become familiar with in the ECE476 course. However, I finally chose the more readily available, Mega32. Even though I had never used this chip, its operation was very similar to the Mega163 and it had all the I/O Ports, USART, memory, and interrupt features that I needed to implement the sequencer.

MIDI Controller/Input Devices

With the intent to interface the sequencer to real-world motion, I considered various input devices that could potentially control the sequencer. Such devices were distance sensors, photo-reflectors, flex sensors, force sensors and stretch sensors. These were narrowed down to the FLX-01 flex sensor and the Sharp GP2D02 IR distance sensor, simply because of their appealed as more interesting devices. The FLX-01 acted as a variable resistor whose resistance was dependant on the degree of flex. This was relatively easy to use in producing a variable voltage that would then be inputted to the Mega32 via the ADC. The GP2D02 produces a byte that reflects the distance of an object within a given range. Both sensors proved to be worthy control devices, however I chose the GP2D02 as a more versatile, and interesting device. It was convenient to use because it returned ready-to-use distance byte.

Another consideration was who or what would control the input devices. Some ideas for this were humans, animals, or nature's elements such as wind or water. Human control was eliminated because of the element of predictability and the deliberateness of

one's own movements. At this time, it was simple happenstance that I owned 9 hamsters. Thus, I decided to use this convenience to facilitate the design process. Utilizing a high number of such animals would introduce an element of randomness, as well as novelty, that I desired for the control unit.

Musical Design

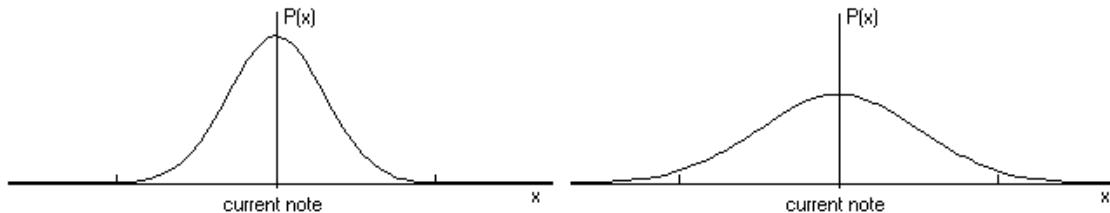
The function of the sequencer was to act as an artificially intelligent melody generator with an involved input control scheme.

Related to sequencer function, another issue was how input events should map to the musical output. One possibility is a discrete mapping such that a given input event directly corresponds to a unique musical output event. For example input A would be directly mapped to middle C being played, while input B would be directly mapped to the D above middle C. Another possibility is a more abstract musical mapping where an input event would call upon an algorithmic grouping of musical events. For example, input A would choose a note from a given group of three notes. And then for an input B adjacent to input A, the output would play a note that is chosen from a group of four notes. Such a grouping style of output events can also be realized via groupings that include weighted probabilities amongst its members. An input A could heavily weight the first member of an output group creating a higher probability that it will be chosen, while an input B could shift that weight to the second member in the group. Regardless of mapping-type, the input to output relationship still needs to be logical and easily perceivable to the user (listener/observer). He or she should be able to notice the difference between different input situations and accurately assign them to what he hears in resulting music. With an obvious potential to yield more fascinating musical results, the abstract mapping type was designed for this project.

Note-choice

There were two main considerations to implement abstract note mapping. The first was note-choices based on normal probabilistic distribution. The set of notes to choose from would be arranged with regard to consonance as a function of the absolute distance between notes. Consonance decreased as the distance increased, with perfect consonance resulting from a distance of zero. Thus if note A was two spaces away from note C and note B is 1 space away from note C, then note A is less consonant than note B with respect to note C. A current note is in the center of its distribution and has the highest probability of being chosen, due to that fact that a note is perfectly consonant with itself. Inputs to such a scheme would adjust the width of the bell curve of the normal distribution. In statistical terms, the standard deviation is adjusted. With a wider bell, it is more probable to jump to notes further from the center producing more frequent dissonance. However with a narrower bell curve, the closer notes, which are more consonant, are much more heavily weight than the further dissonant ones. Figure 1 shows this.

Figure 1 – Normal note-choice distribution



In planning, this concept seemed like a good scheme. However, when implemented it did not sound as good as I would have liked. The main weakness of this was coming up with an arrangement of notes based on a gradient of consonance and dissonance. This was very difficult to accomplish, and I was not able to find an arrangement that satisfied me. It is beyond the scope of this project to map human perception of consonance and to find an algorithm for it.

The next attempt for note-choice was Markov chains¹. A Markov chain uses a more deliberate probability scheme to determine the next note based on a present note. Given a present note A, probabilities for each of the notes that could possibly follow are assigned, producing a probability vector associated with present note A. A matrix made

¹ <<http://peabody.sapp.org/class/dmp2/lab/markov1/>>

up of all the probability vectors for each possible note can then be constructed to be used

Figure 2- Markov		Next Note		
		A	B	C
Current Note	A	.5	.25	.25
	B	.1	.4	.5
	C	0	.75	.25

as the general structure to determine the path of a Markov chain. Figure 2 shows an example of a Markov transition matrix. If the current note is A, then the probabilities are .5 that it will go to A again, .25 that it will go to B and .25 that it will go to C. If it does go to C, then C becomes the new current note. Now there is a .75 chance that it will go to B next, and a .25 chance that it will go to C again. It will never go to A because the probability is zero.

With this scheme, I was able to produce very sensible, yet still somewhat unpredictable note sequences by setting appropriate probability terms. I soon realized that the resulting note sequences could be adjusted based on the terms of the matrix. Thus inputs could control the path of a Markov chain by altering the probabilities, producing noticeably different sequences. Due its simplicity and versatility, this is the scheme that was chosen for note-choice.

The next design issue for note-choice was defining the range of possible notes that could be played. The sequencer could either choose from the full tonal span of notes or it could choose from a subset. It would be beyond the scope of this project to design intelligence based all possible notes that resulted in a sensible output. Thus, the sequencer will work with a limited subset of notes. There are many logical possibilities for this set of notes. To illustrate this, let C4 and C5 be lower and upper limit of this set, respectively. A chromatic set would include all 12 half-steps in between. A diatonic pitch set would only include the tones that belonged to a set major or minor scale. A pentatonic set would only include notes from a pentatonic scale within that range. Chromatic and diatonic sets proved to have a higher degree of dissonances that occurred, which I disliked. However, due to the general consonance that exists among the members of a pentatonic scale, I chose this as the pitch set for this project. To put an specific range, I have chosen ten entries from the C major scale spanning from G3 to E5. This deliberately extends above and below C5 and C4, respectively, because melodies often transcend more than one octave of a given pitch set. The pitch set is shown in Figure 3.

Figure 3 – C major pentatonic scale from G3 to E5



[G3 - A3 - C4 - D4 - E4 - G4 - A4 - C5 - D5 - E5]

Now that the note set is defined, a scheme for movement between notes must be planned. This system for note-choice will be discussed in greater detail in the *Design and Implementation* section.

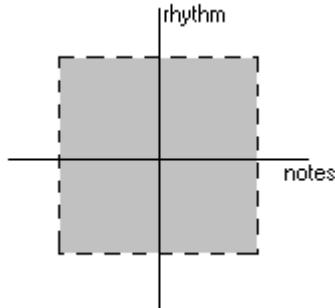
Rhythm

Initially, I planned rhythm to stay constant, producing a melodic line with even time intervals. I was not satisfied with monotony of this and soon realized that another dimension could be added to the music if some scheme for rhythmic variation was employed. The same designed decisions that were applied to note-choice are also applied to rhythm. The abstract rhythm mapping scheme is chosen. Rhythm will also be calculated via Markov chains. As with note-choice, a rhythm choice is kept to a definite set of rhythms. While the use of Markov chains for note-choice has already been explored by many sources, application of Markov theory towards rhythm does not seem to have been documented. A complete detailed discussion of my rhythmic theories and ensuing rhythmic design is included in the forthcoming *Design and Implementation* section.

Making Music

Creating an interesting musical output can be accomplished through infinite possibilities of manipulations of musical elements, such as note-choice, rhythm, volume, tempo, harmony and timbre. However, a great jazz drummer once said that music can be made through an elaboration of simple concepts. Believing in this myself and harnessing the musical complexity of this project, I have focused only on the elements note-choice and rhythm and develop them to create a larger musical form. As is shown in the previous paragraphs, simplicity is a key factor in musical design

Now that both a note-choice and rhythm can be fully manipulated, they can be combined to create music that can be controlled in two dimensions. This 2-D space for musical creation can be visualized in the figure below



Even though constraints are placed on the range of the two elements, combining sensible schemes for rhythm and note-choice can produce a high number of musical results.

Project Ideas

From all the possibilities for input and output schemes, different projects ideas came to mind. After experimenting with the flex sensors, I got the idea of creating *Digital Wind Chimes*. This could be implemented by using a 2-D array of flex sensors each with a note assigned to it. The array would then be mounted upside down with weighted tabs attached to each flex sensor. As the wind blows the tabs, the flex sensors are bent and the assigned notes are activated. Along the same lines as the digital wind chimes, I thought of creating *Fish Chimes*. After experimenting with IR sensors, I thought that it would be interesting to attach sensors to the sides of a fish tank that would be activated whenever a fish swam by. Each sensor would then be assigned to a specific note creating the net effect similar to wind chimes, but based on fish activity. Both of these projects could be completed using the Mega32 and MIDI. However, although interesting ideas, these projects were not chosen because of the lack of musicality incorporated in the output. Note choices are essentially random, and there is no rhythm involved. Finally I decided to use animal movement in a more complex way to control a musically intelligent output. This gave birth to my final project design.

Final Project Definition

The definition of my final project concept is derived from the combination of all of the above design decisions. The Mega32 will used to create a MIDI sequencer that will be an intelligent melody generator using an abstract mapping system. Input control will be done using distance sensors to monitor hamster movement. Markov chains will be used to calculate note-choices and rhythms which are combined to produce melodies. This project description satisfies all of my project objectives. The next section will discuss all the details of my design followed by its implementation.

DESIGN AND IMPLEMENTATION

The block diagram the intelligent MIDI sequencer with hamster control is shown below:
Figure 4 – Project Block Diagram



Sequencer INPUT Design: HamsterMIDI Controller

Hamsters are the control elements for the sequencer. The exact number of hamsters that would be used was not decided until the musical design was complete. For reasons that will be discussed in the following section, I decided to use six hamsters. The hamsters are constrained to only move within a sensing unit, pictured in Appendix C. Each hamster is placed inside a narrow track where they can only move in a lateral motion. A hamster's position along this path is tracked by a GP2D02 distance sensor. Based on the effective range of a distance sensor plus the typical hamster length, I decided that the tracks are to be 36" long. Measurements of one of the larger hamsters resulted in the 3" width and 4" height dimensions of each of the tracks. This allowed for the hamsters to comfortably walk, sit-up, and turn around while still maintain position along the track in the sight of the distance sensor. There are six tracks, each equipped with a distance sensor that produces a data byte representing the distance between the sensor and the hamster. The sensing unit stands upright and the tracks are oriented vertically to facilitate more active observation by the user. Additional pictures of the Hamster Controller are located in Appendix C.

Sequencer OUTPUT Design: Intelligent Melodic Synthesis

The sequencer will act as an artificially intelligent music maker which continuously outputs melodies. A melody is defined by a rhythmic succession of single notes (<http://www.m-w.com>). Using Markov chains, the sequencer will intelligently calculate note sequences and rhythms that will combine to produce the melodies. Determined independently, the combination of note-choice and rhythm yields a wide array of melodic results.

The musical design concentrates on the concept of melodic motion. The motion of the melody can be interpreted as the perceived musical pace that the listener feels. A melody's motion can be regulated by the span of its notes and/or tension of its rhythms. The following sections of *Note-choice design* and *Rhythmic Design* aim to describe a scheme to incrementally manipulate both melodic dimensions

The application of Markov chains¹, which is discussed in both sections, seems very appropriate for modeling music. A Markov chain uses weights to guide a path through a set of elements. However this is only a guide; there is still an element of uncertainty involved with what sequence will come about. This notion of bounded uncertainty is very natural within music. Music theory assigns rules regarding how music should move or where it should go next. However, music is not a strict, predictable model, but is in fact an art form, allowing it to have infinite possibilities. Yet these possibilities are guided by general rules of music theory. After all, music theory was developed to make sense of the vast domain of music. It sets rules that are very often broken, yet with the preservation of musicality.

Note-choice Design

The possible notes that could be played are confined to a 10 note set of a C pentatonic scale between G3 and E5. To design a scheme for moving from a current note to the next note, I use a Markov chains combined with what I refer to as *windowing*.

Consider the following Markov transition matrix for 5 sequential notes:

M1		Next Note				
		A	B	C	D	E
Current Note	A	1	0	0	0	0
	B	0	1	0	0	0
	C	0	0	1	0	0
	D	0	0	0	1	0
	E	0	0	0	0	1

For any current note, the only possible next note is itself. This note sequence has zero room to move and thus has a window size of zero.

¹ <<http://peabody.sapp.org/class/dmp2/lab/markov1/>>

Next consider the adjusted transition matrix:

M2		Next Note				
		A	B	C	D	E
Current Note	A	.5	.5	0	0	0
	B	.33	.33	.33	0	0
	C	0	.33	.33	.33	0
	D	0	0	.33	.33	.33
	E	0	0	0	.5	.5

The maximum step that a current note can take is 1 note up or 1 note down. Thus, its window size is 1. Window size can be increased to 2 by adjusting the previous matrix:

M3		Next Note				
		A	B	C	D	E
Current Note	A	.33	.33	.33	0	0
	B	.25	.25	.25	.25	0
	C	.2	.2	.2	.2	.2
	D	0	.25	.25	.25	.25
	E	0	0	.33	.33	.33

Applying this concept to 10 notes and spanning a from a window size of 0 to 9 is shown in Appendix D

With a smaller window size, a current note is restrained to only moving to nearby notes within the note set. Conversely, a large window size allows current note to opportunity to jump a larger interval. It is important to mention that even with a large window size, a note can still jump to itself or to an adjacent note. Larger windows include smaller windows and simply offer notes the opportunity for jumps. This is essentially a loosening of the constraints on the note-sequence. Because, a pentatonic pitch set is being used, I do not focus on specific, single note relationships. A pentatonic set's consonant nature allows agreeable note sequences with almost any combinations. Thus, this sequencer focuses on melodic motion through the manipulation of the window that notes are allowed to span. Input control determines the window size.

Rhythmic Design

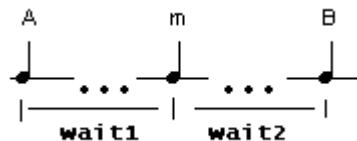
As a percussionist, I am constantly internalizing and contemplating the notion of rhythm. I often ask myself questions such as “Why are beats 2 and 4 so important in western popular music?” and “Why does music seemed to move so much more when it suddenly goes into double time, but stays mathematically the same?” In response to these questions, I have developed and implemented a scheme for rhythmic intelligence

based on principles that I call *rhythmic consonance* and *rhythmic dissonance*. More commonly associated with pitch and harmony, *consonance* refers to a sense of accord and resolve with respect to the auditory relationship between tones. Conversely, *dissonance* refers to a sense of discord and tension. Such concepts may also be applied rhythm. However, rather than being created through psychoacoustic perception of frequencies, consonance and dissonance are a product of time and relative space between events.

Take the example of constant beats on a drum, spaced about 1 second apart. After a given beat *A*, the listener is simply left to wait for the next beat *B* to arrive 1 second later. When *B* arrives, the waiting is satisfied. Thus, the anticipation of beat *B* is prepared by the previous beat, *A*. The time between *A* and *B* creates a tension which is finally resolved when *B* is heard. If *A* and *B* were spaced further apart, more tension would result. It is a general rule of thumb that a longer time interval between beats results in a greater sense of tension.

This tension can be reduced if a beat *m* is placed somewhere inside of that 1 second space between *A* and *B*. The waiting is somewhat satiated through the sounding of another beat. However, the anticipation of the beat *B* still remains. This is because *B* is what is primarily expected as the next note and not only until beat *B* arrives is the waiting completely satisfied. This new intermediate beat *m* modifies the sense of waiting and the feeling of tension between the main beats *A* and *B*. This beat *m* now initiates a new wait that is shorter than 1 sec and, the sense of tension in waiting for beat *B* is less. Beat *m* now acts as the preparation note for the arrival of beat *B*. Let wait1 be the wait between *A* and *m*, and let wait2 be the wait between *m* and *B*:

Figure 6 – Two-note Case



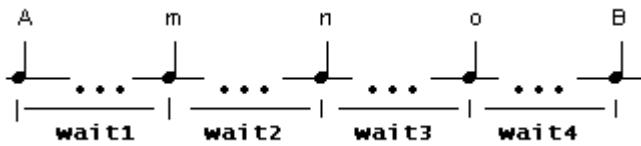
It is important to compare the tension from wait1 with the tension from wait2. Because beat *B* is what ultimately resolves the overall tension, it is the space that immediately precedes *B* that is most important. Thus the overall role of second tension is more crucial than the first.

Analysis of the effect of beat m begins with the absence of m . The tension leading up to B corresponds to the total time, 1 second. This total wait without beat m can be considered as wait2 since it leads to B . If m appears immediately after A , wait2 is slightly decreased and wait1 is very short. As beat m moves away from A , wait2 shortens and wait1 lengthens. When m passes the midpoint between A and B , wait2 is now significantly shorter than the total wait, creating a reduced sense of anticipation of beat B . Now that wait1 is longer than wait2, this secondary tension from wait1 becomes more prominent. Finally when m is very close to B , the resolution into B from m is minimal. However, a large amount of tension from wait1 is still produced.

Next, It is essential to compare the effect of m coming a time d after A , with the effect of exactly opposite case: m coming d before B . Wait1 and wait2 are perfectly exchanged. Which case produces less tension? The answer is the latter case: m coming d before B . This is because of the idea that wait2 is more significant than wait1. The resolve into B is the most important factor and is the least with the latter case. To achieve the lowest level of cumulative tension, wait1 and wait2 must be balanced by putting m exactly at the midpoint between A and B . This concept of rhythmic tension brings the definition of rhythm consonance: It refers to the level of tension produced between events. If the tension is relatively low, then the time between two primary events is said to be rhythmically consonant. Thus, given an intermediate note m , the optimal level of rhythmic consonance occurs when m arrives at the midpoint between A and B . Placing m anywhere else, or even completely omitting it, results in a situation that is less rhythmically consonant, or more rhythmically dissonant. Also, the feeling of less tension and thus less waiting results in a sense of greater motion, or rhythmic flow.

This rhythmic situation of 1 beat coming between 2 main beats is known as the *two-note case*. It is called this because A and m are effectively the only two notes at hand because when B arrives it immediately becomes the next A . This two-note case can easily be translated to a three-note or four-note case. Due to ease of the mathematical adaptability between 2 and 4, the four-note case is will discussed.

Figure 7 – Four-note Case

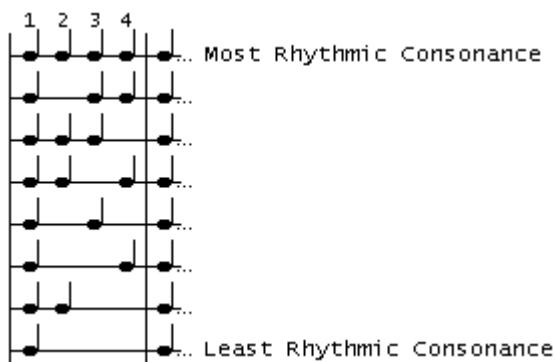


Performing a similar analysis as the two-note case, it will be concluded that placing the intermediate notes *m*, *n*, and *o* at .25 sec, .5 sec. and .75 sec produces the most rhythmic consonance.

Let us compare the optimal two-note case with the optimal four-note cases. The four-note case can be viewed as 2 sequential two-note cases where *A*, *n* and *B* are the main beats, and the intermediate beats are *m* and *o*. Alternatively, the two-note case can also be viewed as 2 sequential two-note cases without any intermediate beats. The four note case can be concluded as more rhythmically consonant because those additional intermediate beats reduce the tension, while they are absent in the two-note case. Such relationships are purely relative and can be expanded over powers of 2 to eight-note cases, sixteen-note cases, etc. To keep within a reasonable musical scope for this project the four-note case will be the main focus.

The next issue developing a scheme that ranks rhythms based on rhythmic consonance. These rhythms will stay within the four-note case frame work, and will only be changed by removing or keeping some of the 4 notes. Via similar analysis, the following scale has been derived to depict consonance levels

Figure 8 – Rhythm Scale



The following are the rules for creating the scale:

- Beat 1 always happens.
- Having more beats between the primary beats is more rhythmically consonant than having fewer notes.
- The tension before the arrival of the primary beat is most important.

Now that a rhythm scale has been established, the method for movement between the elements must be designed. Similar to the scheme used in the note-choice design, Markov chains and windowing are utilized.

Unlike note-choice, rhythm is not calculated one note at a time. When a drummer creates a beat, he does not play a note and then begin to think about when the next beat will come. Rather, he conceptualizes the rhythm as a whole before he plays it. This is because beat calculation sometimes needs to go backwards in time. For example, the presence of a beat J might call for another beat I to happen before it in order to prepare beat J , and to reduce the tension in waiting. If individual beat calculation occurs on the fly, it would be impossible to go back in time after already playing beat J . Referring back to the rhythm scale in Figure 8, if beat 3 has happened it sometimes wants beat 2 to have happened also to prepare beat 3. This must be planned well in advance of when beat 3 occurs. The solution is to determine the entire rhythmic figure before beat 1 happens. If desired this concept may be extended to more beats. Again to limit the musical scope of this project, this sequencer only looks forward 4 beats for rhythm calculation.

Markov transition matrices are also used for this calculation. With this application ‘current note’ is replaced with ‘latest scheduled beat’, and ‘next note’ is replaced with ‘other scheduled beat’.

M4	Other Scheduled Beat			
	1	2	3	4
Latest	1	1	0	0
Scheduled	2	0	1	0
Beat	3	0	0	1
	4	0	0	0

For purposes of this project, beat 1 will always happen to preserve the identity of the main beats. This is the first ‘scheduled beat’. Based on the above matrix, beats 2, 3, or 4 will never be scheduled. Consider a modified matrix:

M5	Other Scheduled Beat			
	1	2	3	4
Latest	1	.25	.25	.25
Scheduled	2	0	1	0
Beat	3	0	0	1
	4	0	0	1

After beat 1, there is equal probability for each of the 4 beats to become scheduled next.

If beat 3 is chosen, then beat 3 becomes the ‘latest scheduled beat’. With the above matrix, only beat 3 would be chosen after beat 1 because it is guaranteed to go to itself.

Finally consider the further modified matrix:

M6	Other Scheduled Beat			
	1	2	3	4
Latest	1	.25	.25	.25
Scheduled	2	0	1	0
Beat	3	0	.5	.5
	4	0	0	1

If beat 3 is scheduled there is a .5 chance that it will go to beat 2 and a .5 chance that it will go to beat 4. If it goes to beat 2, then the final calculated rhythm will be beat 1, beat 2, and beat 3. To fully perform the beat calculation, the Markov transition matrix operation is iterated 3 times to give opportunity for beats 2, 3, and 4 to be considered. If beat 1 was not guaranteed, then 4 iterations would be required.

Finally such beat calculation incorporates windowing over the elements of the rhythm scale. The first matrix shown, M4, corresponds to the solely beat 1 rhythm with least consonance. The matrices implemented for the entire rhythm scale are shown in Appendix D. The largest rhythmic window occurs in the matrix for the most rhythmically consonant case. Similar to the note-choice windows, the larger rhythmic windows also allow the smaller windows cases to happen. Again, input control determines the window size.

Creating Intelligent Melodies

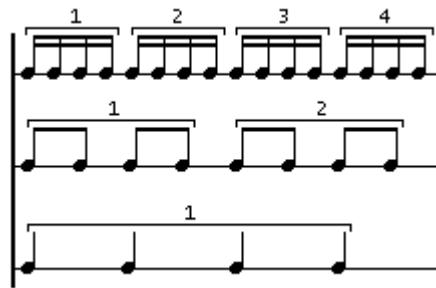
The calculated rhythms are then combined with note-choice. If beat 2 and 3 are to be played then the note for beat 2 is calculated immediately before beat 2, and the next note for beat 3 is calculated after beat 2 and before beat 3 is played. To summarize the timing of note and rhythm calculations:

- Notes are calculated for the next single beat
- Rhythms are calculated all at once for the next 4 beats.

If a beat does not happen, then note calculation waits for the next beat.

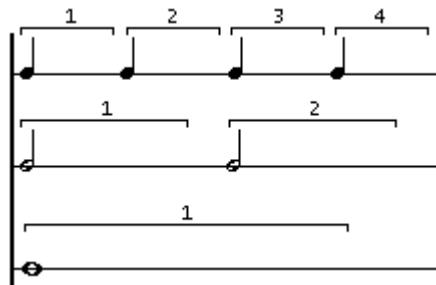
This combination of notes and rhythms can then be expanded over 3 voices to span 3 different rhythm tiers as shown below.

Figure 9 – Rhythmic Tiers with 4 note groupings – rhythmically busiest state



Voice 1 deals with groups of 4 sixteenth notes, voice 2 deals with groups of 4 eighth notes, and voice 3 deals with groups of 4 quarter notes. Thus, in one 4/4 measure, voice 1 executes 4 rhythm calculations, voice 2 executes 2 rhythm calculations, and voice 3 executes 1 rhythm calculation. Above is depiction of the busiest possible rhythmic situation. The most static situation shown below:

Figure 10 – Rhythmic Tiers with 4 note groupings – rhythmically most static state



Thus, the input controls to the rhythm are able to produce all rhythmic situations in between the two extremes. In terms of notes, all of the voices use the same note set, except an octave off-set from each other to cover a wider tonal range. The higher octave and fastest rhythm is assigned to voice 1. The middle octave, and medium paced rhythm are assigned to voice 2, and the lower octave, and slowest rhythm are assigned to voice 3. These assignments make sense because in an ensemble it is usually the instruments with the higher tonal range that usually play the faster notes. In an orchestra, one would seldom hear the tuba playing sixteenth notes while the piccolos are playing quarter notes! The specific instruments that are assigned to the voices are:

Voice 1 – Clarinet

Voice 2 – Marimba

Voice 3 – Acoustic Bass

These choices were based upon the tonal ranges in which these instruments typically play.

The 6 hamsters are then divided into 3 pairs and are each assigned to a voice (See Appendix C). Numbering from top to bottom, hamsters 1 and 2 are responsible for voice 1, hamsters 3 and 4 are responsible for voice 2 and hamsters 5 and 6 are responsible for voice 3. Within the pair, one hamster is the rhythm hamster, and the other is the tonal hamster. In terms of hamster numbers, 1, 3, and 5 are the tonal hamsters and 2, 4, and 6 are the rhythm hamsters. Referring to the pictures of the hamster controller in Appendix C, the yellow rows are the note hamsters and the blue rows are the rhythm hamsters.

Finally, with all of this control over different tonal and rhythmic windows, spanning 3 octaves, playing over 3 rhythmic tiers, and control by 6 hamsters, relatively complex musical situations arise over the simple elaboration of a few simple concepts.

Hardware Implementation

An Atmel Mega32 microcontroller, running at 4 MHz, was used as the heart of the sequencer. For the input, each of the sensors used two I/O port pins. All of PORTC as well as the PORTA0..3 bits were used to interface to the 6 distance sensors. I was able to create a MIDI OUT port and to transmit MIDI data via the Mega32's USART. The schematic for the MIDI out circuit and the detailed Mega32 schematic is shown in Appendix B.

For the sequencer input control device, the hamster sensing unit was built using the six sensors, wood, plexi-glass, hot glue, and electrical tape. During the construction, I tried to leave as few nooks and cracks as possible to discourage the hamsters from gnawing on the materials. However, the initial mounting of the distance sensors, as well as the accompanying wires, were very prone to being chewed on. To remedy this, fitted wooden blocks were placed around the mountings and protruded beyond the front face of the sensors. Because the hamsters did chew on these, covering the wood with electrical tape seemed to hinder this activity. Appendix C shows illustrations of different parts of the hamster sensing unit.

Six Sharp GP2D02 distance sensors were used within the sensing units. The protocol for the control bit and output bit are shown in Appendix C. One problem that arose during the debugging process was a seemingly random reset in the microcontroller. It was discovered that operating all six sensors in parallel was causing a loading problem. Turning on all six simultaneously drew a large amount of current (~210 mA) causing sharp, periodic drops of over 1V on the supply rails of the Mega32. However, this was remedied by placing a 10uF bypass capacitor on Vcc rail in order to dampen this high-frequency drop on voltage.

The sequencer output follows the MIDI standard and can be connected to the MIDI IN port of any MIDI Synthesizer. For this project, the Boss Dr. Synth, DS-330 Synthesizer was used. Final audio output was delivered via 1/8" audio jack to speakers or headphones.

Software Implementation

The embedded program was responsible for computing the musically intelligent output.

C Code

The Timer0 overflow interrupt was used to create the appropriate clocking to communicate with the sensors as well to set the tempo (approx. 156 quarter notes per minute) for the musical output. The Mega32 flash memory was used to store the Markov transition matrices for note-choice and rhythm. Rather than having the input adjust the terms of a single matrix, the inputs instead chose among a scale of gradually differing matrices.

The functions used are as follows:

char pitch(char i, char note_window) {}

-Returns the next note based on the present note and appropriate note window

-Called by state machine

char beat(char *beatgo_1, char *beatgo_2, char *beatgo_3,

r *beatgo_4){}

-Returns a flag that signals the completion of the beat calculation. It then updates the appropriate variables with new beat calculation

-Called in main() after the state machine sends a flag requesting a new beat calculation

void dist(char *dist1, char *dist2, char *dist3, char *dist4, char *dist5,

char *dist6); {}

-Continually gets data from the 6 distance sensors

-Called by timer0 overflow interrupt every 128 usec

void playnotes(void) {}

-statemachine that is responsible for playing the notes based on the the beat(), pitch(), and dist() function

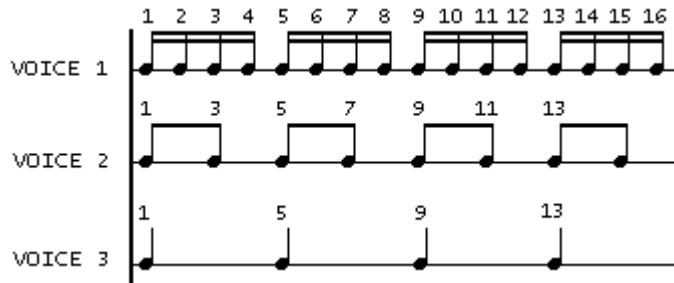
-calibrates exponential response data from distance sensors (see Appendix C) to be linear

-Called every 20 msec, and then counts to 5 every time before playing the next note

The overall operation of the code is as follows:

To play each measure the state machine, *playnotes()* is used. There are 16 states; one for each sixteenth note in the measure.

Figure 11 – States numbers assigned to notes



Each state checks the beat calculations for each voice and decides whether or not a note is to be played. The *beat()* function performs beat calculations for each voice every 4 beats within that voice. Voice 1 calls *beat()* before beats 1, 5, 9 and 13. Voice 2 calls *beat()* before beats 1 and 9. And Voice 3 calls *beat()* only before beat 1. If a beat is indeed scheduled to be played, the state machine calls the *pitch()* function to find the next note, based on the previously played note. Thus, *pitch* is called on the fly as needed. While this is occurring, global variables for the distance data for the 6 sensors are continually being updated. Based on these, the *beat()* and *pitch()* are direct as to which Markov transition matrices in flash are to be used. At the end of the measure, that state machine simply loops back to begin a new measure.

Debugging

Debugging was performed in two ways. The first was via serial communication to the PC's Hyper Terminal program. The USART baud rate need to be adjusted based on whether it was to run at 31.25 kBaud for MIDI data transmission, or at 9600 baud for communication with the PC. This was useful in following the path of the program and probing variables. The second form of debugging and testing was simply using my ears and listening to the music that the sequencer produced. This was an extremely effective method in developing the musical design.

RESULTS

Overall, I was extremely pleased with the results. Upon completion, all of my design requirements were realized, and the project turned out musically better than I expected.

Firstly, I learned a great deal about MIDI, both through the research leading up the project and through its actual design and implementation. I have successfully designed a MIDI sequencer that can send MIDI messages to any standard MIDI synthesizer. Was it novel and unique? Sitting there in a daze of glory in front of the hamster controller filled with my scurrying pets, listening to this odd, yet musically pleasing pseudo-improvised output, I felt quite satisfied that my whacky vision for hamster-controlled intelligent MIDI device had come to fruition. In developing the musical design, I spent a lot of time just listening to the output for significant periods of time to make sure that the matrices behaved as I had designed them to. In the end, I was very satisfied with the musical depth that came about from the simple manipulation of rhythm and note windows. The melodies that were produced by the hamster pairs to combine notes and rhythm were very logical, but still exciting due to clear elements of unpredictability.

The perceived response in the music as a function of the hamsters' movements was quite good. It was evident as one of the hamsters made its way down the path that larger note intervals became more frequent, or more consonant rhythms were being played. A surprising, yet simple, musical effect that was produce occurred when the note hamster activated a window of size zero, while the rhythm hamster activity a very busy consonant rhythm. This resulted in the same note being playing over and over in a fast rhythm which created a significant amount of melodic tension that released when the note hamster finally moved to again allow note movement around the note set.

The hamsters behaved very well. They walked back and forth at a sufficient pace so that the listener could pick out what the hamster was doing to the music. They began to chew in the wood fittings surrounding the sensor, but purpose of the wood was just that. The only problem that arose was when I used the hamster controller device in the afternoon for long periods of time. As nocturnal creatures, they sometimes fell asleep, creating a constant input to their distance sensor.

It was also fortunate that typical musical tempo is significantly slower than computation time within the microcontroller. This allowed for a good deal of operations and code to be executed without noticeable latency in the music.

The results that I achieved met and surpassed my expectations. Examples of the results can be found at the project website in the form of a music file and digital video.

<http://www.nbb.cornell.edu/neurobio/land/STUDENTPROJ/2002to2003/lil2/>

CONCLUSION

I was successfully able to create a MIDI device by combining both musical concepts and electrical engineering techniques. I was able to use my knowledge as a musician to formulate a design idea, and was then able to implement it by means of my engineering education. The musical design yielded many interesting theories in melodic intelligence. Although Markov note sequencing has been done before, its application to rhythm through this project has proven to be quite innovative. Ultimately, I was very pleased with the music that I was able to produce with my sequencer.

There are a few things that I could go back to and improve on. Many of the musical concepts for notes and rhythm have the potential for further modeling. Though this would in turn result in more complex software implementation, it could yield even more realistic musical results.

This was an excellent experience in the design process. I embraced the freedom that I was given in defining a MIDI project and I feel that I choose project that fulfilled all of my goals. This project was indeed an inspiration in showing me that I can fully combine my two sides as a musician and electrical engineer into one.

ACKNOWLEDGEMENTS

There are several people to whom I owe thanks regarding this project. I would like to thank Bruce Land and Ron Hoy for their guidance and creative inspiration. For his support and rhythmic counsel, I thank James Armstrong. I would also like to thank my parents for their understanding and support for following my dreams. Finally, I would like to thank the hamsters: Lollipapa, Lollimama, Minimama, Bighead, Minoru, Yoshimi, and Gary for being so cooperative throughout the process.

REFERENCES

1. “Digital Music Programming: Markov Chains”
[<http://peabody.sapp.org/class/dmp2/lab/markov1>](http://peabody.sapp.org/class/dmp2/lab/markov1)
2. “Grove Music Online” <<http://www.grovemusic.com>>
3. *Kientzle, Tim.* “A Programmer’s Guide to Sound”, Boston, MA: Addison-Wesley, 1998
4. *Messick, Paul.* “Maximum MIDI”. Greenwich, CT: Manning, 1998
5. “Merriam-webster Dictionary”
6. MIDI Manufacturer’s Association <<http://www.midi.org>>
7. “The MIDI Specification” <<http://www.borg.com/~jglatt/tech/midispec.htm>>

APPENDIX A –Glossary of Musical Terms Used

4/4 time signature

- musical time base with 4 beats per measure, 1 beat equals one quarter note

Chromatic scale

- a musical scale consisting entirely of half steps

<http://www.m-w.com>

Consonance

- Acoustically, the sympathetic vibration of sound waves of different frequencies related as the ratios of small whole numbers; psychologically, a harmonious sounding together of two or more notes, that is with an ‘absence of roughness’, ‘relief of tonal tension’ or the like. <http://www.grovemusic.com>

- Harmony or agreement among components <http://www.m-w.com>

Diatonic scale

- of or relating to a major or minor musical scale comprising intervals of five whole steps and two half steps

<http://www.m-w.com>

Dissonance

- The antonym to consonance with corresponding criteria of ‘roughness’ or ‘tonal tension’ <http://www.grovemusic.com>

- a mingling of discordant sounds; *especially* : a clashing or unresolved musical interval or chord <http://www.m-w.com>

Double Time

- The apparent doubling of the tempo

<http://www.grovemusic.com>

Half step

- a musical interval (as E-F or B-C) equivalent to one twelfth of an octave -- called also *semitone* <http://www.m-w.com>

Measure

- musical time

- a grouping of a specified number of musical beats located between two consecutive vertical lines on a staff

<http://www.m-w.com>

Melody

- Defined as pitched sounds arranged in musical time in accordance

<http://www.grovemusic.com>

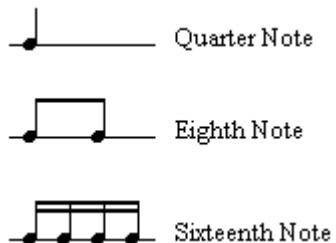
- a sweet or agreeable succession or arrangement of sounds

- a rhythmic succession of single tones organized as an aesthetic whole

<http://www.m-w.com>

Note Values

- Basic rhythmic subdivisions. 1 quarter note = 2 eighth notes = 4 sixteenth notes



Octave

- a musical interval embracing eight diatonic degrees

- the whole series of notes, tones, or digits comprised within this interval and forming the unit of the modern scale

<http://www.m-w.com>

Pentatonic scale

- consisting of five tones; *specifically* : being or relating to a scale in which the tones are arranged like a major scale with the fourth and seventh tones omitted

<http://www.m-w.com>

Polyphony

- a style of musical composition in which two or more independent melodies are juxtaposed in harmony

<http://www.m-w.com>

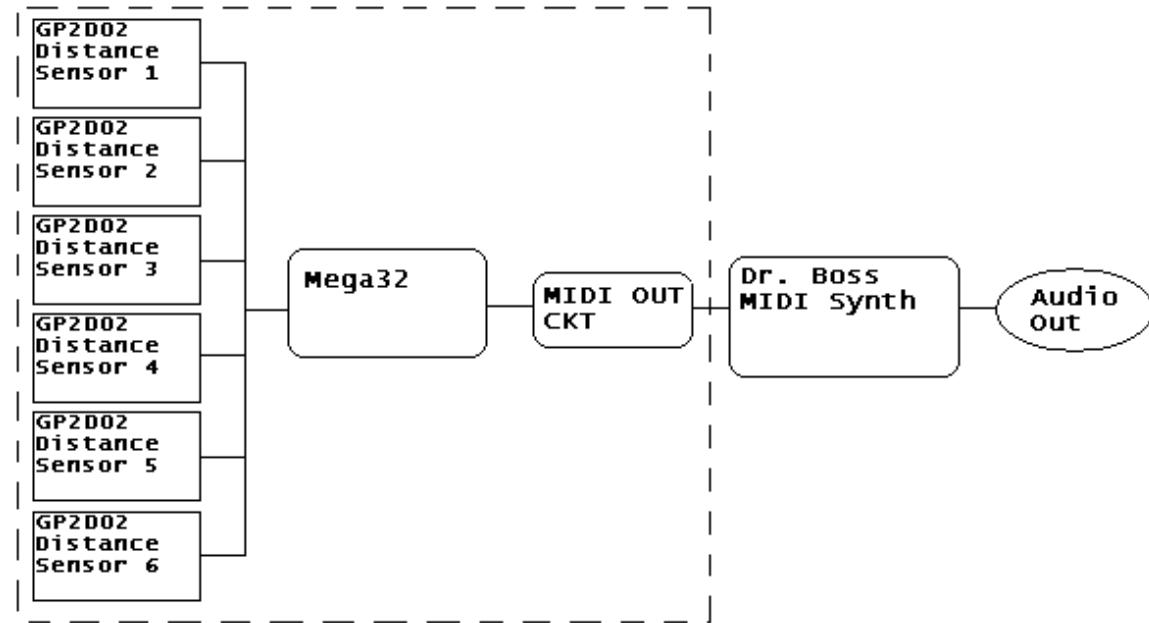
Whole step

- a musical interval (as C-D or G-A) comprising two half steps -- called also *whole tone*

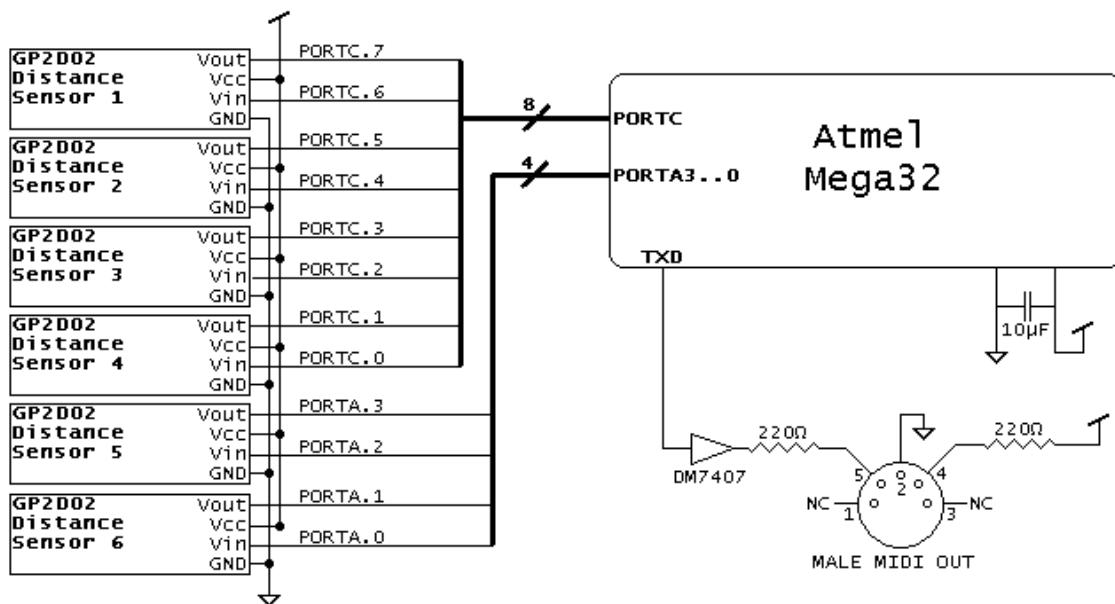
<http://www.m-w.com>

APPENDIX B –Hardware Schematics and Diagrams

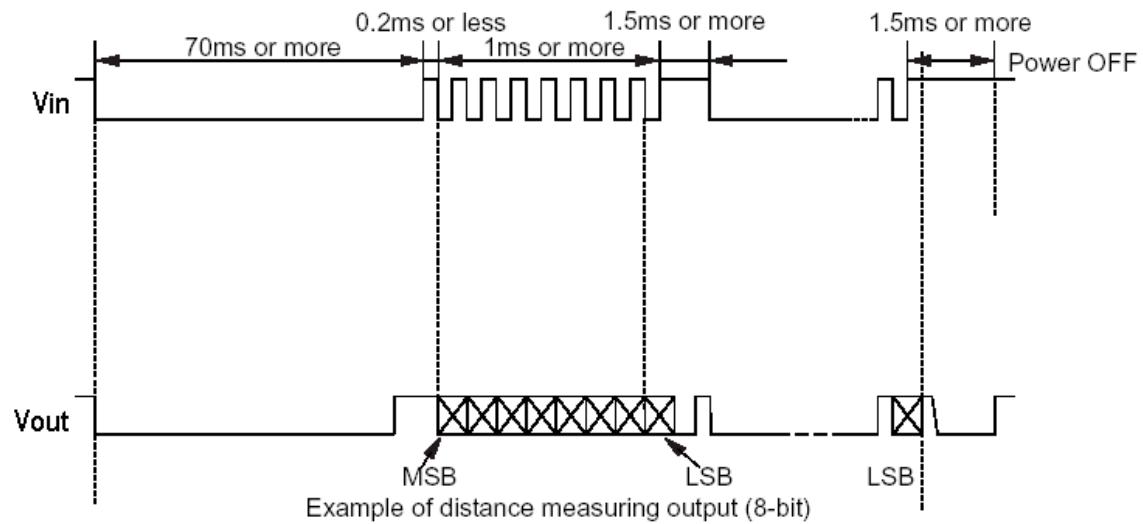
Hardware Schematic - Block Diagram



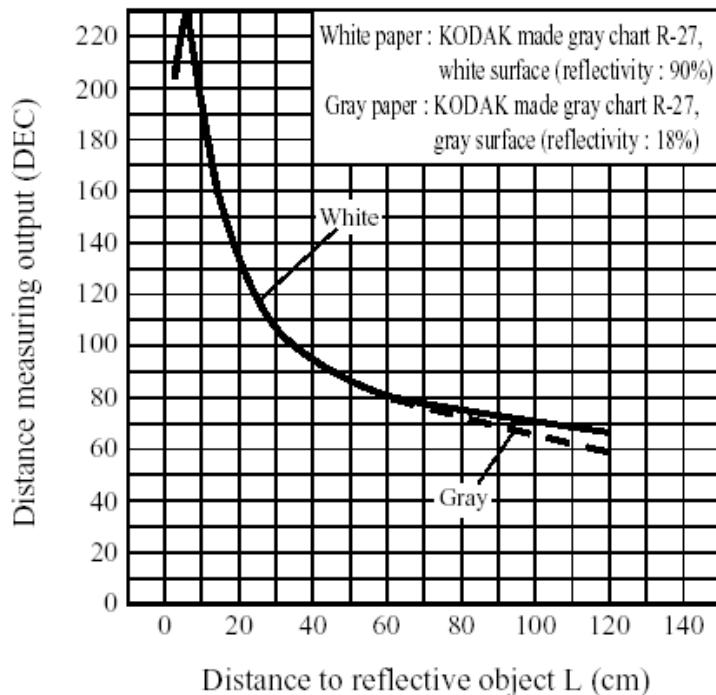
Hardware Schematic - Detailed



Sharp GP2D02 Distance Sensor – Timing Diagram



Sharp GP2D02 – Distance Measuring Output vs. Distance to Reflective Object



APPENDIX C – Hamster MIDI Controller Photographs

Complete System with Hamsters

(see link to file on webpage)

Hamster Sensing Unit

(see link to file on webpage)

Sequencer with MIDI OUT

(see link to file on webpage)

Controller Side Panel- Outside

(see link to file on webpage)

Controller Side Panel- Inside

(see link to file on webpage)

Mounted GP2D02 Sensors

(see link to file on webpage)

Controller Prototype Board

(see link to file on webpage)

APPENDIX D – Markov Transition Matrices

NOTE-CHOICE

```
{0xF, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},  
{0x0, 0xF, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, //Jump +/- UNISON  
{0x0, 0x0, 0xF, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0} //window = 0  
{0x0, 0x0, 0x0, 0xF, 0x0, 0x0, 0x0, 0x0, 0x0},  
{0x0, 0x0, 0x0, 0xF, 0x0, 0x0, 0x0, 0x0, 0x0},  
{0x0, 0x0, 0x0, 0x0, 0xF, 0x0, 0x0, 0x0, 0x0},  
{0x0, 0x0, 0x0, 0x0, 0x0, 0xF, 0x0, 0x0, 0x0},  
{0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0xF, 0x0, 0x0},  
{0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0xF, 0x0},  
{0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0xF},  
  
{0x7, 0x8, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},  
{0x5, 0x5, 0x5, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, //MAX JUMP. +/- 2nd  
{0x0, 0x5, 0x5, 0x5, 0x0, 0x0, 0x0, 0x0, 0x0}, //window = 1  
{0x0, 0x0, 0x5, 0x5, 0x5, 0x0, 0x0, 0x0, 0x0},  
{0x0, 0x0, 0x0, 0x5, 0x5, 0x5, 0x0, 0x0, 0x0},  
{0x0, 0x0, 0x0, 0x5, 0x5, 0x5, 0x0, 0x0, 0x0},  
{0x0, 0x0, 0x0, 0x0, 0x5, 0x5, 0x5, 0x0, 0x0},  
{0x0, 0x0, 0x0, 0x0, 0x0, 0x5, 0x5, 0x5, 0x0},  
{0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x5, 0x5, 0x5},  
{0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x8, 0x7},  
  
{0x5, 0x5, 0x5, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},  
{0x4, 0x3, 0x4, 0x4, 0x0, 0x0, 0x0, 0x0, 0x0}, //MAX JUMP. +/- 3rd  
{0x3, 0x3, 0x3, 0x3, 0x3, 0x0, 0x0, 0x0, 0x0, 0x0}, //window = 2  
{0x0, 0x3, 0x3, 0x3, 0x3, 0x3, 0x0, 0x0, 0x0, 0x0},  
{0x0, 0x0, 0x3, 0x3, 0x3, 0x3, 0x3, 0x0, 0x0, 0x0},  
{0x0, 0x0, 0x0, 0x3, 0x3, 0x3, 0x3, 0x3, 0x0},  
{0x0, 0x0, 0x0, 0x3, 0x3, 0x3, 0x3, 0x3, 0x0},  
{0x0, 0x0, 0x0, 0x0, 0x3, 0x3, 0x3, 0x3, 0x3},  
{0x0, 0x0, 0x0, 0x0, 0x0, 0x4, 0x4, 0x3, 0x4},  
{0x0, 0x0, 0x0, 0x0, 0x0, 0x5, 0x5, 0x5},  
  
{0x3, 0x4, 0x4, 0x4, 0x0, 0x0, 0x0, 0x0, 0x0},  
{0x3, 0x3, 0x3, 0x3, 0x3, 0x0, 0x0, 0x0, 0x0, 0x0}, //MAX JUMP. +/- 4TH  
{0x3, 0x2, 0x2, 0x2, 0x3, 0x3, 0x0, 0x0, 0x0, 0x0}, //window = 3  
{0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x3, 0x0, 0x0, 0x0},  
{0x0, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x3, 0x0, 0x0},  
{0x0, 0x0, 0x3, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x0},  
{0x0, 0x0, 0x0, 0x3, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2},  
{0x0, 0x0, 0x0, 0x0, 0x3, 0x3, 0x2, 0x2, 0x2, 0x3},  
{0x0, 0x0, 0x0, 0x0, 0x3, 0x3, 0x3, 0x3, 0x3},
```

```

{0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x4, 0x4, 0x4, 0x3},

{0x3, 0x3, 0x3, 0x3, 0x3, 0x0, 0x0, 0x0, 0x0, 0x0},,
{0x2, 0x2, 0x2, 0x3, 0x3, 0x0, 0x0, 0x0, 0x0},//MAX JUMP. +/- 5th
{0x2, 0x2, 0x2, 0x2, 0x2, 0x3, 0x0, 0x0, 0x0},//window = 4
{0x2, 0x2, 0x2, 0x1, 0x2, 0x2, 0x2, 0x2, 0x2, 0x0},,
{0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x0},,
{0x0, 0x2, 0x2, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2},,
{0x0, 0x0, 0x2, 0x2, 0x2, 0x1, 0x1, 0x2, 0x2, 0x2},,
{0x0, 0x0, 0x3, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2},,
{0x0, 0x0, 0x0, 0x3, 0x3, 0x3, 0x2, 0x2, 0x2},,
{0x0, 0x0, 0x0, 0x0, 0x3, 0x3, 0x3, 0x3, 0x3},,

{0x2, 0x3, 0x2, 0x3, 0x2, 0x3, 0x0, 0x0, 0x0, 0x0},,
{0x2, 0x2, 0x2, 0x2, 0x2, 0x3, 0x0, 0x0, 0x0},//MAX JUMP. +/-6th
{0x2, 0x2, 0x1, 0x2, 0x2, 0x2, 0x2, 0x2, 0x0},//window = 5
{0x2, 0x2, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x0},,
{0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2},,
{0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2},,
{0x0, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x2},,
{0x0, 0x0, 0x2, 0x2, 0x2, 0x2, 0x1, 0x2, 0x2},,
{0x0, 0x0, 0x0, 0x3, 0x2, 0x2, 0x2, 0x2, 0x2},,
{0x0, 0x0, 0x0, 0x3, 0x3, 0x2, 0x2, 0x2},,

{0x2, 0x2, 0x2, 0x2, 0x2, 0x3, 0x0, 0x0, 0x0},,
{0x2, 0x1, 0x2, 0x2, 0x2, 0x2, 0x2, 0x0, 0x0},//MAX JUMP. +/-7th
{0x2, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2, 0x0},//window = 6
{0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2},,
{0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2},,
{0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2},,
{0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1},,
{0x0, 0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x2},,
{0x0, 0x0, 0x2, 0x2, 0x2, 0x2, 0x2, 0x1, 0x2},,
{0x0, 0x0, 0x0, 0x3, 0x2, 0x2, 0x2, 0x2, 0x2},,

{0x1, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x0, 0x0},,
{0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2, 0x2, 0x0},//MAX JUMP. +/- Octave
{0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2},//window = 7
{0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2},,
{0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2},,
{0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2},,
{0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1},,
{0x2, 0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1},,
{0x0, 0x2, 0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1},,
{0x0, 0x0, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x1},,

```

```

{0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2, 0x0},  

{0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2},//MAX JUMP. +/- 9th  

{0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2},//window = 8  

{0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2},  

{0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2},  

{0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2},  

{0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1},  

{0x2, 0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1},  

{0x0, 0x2, 0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1},  

{0x0, 0x2, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2, 0x2},  

{0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2},//MAX JUMP. +/- 10th  

{0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2},//window = 9  

{0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2},  

{0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2},  

{0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2},  

{0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1},  

{0x2, 0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1},  

{0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1}

```

RHYTHM CALCULATION

```

{0xF, 0x0, 0x0, 0x0}, //X _ _ _ guaranteed only beat 1
{0x0, 0x0, 0x0, 0x0},
{0x0, 0x0, 0x0, 0x0},
{0x0, 0x0, 0x0, 0x0},  

*** LEAST RHYTMIC CONSONANCE ***
{0xE, 0x1, 0x0, 0x0}, //0 _ _ _
{0x0, 0xF, 0x0, 0x0},
{0x0, 0x0, 0x0, 0x0},
{0x0, 0x0, 0x0, 0x0},  

{0x7, 0x5, 0x1, 0x2}, //0 0 _ _
{0x0, 0xF, 0x0, 0x0},
{0x0, 0x0, 0xF, 0x0},
{0x0, 0x0, 0x0, 0xF},  

{0x7, 0x1, 0x1, 0x6}, //0 _ _ 0
{0x0, 0xD, 0x0, 0x2},
{0x0, 0x0, 0xF, 0x0},

```

```

{0x0, 0x0, 0x0, 0xF},

{0x7, 0x1, 0x6, 0x1}, //0 _ 0 _
{0x0, 0xD, 0x0, 0x2},
{0x0, 0x1, 0xE, 0x1},
{0x0, 0x2, 0x1, 0xC},

{0x7, 0x3, 0x2, 0x3}, //0 0 _ 0
{0x0, 0x1, 0x1, 0xD},
{0x0, 0x1, 0xD, 0x1},
{0x0, 0xD, 0x1, 0x1},

{0x6, 0x4, 0x3, 0x2}, //0 0 0 _
{0x0, 0x1, 0xD, 0x1},
{0x0, 0xD, 0x1, 0x1},
{0x0, 0x6, 0x6, 0x3},

{0x6, 0x2, 0x3, 0x4}, //0 _ 0 0
{0x0, 0x1, 0xD, 0x1},
{0x0, 0x1, 0x1, 0xD},
{0x0, 0x1, 0xD, 0x1},

{0x5, 0x3, 0x4, 0x3}, //0 0 0 0
{0x0, 0x1, 0xD, 0x1},
{0x0, 0x1, 0x1, 0xD},
{0x0, 0xD, 0x1, 0x1},

*** MOST RHYTMIC CONSONANCE ***
{0x0, 0xF, 0x0, 0x0}, //X X X X guaranteed all four beats
{0x0, 0x0, 0xF, 0x0},
{0x0, 0x0, 0x0, 0xF},
{0xF, 0x0, 0x0, 0x0}

```

APPENDIX E – hamsterMIDI.c

```
// HamsterMIDI for MEGA 32
//      - now integrate distance sensor to affect music
//      - now integrate variations in intelligence
//      - calibrate distance data
// -- Hamster roles:
//          (6)    3 pairs Melodic hamsters – 1 rhythm hamster, 1 note hamster per pair

#include <Mega32.h>
#include <stdio.h> // sprintf
#include <stdlib.h>
#define begin {
#define end   }
#define beat states
#define BEAT1 0
#define BEAT2 1
#define BEAT3 2
#define BEAT4 3
#define BEAT5 4
#define BEAT6 5
#define BEAT7 6
#define BEAT8 7
#define BEAT9 8
#define BEAT10 9
#define BEAT11 10
#define BEAT12 11
#define BEAT13 12
#define BEAT14 13
#define BEAT15 14
#define BEAT16 15
#define distance sensor states
#define RESET 0
#define PREP 1
#define GET 2
#define WAIT 3

//RRRRRRRRRRRRRRRR RHYTHM CHOICE RRRRRRRRRRRRRRRRRRRRRR//
// rhythm markov transition matrices
flash unsigned char beat_prob[40][4]=
{
    //                                _bwindow_
    {0xF, 0x0, 0x0, 0x0}, //X _ _ _           +0
    {0x0, 0x0, 0x0, 0x0},
    {0x0, 0x0, 0x0, 0x0},
    {0x0, 0x0, 0x0, 0x0},
    {0xE, 0x1, 0x0, 0x0}, //0 _ _ _           +4
```



```

{0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0xF, 0x0, 0x0},
{0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0xF, 0x0},
{0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0xF},

/***/ {0x7, 0x8, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},//      +10
{0x5, 0x5, 0x5, 0x0, 0x0, 0x0, 0x0, 0x0},//MAX JUMP. +/- 2nd
{0x0, 0x5, 0x5, 0x5, 0x0, 0x0, 0x0, 0x0, 0x0},
{0x0, 0x0, 0x5, 0x5, 0x5, 0x0, 0x0, 0x0, 0x0, 0x0},
{0x0, 0x0, 0x5, 0x5, 0x5, 0x5, 0x0, 0x0, 0x0, 0x0},
{0x0, 0x0, 0x0, 0x5, 0x5, 0x5, 0x5, 0x0, 0x0, 0x0, 0x0},
{0x0, 0x0, 0x0, 0x0, 0x5, 0x5, 0x5, 0x5, 0x0, 0x0, 0x0, 0x0},
{0x0, 0x0, 0x0, 0x0, 0x0, 0x5, 0x5, 0x5, 0x5, 0x0, 0x0, 0x0, 0x0},
{0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x5, 0x5, 0x5, 0x5, 0x0, 0x0, 0x0, 0x0},
{0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x5, 0x5, 0x5, 0x5, 0x0, 0x0, 0x0, 0x0},
{0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x5, 0x5, 0x5, 0x5},,
{0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x8, 0x7},,

/***/ {0x5, 0x5, 0x5, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},//      +20
{0x4, 0x3, 0x4, 0x4, 0x0, 0x0, 0x0, 0x0, 0x0},//MAX JUMP. +/- 3rd
{0x3, 0x3, 0x3, 0x3, 0x3, 0x0, 0x0, 0x0, 0x0, 0x0},
{0x0, 0x3, 0x3, 0x3, 0x3, 0x3, 0x0, 0x0, 0x0, 0x0},
{0x0, 0x0, 0x3, 0x3, 0x3, 0x3, 0x3, 0x0, 0x0, 0x0, 0x0},
{0x0, 0x0, 0x0, 0x3, 0x3, 0x3, 0x3, 0x3, 0x0, 0x0, 0x0},
{0x0, 0x0, 0x0, 0x0, 0x3, 0x3, 0x3, 0x3, 0x3, 0x0, 0x0, 0x0},
{0x0, 0x0, 0x0, 0x0, 0x0, 0x3, 0x3, 0x3, 0x3, 0x3, 0x0, 0x0, 0x0},
{0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x3, 0x3, 0x3, 0x3, 0x3, 0x0, 0x0, 0x0},
{0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x4, 0x4, 0x3, 0x4},,
{0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x5, 0x5, 0x5},

/***/ {0x3, 0x4, 0x4, 0x4, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},//      +30
{0x3, 0x3, 0x3, 0x3, 0x3, 0x0, 0x0, 0x0, 0x0, 0x0},//MAX JUMP. +/- 4TH
{0x3, 0x2, 0x2, 0x2, 0x3, 0x3, 0x0, 0x0, 0x0, 0x0},
{0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x3, 0x0, 0x0, 0x0},
{0x0, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x3, 0x0, 0x0},
{0x0, 0x0, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x0},,
{0x0, 0x0, 0x0, 0x3, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2},
{0x0, 0x0, 0x0, 0x0, 0x3, 0x3, 0x2, 0x2, 0x2, 0x3},,
{0x0, 0x0, 0x0, 0x0, 0x0, 0x3, 0x3, 0x3, 0x3, 0x3},,
{0x0, 0x0, 0x0, 0x0, 0x0, 0x4, 0x4, 0x4, 0x3},,

/***/ {0x3, 0x3, 0x3, 0x3, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},//      +40
{0x2, 0x2, 0x2, 0x3, 0x3, 0x0, 0x0, 0x0, 0x0, 0x0},//MAX JUMP. +/- 5th
{0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x3, 0x0, 0x0, 0x0},
{0x2, 0x2, 0x2, 0x1, 0x2, 0x2, 0x2, 0x2, 0x0, 0x0},,
{0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x0},,
{0x0, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2},,
{0x0, 0x0, 0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x2, 0x2, 0x2},,
{0x0, 0x0, 0x0, 0x3, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2},,
{0x0, 0x0, 0x0, 0x0, 0x3, 0x3, 0x2, 0x2, 0x2, 0x2},,
{0x0, 0x0, 0x0, 0x0, 0x0, 0x3, 0x3, 0x3, 0x3, 0x3},,
// ENCOURAGE JUMPS!!!
{0x0, 0x0, 0x0, 0x0, 0x3, 0x3, 0x3, 0x3, 0x3},,

```

```

/***/ {0x2, 0x3, 0x2, 0x3, 0x2, 0x3, 0x0, 0x0, 0x0, 0x0},//      +50
{0x2, 0x2, 0x2, 0x2, 0x2, 0x3, 0x0, 0x0, 0x0},//MAX JUMP. +/-6th
{0x2, 0x2, 0x1, 0x2, 0x2, 0x2, 0x2, 0x0, 0x0},
{0x2, 0x2, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2, 0x0},
{0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2},
{0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2},
{0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2},
{0x0, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2},
{0x0, 0x0, 0x2, 0x2, 0x2, 0x2, 0x1, 0x2, 0x2, 0x2},
{0x0, 0x0, 0x3, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2},
{0x0, 0x0, 0x0, 0x3, 0x3, 0x3, 0x2, 0x2, 0x2, 0x2},

/***/ {0x2, 0x2, 0x2, 0x2, 0x2, 0x3, 0x0, 0x0, 0x0},//      +60
{0x2, 0x1, 0x2, 0x2, 0x2, 0x2, 0x2, 0x0, 0x0},//MAX JUMP. +/-7th
{0x2, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2, 0x0},
{0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2},
{0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2},
{0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2},
{0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2},
{0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2},
{0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2},
{0x0, 0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x2},
{0x0, 0x0, 0x2, 0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x2},
{0x0, 0x0, 0x3, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2},

/***/ {0x1, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x0, 0x0},//      +70
{0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2, 0x2, 0x0},//MAX JUMP. +/- Octave
{0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2, 0x2},
{0x2, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2},
{0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2},
{0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2},
{0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2},
{0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2},
{0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2},
{0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2},
{0x0, 0x2, 0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2},
{0x0, 0x0, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x2},

/***/ {0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2, 0x0},//      +80
{0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2, 0x2},//MAX JUMP. +/- 9th
{0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2, 0x2},
{0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2},
{0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2},
{0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2},
{0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2},
{0x2, 0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2},
{0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2},
{0x0, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1, 0x2},

/***/ {0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2, 0x2},//      +90
{0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2, 0x2},//MAX JUMP. +/- 10th

```

```

{0x1, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2},  

{0x2, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2, 0x2},  

{0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2, 0x2},  

{0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x2, 0x2},  

{0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x2},  

{0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1},  

{0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1},  

{0x2, 0x2, 0x2, 0x2, 0x1, 0x1, 0x1, 0x1, 0x1}  

};  

//PPPPPPPPPPPPPPPPPPPPPPPPPPPP PITCHES PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP//  

// set of notes to choose from  

flash unsigned char notes[2][10]=  

{  

    // White Keys  

    // A3 B3 C4 D4 E4 F4 G4 A4 B4 C5  

    {0x39,0x3B,0x3C,0x3E,0x40,0x41,0x43,0x45,0x47,0x48},  

    // Cmaj/Amin Pentatonic G3 to C5      THIS ONE USED!!!  

    // G3 A3 C4 D4 E4 G4 A4 C5 D5 E5  

    {0x37,0x39,0x3C,0x3E,0x40,0x43,0x45,0x48,0x4A,0x4C}  

};  


```

```

//-----  

//MIDI MESSAGE PRESETS  

//NOTE ON/OFF messages  

unsigned char noteon0 = 0x90; //note on channel 0  

unsigned char noteoff0 = 0x80; //note off channel 0  

unsigned char noteon1 = 0x91; //note on channel 1  

unsigned char noteoff1 = 0x81; //note off channel 1  

unsigned char noteon2 = 0x92; //note on channel 2  

unsigned char noteoff2 = 0x82; //note off channel 2  

unsigned char noteon_perc = 0x99; //Percussion ON  

unsigned char noteoff_perc = 0x89; //Percussion OFF  

//misc. messages  

unsigned char progch0 = 0xC0;           //Program Change Channel 0  

unsigned char progch1 = 0xC1;           //Program Change Channel 1  

unsigned char progch2 = 0xC2;           //Program Change Channel 2  

unsigned char def_vol0 = 0x64;          //default volume 100  

unsigned char def_vol1 = 0x64;          //default volume 100  

unsigned char def_vol2 = 0x64;          //default volume 100  

unsigned char C4 = 0x3c;                //middle C  

//instrument patches  

unsigned char acousticBD = 0x23;  

unsigned char BD = 0x24;  

unsigned char rim = 0x25;  

unsigned char acousticSD = 0x26;
```

```

unsigned char clap = 0x27;
unsigned char hibongo = 0x3C;
unsigned char lobongo = 0x3d;
unsigned char vibraphone = 0x0C;
unsigned char clarinet = 0x47;
unsigned char square = 0x50;
unsigned char saw = 0x51;
unsigned char piano = 0x00;
unsigned char banjo = 0x9A;
unsigned char acousticBass = 0x20;

//timer0 declarations
unsigned char reload;
unsigned char time0;
//state machine
unsigned char state;
unsigned char state_go;
unsigned char count; //counter for tempo
unsigned char i0, i1, i2; //indices for voice beat plays

unsigned char nwindow0, nwindow1, nwindow2; //NOTE Prob MATRIX - chose note window
unsigned char bwindow0, bwindow1, bwindow2; //BEAT Prob MATRIX - chose rhythm
unsigned char scale, offset0, offset1, offset2; //NOTES MATRIX - chose which scale to use

//beat declarations
unsigned char beat_found, beat_found0, beat_found1, beat_found2;
unsigned char beatgo0[4] = {0, 0, 0, 0};
unsigned char beatgo1[4] = {0, 0, 0, 0};
unsigned char beatgo2[4] = {0, 0, 0, 0};
unsigned char beatgo0_1, beatgo0_2, beatgo0_3, beatgo0_4;
unsigned char beatgo1_1, beatgo1_2, beatgo1_3, beatgo1_4;
unsigned char beatgo2_1, beatgo2_2, beatgo2_3, beatgo2_4;
unsigned char Bi, Bj; //beat counters
unsigned char cum_prob_lo, cum_prob_hi;
unsigned int random;
unsigned char rnd;
unsigned char iter;
unsigned char BEAT4_played;
unsigned char BEAT8_played;
unsigned char BEAT12_played;
unsigned char BEAT16_played;
unsigned char bigBEAT4_played;
unsigned char bigBEAT8_played;
unsigned char hugeBEAT4_played;
//distance sensor declarations
//unsigned char reload; //-----
unsigned char d_state;

```

```

unsigned char dist_go;
unsigned int sensor_time;
unsigned char distance1, distance2, distance3, distance4, distance5, distance6;
unsigned char d1, d2, d3, d4, d5, d6;
unsigned char bits_taken;
unsigned char wait_time, in_dist;
bit bbb = 1;
//subroutines
void initialize (void);
void playnotes (void);
char beat(char *beatgo_1, char *beatgo_2, char *beatgo_3, char *beatgo_4, char beat_window); //take in
beatgo array, return beatfound
char pitch(char i, char note_window);
int rand(void);
//void dist(void);

//-----TIMER0 Overflow ISR-----
interrupt [TIM0_OVF] void timer0_overflow(void)
{
    TCNT0 = reload;
    if(time0 > 0) --time0;          //&&&& Tempo - TIME BASE &&&&&&
    dist_go = 1;
}
//-----MAIN MAIN MAIN-----
void main(void)
{
    initialize();

    putchar(progch0);    putchar(clarinet);    //set voice1 on channel 0
    putchar(progch1);    putchar(vibraphone); //set voice2 on channel 1
    putchar(progch2);    putchar(acousticBass); //set voice3 on channel 2
    while (1) {
        if(!time0) /*&& (beat_found0) /*&& (beat_found1) && (beat_found2) */ {
            playnotes();
            time0 = 150;           //&&&& Tempo - TIME BASE... ~156 bpm &&&&
        }
        if(!beat_found0) beat_found0 = beat(&beatgo0_1, &beatgo0_2, &beatgo0_3, &beatgo0_4,
bwindow0);
        if(!beat_found1) beat_found1 = beat(&beatgo1_1, &beatgo1_2, &beatgo1_3, &beatgo1_4,
bwindow1);
        if(!beat_found2) beat_found2 = beat(&beatgo2_1, &beatgo2_2, &beatgo2_3, &beatgo2_4,
bwindow2);
        if((dist_go) && (!in_dist))dist(&distance1, &distance2, &distance3, &distance4,
&distance5, &distance6);
    }
}

```

```

//-----New Note----- calculate next note
char pitch(char i, char note_window)
begin
    unsigned char j;
    random = rand();           //get 16-bit random number
    rnd = 0x00 | random;       //get bottom 8-bits
    rnd = (0xF & rnd);        //get bottom 4-bits...
    for(j=0;j<10;j++){
        if(j==0){
            cum_prob_hi = note_prob[i+note_window][0]; //upper bound of prob
        }
        cum_prob_lo = 0;           //lower bound of prob window
    }
    else{
        cum_prob_hi = cum_prob_hi + note_prob[i+note_window][j];
        cum_prob_lo = cum_prob_lo + note_prob[i+note_window][j-1];
    }
    //test to see if rnd # falls inside the
    //probability window assigned to the next note
    if(note_prob[i+note_window][j] != 0) {
        if(rnd <= cum_prob_hi && rnd >= cum_prob_lo){
            i = j; //set new current note and to move to new row
            j = 10; //stop looking through row
        }
    }
}
return i; //new i has been found!!!
end //----- End New Note
//-----New Beat-----
//calculate next beat
char beat (char *beatgo_1, char *beatgo_2, char *beatgo_3, char *beatgo_4, char beat_window)
begin
    Bi=0;
    *beatgo_1 = 0;      *beatgo_2 = 0; *beatgo_3 = 0; *beatgo_4 = 0;
    for(iter=0;iter<3;iter++){ //iterate calculations 3 times to give a chance for all beats to exist
        random = rand();           //get 16-bit random number
        rnd = 0x00 | random;       //get bottom 8-bits
        rnd = (0xF & rnd);        //get bottom 4-bits...
        for(Bj=0;Bj<4;Bj++){
            if(Bj==0){
                cum_prob_hi = beat_prob[Bi+beat_window][0]; //upper bound of prob
            }
            cum_prob_lo = 0;           //lower bound of prob window
        }
        else{
            cum_prob_hi = cum_prob_hi + beat_prob[Bi+beat_window][Bj];
            cum_prob_lo = cum_prob_lo + beat_prob[Bi+beat_window][Bj-1];
        }
    }
}

```

```

        }
        //test to see if rnd # falls inside the probability window assigned to the next note
        if(beat_prob[Bi+beat_window][Bj] != 0) {
            if(rnd <= cum_prob_hi && rnd >= cum_prob_lo){
                Bi = Bj;           //set new current note and to move to new row
                if(!iter) *beatgo_1 = 1;      //guarantee that beat 1 always happens
                switch(Bi){
                    case 0: *beatgo_1 = 1;          break;
                    case 1: *beatgo_2 = 1;          break;
                    case 2: *beatgo_3 = 1;          break;
                    case 3: *beatgo_4 = 1;          break;
                }
                Bj = 4; //STOP looking through row and go to 'next' row
            }
        }
    }
    beat_found = 1;
end //----- END New Beat
//-----Get Distances from 6 Sensors-----
void dist(char *dist1, char *dist2, char *dist3, char *dist4, char *dist5, char *dist6)
begin
unsigned char data1[8], data2[8], data3[8], data4[8], data5[8], data6[8];
dist_go=0;
in_dist = 1; //say " I'M IN DIST()!!! WAIT!!! "
switch(d_state) {
    case RESET:
        PORTC.0 = 0;          PORTC.2 = 0;          PORTC.4 = 0;          PORTC.6 = 0;
PORTA.0 = 0; PORTA.2 = 0;
        sensor_time =0;       d_state = PREP;
        in_dist = 0;
    break;
    case PREP:
        if(sensor_time++ == 500) {
            sensor_time = 0;       d_state = GET;
        }
        in_dist = 0;
    break;
    case GET:
        if(sensor_time++ <= 15) {
            if(sensor_time%2 ==0){
                PORTC.0 = 1;
                PORTC.2 = 1;
                PORTC.4 = 1;
                PORTC.6 = 1;
                PORTA.0 = 1;
                PORTA.2 = 1;

```

```

    }
else {
    PORTC.0 = 0; PORTC.2 = 0; PORTC.4 = 0; PORTC.6 = 0; PORTA.0 = 0;
PORTA.2 = 0;
}
//PORTC.0 = ~PORTC.0; PORTC.2 = ~PORTC.2; PORTC.4 = ~PORTC.4;
PORTC.6 = ~PORTC.6; PORTB.0 = ~PORTB.0; PORTB.2 = ~PORTB.2;
if((PORTA.0 == 1) & (bits_taken != 8)) {
    data1[bits_taken] = PINC.7; //assume that all PORTC.0,2,4,6, PORTA.0
    data2[bits_taken] = PINC.5; // are in acting parallel
    data3[bits_taken] = PINC.3;
    data4[bits_taken] = PINC.1;
    data5[bits_taken] = PINA.3;
    data6[bits_taken] = PINA.1;
    bits_taken++;
}
else { //adjust range distance data (~30" - 4") -> (~100 - 0)
    *dist1 = 250-
((data1[0]*128)+(data1[1]*64)+(data1[2]*32)+(data1[3]*16)+(data1[4]*8)+(data1[5]*4)+(data1[6]*2)+(data1[7]));
    *dist2 = 250-
((data2[0]*128)+(data2[1]*64)+(data2[2]*32)+(data2[3]*16)+(data2[4]*8)+(data2[5]*4)+(data2[6]*2)+(data2[7]));
    /*dist3 = 250-
((data3[0]*128)+(data3[1]*64)+(data3[2]*32)+(data3[3]*16)+(data3[4]*8)+(data3[5]*4)+(data3[6]*2)+(data3[7]));
    /*dist4 = 250-
((data4[0]*128)+(data4[1]*64)+(data4[2]*32)+(data4[3]*16)+(data4[4]*8)+(data4[5]*4)+(data4[6]*2)+(data4[7]));
    /*dist5 = 250-
((data5[0]*128)+(data5[1]*64)+(data5[2]*32)+(data5[3]*16)+(data5[4]*8)+(data5[5]*4)+(data5[6]*2)+(data5[7]));
    /*dist6 = 250-
((data6[0]*128)+(data6[1]*64)+(data6[2]*32)+(data6[3]*16)+(data6[4]*8)+(data6[5]*4)+(data6[6]*2)+(data6[7]));
    PORTC.0 = 1;          PORTC.2 = 1;          PORTC.4 = 1;          PORTC.6 = 1;
    PORTA.0 = 1; PORTA.2 = 1;
    bits_taken = 0; sensor_time = 0;      d_state = WAIT;
}
in_dist = 0;
break;
case WAIT:
    if(wait_time++ == 2) {
        PORTC.0 = 0;          PORTC.2 = 0;          PORTC.4 = 0;          PORTC.6 = 0;
    PORTA.0 = 0; PORTA.2 = 0;
    wait_time = 0;      d_state = RESET;
}

```

```

    }

    if(wait_time == 0) {
        *dist3 = 250-
((data3[0]*128)+(data3[1]*64)+(data3[2]*32)+(data3[3]*16)+(data3[4]*8)+(data3[5]*4)+(data3[6]*2)+(d
ata3[7]));
        *dist4 = 250-
((data4[0]*128)+(data4[1]*64)+(data4[2]*32)+(data4[3]*16)+(data4[4]*8)+(data4[5]*4)+(data4[6]*2)+(d
ata4[7]));
    }
    if(wait_time == 1) {
        *dist5 = 250-
((data5[0]*128)+(data5[1]*64)+(data5[2]*32)+(data5[3]*16)+(data5[4]*8)+(data5[5]*4)+(data5[6]*2)+(d
ata5[7]));
        *dist6 = 250-
((data6[0]*128)+(data6[1]*64)+(data6[2]*32)+(data6[3]*16)+(data6[4]*8)+(data6[5]*4)+(data6[6]*2)+(d
ata6[7]));
    }
    in_dist = 0;
    break;

}

end //----- END distance sensors
//-----STATE MACHINE-----
void playnotes(void)
begin
state_go=0; //reset state change
//    printf(" distance1 = %d", distance1);//ready for debuggin!!!
//    printf(" distance2 = %d", distance2);
//    printf(" distance3 = %d", distance3);
//    printf(" distance4 = %d", distance4);
//    printf(" distance5 = %d", distance5);
//    printf(" distance6 = %d", distance6);
//----- TEST TEST TEST TEST -----
//manipulate distances to see if the delay on the if chain is fucking things up!!!
//    distance1 = 87;      //87 - smallest delay, <35 largest delay
//    distance2 = 83;      //83 - smallest delay, <30 largest delay
//    distance3 = 84;      //84 - smallest delay, <30 largest delay
//    distance4 = 88;      //88 - smallest delay, <30 largest delay
//    distance5 = 78;      //77 - smallest delay, <35 largest delay
//    distance6 = 78;      //77 - smallest delay, <25 largest delay
//***** CALIBRATE DISTANCE DATA TO LINEAR *****//
//Distance Sensor 1
if      (distance1 >= 86)           d1 = 0;
else if ((distance1 >= 85) && (distance1 <86))   d1 = 10;
else if ((distance1 >= 84) && (distance1 <85))   d1 = 20;
else if ((distance1 >= 83) && (distance1 <84))   d1 = 30;

```

```

else if ((distance1 >= 82) && (distance1 <83))      d1 = 40;
else if ((distance1 >= 80) && (distance1 <82))      d1 = 50;
else if ((distance1 >= 74) && (distance1 <80))      d1 = 60;
else if ((distance1 >= 59) && (distance1 <74))      d1 = 70;
else if ((distance1 >= 35) && (distance1 <59))      d1 = 80;
else if ((distance1 >= 1) && (distance1 <35))       d1 = 90;
//Distance Sensor 2
if          (distance2 >= 82)                         d2 = 0;
else if ((distance2 >= 81) && (distance2 <82))      d2 = 4;
else if ((distance2 >= 80) && (distance2 <81))      d2 = 8;
else if ((distance2 >= 78) && (distance2 <80))      d2 = 12;
else if ((distance2 >= 76) && (distance2 <78))      d2 = 16;
else if ((distance2 >= 75) && (distance2 <76))      d2 = 20;
else if ((distance2 >= 70) && (distance2 <73))      d2 = 24;
else if ((distance2 >= 50) && (distance2 <70))      d2 = 28;
else if ((distance2 >= 30) && (distance2 <50))      d2 = 32;
else if ((distance2 >= 1) && (distance2 <30))       d2 = 36;
//Distance Sensor 3
if          (distance3 >= 83)                         d3 = 0;
else if ((distance3 >= 80) && (distance3 <83))      d3 = 10;
else if ((distance3 >= 68) && (distance3 <75))      d3 = 20;
else if ((distance3 >= 65) && (distance3 <68))      d3 = 30;
else if ((distance3 >= 63) && (distance3 <65))      d3 = 40;
else if ((distance3 >= 59) && (distance3 <63))      d3 = 50;
else if ((distance3 >= 54) && (distance3 <59))      d3 = 60;
else if ((distance3 >= 45) && (distance3 <54))      d3 = 70;
else if ((distance3 >= 30) && (distance3 <45))      d3 = 80;
else if ((distance3 >= 1) && (distance3 <30))       d3 = 90;
//Distance Sensor 4
if          (distance4 >= 87)                         d4 = 0;
else if ((distance4 >= 85) && (distance4 <87))      d4 = 4;
else if ((distance4 >= 79) && (distance4 <85))      d4 = 8;
else if ((distance4 >= 75) && (distance4 <79))      d4 = 12;
else if ((distance4 >= 73) && (distance4 <75))      d4 = 16;
else if ((distance4 >= 70) && (distance4 <73))      d4 = 20;
else if ((distance4 >= 65) && (distance4 <70))      d4 = 24;
else if ((distance4 >= 55) && (distance4 <65))      d4 = 28;
else if ((distance4 >= 30) && (distance4 <55))      d4 = 32;
else if ((distance4 >= 1) && (distance4 <30))       d4 = 36;
//Distance Sensor 5
if          (distance5 >= 77)                         d5 = 0;
else if ((distance5 >= 75) && (distance5 <77))      d5 = 10;
else if ((distance5 >= 70) && (distance5 <75))      d5 = 20;
else if ((distance5 >= 68) && (distance5 <70))      d5 = 30;
else if ((distance5 >= 66) && (distance5 <68))      d5 = 40;
else if ((distance5 >= 64) && (distance5 <66))      d5 = 50;
else if ((distance5 >= 62) && (distance5 <64))      d5 = 60;

```

```

else if ((distance5 >= 55) && (distance5 <62))      d5 = 70;
else if ((distance5 >= 35) && (distance5 <55))      d5 = 80;
else if ((distance5 >= 1) && (distance5 <35))       d5 = 90;

//Distance Sensor 6
if      (distance6 >= 77)                      d6 = 0;
else if ((distance6 >= 75) && (distance6 <77))    d6 = 4;
else if ((distance6 >= 73) && (distance6 <75))    d6 = 8;
else if ((distance6 >= 68) && (distance6 <73))    d6 = 12;
else if ((distance6 >= 61) && (distance6 <68))    d6 = 16;
else if ((distance6 >= 57) && (distance6 <61))    d6 = 20;
else if ((distance6 >= 50) && (distance6 <57))    d6 = 24;
else if ((distance6 >= 43) && (distance6 <50))    d6 = 28;
else if ((distance6 >= 25) && (distance6 <43))    d6 = 32;
else if ((distance6 >= 1) && (distance6 <25))     d6 = 36;

// printf(" d1 = %d ", d1);
// printf(" d2 = %d ", d2);
// printf(" d3 = %d ", d3);  c
// printf(" d4 = %d ", d4);
// printf(" d5 = %d ", d5);
// printf(" d6 = %d ", d6);

//***** INTELLIGENCE HAPPENS HERE!!! *****/
/*Scale Parameters*/ scale = 1;           // 0 = diatonic
                                         // 1 = pentatonic
/*Offset Parameters*/ offset0 = 12;        //Voice1 - up 1 octave
                                         //Voice2 - same octave
                                         //Voice3 - down 1 octave
offset1 = 0;
offset2 = -12;
/*Note Parameters*/ nwindow0 = d1;         // /MELODY Hamster 1 - notes
                                         // /MELODY Hamster 2 - notes
                                         // /BASS Hamster 2 - notes
nwindow1 = d3;
nwindow2 = d5;
/*Beat Parameters*/ bwindow0 = d2;         // /MELODY Hamster 1 - rhythm
                                         // /MELODY Hamster 2 - rhythm
                                         // /BASS Hamster 3 - rhythm
bwindow1 = d4;
bwindow2 = d6;
/*Velocity Parameters*/ def_vol0 = 90;      // /MELODY Hamster 1 - volume
                                         // /MELODY Hamster 2 - volume
                                         // /BASS HAMSTER - volume
def_vol1 = 100;
def_vol2 = 110;
//***** *****/
// printf(" nwindow0 = %d ", nwindow0);   //ready for debugging!!!
//printf(" nwindow1 = %d ", nwindow1);
//printf(" nwindow2 = %d ", nwindow2);
//printf(" bwindow0 = %d ", bwindow0);
//printf(" bwindow1 = %d ", bwindow1);
//printf(" bwindow2 = %d ", bwindow2);

```

```

//assign beat calculations to beatgox[] arrays
beatgo0[0] = beatgo0_1;    beatgo0[1] = beatgo0_2; beatgo0[2] = beatgo0_3; beatgo0[3] = beatgo0_4;
beatgo1[0] = beatgo1_1;    beatgo1[1] = beatgo1_2; beatgo1[2] = beatgo1_3; beatgo1[3] = beatgo1_4;
beatgo2[0] = beatgo2_1;    beatgo2[1] = beatgo2_2; beatgo2[2] = beatgo2_3; beatgo2[3] = beatgo2_4;

switch (state)           //based on beat calculations, see if a state/beat should play VoiceX note
begin
    case BEAT1://----- Begin New 4/4 Measure-----
        if(count++ == 5){
            //printf(" BEAT1 ");
putchar(noteon_perc); putchar(rim);  putchar(def_vol0);
            count=0;
            state = BEAT2;
            if (BEAT16_played == 1){ //Chan0 - turn off Beat 4-4
                putchar(noteoff0);   putchar(notes[scale][i0]+offset0);
putchar(0x00); BEAT16_played = 0;
            }
            if (bigBEAT8_played == 1){ //Chan1 - turn off Beat [2-4]
                putchar(noteoff1);   putchar(notes[scale][i1]+offset1);
putchar(0x00); bigBEAT4_played = 0;
            }
            if (hugeBEAT4_played == 1){ //Chan2 - turn off Beat ((1-4))
                putchar(noteoff2);   putchar(notes[scale][i2]+offset2);
putchar(0x00); hugeBEAT4_played = 0;
            }
            //=====
            if (beatgo0[0] == 1) { //Chan0 - Play Beat 1-1
                i0 = pitch(i0, nwindow0);
                putchar(noteon0);   putchar(notes[scale][i0]+offset0);
putchar(def_vol0);
            }
            if (beatgo1[0] == 1) { //Chan1 - Play Beat [1-1]
                i1 = pitch(i1, nwindow1);
                putchar(noteon1);   putchar(notes[scale][i1]+offset1);
putchar(def_vol1);
            }
            if (beatgo2[0] == 1) { //Chan2 - Play Beat ((1))
                i2 = pitch(i2, nwindow2);
                putchar(noteon2);   putchar(notes[scale][i2]+offset2);
putchar(def_vol2);
            }
            PORTB = 0xFE;
        }
        break;
    case BEAT2:
        if(count++ == 5){
            //printf(" BEAT2 ");

```

```

        count=0;
        state = BEAT3;
        if (beatgo0[0] == 1) {      //Chan0 - Turn off Beat 1-1
            putchar(noteoff0);    putchar(notes[scale][i0]+offset0);
putchar(0x00);
        }
//=====
        if (beatgo0[1] == 1) {      //Chan0 - Play Beat 1-2
            i0 = pitch(i0, nwindow0);
            putchar(noteon0);    putchar(notes[scale][i0]+offset0);
putchar(def_vol0);
        }
        PORTB = 0xFD;
    }
    break;
case BEAT3:
    if(count++ == 5){
//        printf(" BEAT3 ");
        count=0;
        state = BEAT4;
        if (beatgo0[1] == 1) {      //Chan0 - turn off Beat 1-2
            putchar(noteoff0);    putchar(notes[scale][i0]+offset0);
putchar(0x00);
        }
        if (beatgo1[0] == 1) {      //Chan1 - turn off Beat [1-1]
            putchar(noteoff1);    putchar(notes[scale][i1]+offset1);
putchar(0x00);
        }
//=====
        if (beatgo0[2] == 1) {      //Chan0 - Play Beat 1-3
            i0 = pitch(i0, nwindow0);
            putchar(noteon0);    putchar(notes[scale][i0]+offset0);
putchar(def_vol0);
        }
        if (beatgo1[1] == 1) {      //Chan1 - Play Beat [1-2]
            i1 = pitch(i1, nwindow1);
            putchar(noteon1);    putchar(notes[scale][i1]+offset1);
putchar(def_vol1);
        }
        PORTB = 0xFB;
    }
    break;
case BEAT4:
    if(count++ == 5){
//        printf(" BEAT4 ");
        count=0;
        state = BEAT5;
    }
}

```

```

        if (beatgo0[2] == 1) {      //Chan0 - turn off Beat 1-3
            putchar(noteoff0);    putchar(notes[scale][i0]+offset0);
putchar(0x00);
        }
//=====
        if (beatgo0[3] == 1) {      //Chan0 - Play Beat 1-4
            i0 = pitch(i0, nwindow0);
            putchar(noteon0);    putchar(notes[scale][i0]+offset0);
putchar(def_vol0);   BEAT4_played = 1;
        }
        beat_found0 = 0;          //look for next rhythm 2-x
    }
    PORTB = 0xF7;
    break;
case BEAT5: //----- 1/4 Measure -----
    if(count++ == 5){
//        printf(" BEAT5 ");
        count=0;
        state = BEAT6;
        if (BEAT4_played == 1){      //Chan0 - turn off Beat 1-4
            putchar(noteoff0);    putchar(notes[scale][i0]+offset0);
putchar(0x00); BEAT4_played = 0;
        }
        if (beatgo1[1] == 1) {      //Chan1 - turn off Beat [1-2]
            putchar(noteoff1);    putchar(notes[scale][i1]+offset1);
putchar(0x00);
        }
        if (beatgo2[0] == 1) {      //Chan2 - turn off Beat ((1-1)))
            putchar(noteoff2);    putchar(notes[scale][i2]+offset2);
putchar(0x00);
        }

//=====
        if (beatgo0[0] == 1) {      //Chan0 - Play Beat 2-1
            i0 = pitch(i0, nwindow0);
            putchar(noteon0);    putchar(notes[scale][i0]+offset0);
putchar(def_vol0);
        }
        if (beatgo1[2] == 1) {      //Chan1 - Play Beat [1-3]
            i1 = pitch(i1, nwindow1);
            putchar(noteon1);    putchar(notes[scale][i1]+offset1);
putchar(def_vol1);
        }
        if (beatgo2[1] == 1) {      //Chan2 - Play Beat ((1-2))
            i2 = pitch(i2, nwindow2);
            putchar(noteon2);    putchar(notes[scale][i2]+offset2);
putchar(def_vol2);
    }

```

```

        }
        PORTB = 0xEF;
    }
    break;
case BEAT6:
    if(count++ == 5){
//        printf(" BEAT6 ");
        count=0;
        state = BEAT7;
        if (beatgo0[0] == 1) { //Chan0 - turn off Beat 2-1
            putchar(noteoff0); putchar(notes[scale][i0]+offset0);
putchar(0x00);
        }
//=====
        if (beatgo0[1] == 1) { //Chan0 - Play Beat 2-2
            i0 = pitch(i0, nwindow0);
            putchar(noteon0); putchar(notes[scale][i0]+offset0);
putchar(def_vol0);
        }
        PORTB = 0xDF;
    }
    break;
case BEAT7:
    if(count++ == 5){
//        printf(" BEAT7 ");
        //beat_found1 = 0; //look for next rhythm [2-x]
        count=0;
        state = BEAT8;
        if (beatgo0[1] == 1) { //Chan0 - turn off Beat 2-2
            putchar(noteoff0); putchar(notes[scale][i0]+offset0);
putchar(0x00);
        }
        if (beatgo1[2] == 1) { //Chan1 - turn off Beat [1-3]
            putchar(noteoff1); putchar(notes[scale][i1]+offset1);
putchar(0x00);
        }
//=====
        if (beatgo0[2] == 1) { //Chan0 - Play Beat 2-3
            i0 = pitch(i0, nwindow0);
            putchar(noteon0); putchar(notes[scale][i0]+offset0);
putchar(def_vol0);
        }
        if (beatgo1[3] == 1) { //Chan1 - Play Beat [1-4]
            i1 = pitch(i1, nwindow1);
            putchar(noteon1); putchar(notes[scale][i1]+offset1);
putchar(def_vol1); bigBEAT4_played = 1;
        }
    }
}

```

```

beat_found1 = 0;      //look for next rhythm [2-x]
PORTB = 0xBF;
}
break;
case BEAT8:
//    if(count++ == 5){
//        printf(" BEAT8 ");
//        count=0;
//        state = BEAT9;
//        if (beatgo0[2] == 1) {      //Chan0 - turn off Beat 2-3
//            putchar(noteoff0);    putchar(notes[scale][i0]+offset0);
putchar(0x00);
}
//=====
if (beatgo0[3] == 1) {      //Chan0 - Play Beat 2-4
i0 = pitch(i0, nwindow0);
putchar(noteon0);    putchar(notes[scale][i0]+offset0);
putchar(def_vol0);    BEAT8_played = 1;
}
beat_found0 = 0;      //look for next rhythm 3-x
PORTB = 0x7F;
}
break;
case BEAT9: //----- 1/2 Measure -----
if(count++ == 5){
//    printf(" BEAT9 ");
//    count=0;
state = BEAT10;
if (BEAT8_played == 1){      //Chan0 - turn off Beat 2-4
putchar(noteoff0);    putchar(notes[scale][i0]+offset0);
putchar(0x00); BEAT8_played = 0;
}
if (bigBEAT4_played == 1){ //Chan1 - turn off Beat [1-4]
putchar(noteoff1);    putchar(notes[scale][i1]+offset1);
putchar(0x00); bigBEAT4_played = 0;
}
if (beatgo2[1] == 1) {      //Chan2 - turn off Beat ((1-2))
putchar(noteoff2);    putchar(notes[scale][i2]+offset2);
putchar(0x00);
}
//=====
if (beatgo0[0] == 1) {      //Chan0 - Play Beat 3-1
i0 = pitch(i0, nwindow0);
putchar(noteon0);    putchar(notes[scale][i0]+offset0);
putchar(def_vol0);
}
if (beatgo1[0] == 1) {      //Chan1 - Play Beat [2-1]

```

```

        i1 = pitch(i1, nwindow1);
        putchar(noteon1);    putchar(notes[scale][i1]+offset1);

putchar(def_vol1);
}
if (beatgo2[2] == 1) {           //Chan2 - Play Beat ((1-3))
    i2 = pitch(i2, nwindow2);
    putchar(noteon2);    putchar(notes[scale][i2]+offset2);
}

putchar(def_vol2);
}
PORTB = 0xFE;
}
break;

case BEAT10:
if(count++ == 5){
//    printf(" BEAT10 ");
    count=0;
    state = BEAT11;
    if (beatgo0[0] == 1) {           //Chan0 - turn off Beat 3-1
        putchar(noteoff0);    putchar(notes[scale][i0]+offset0);
    }
    putchar(0x00);
}
//=====
if (beatgo0[1] == 1) {           //Chan0 - Play Beat 3-2
    i0 = pitch(i0, nwindow0);
    putchar(noteon0);    putchar(notes[scale][i0]+offset0);
}

putchar(def_vol0);
}
PORTB = 0xFD;
}
break;

case BEAT11:
if(count++ == 5){
//    printf(" BEAT11 ");
    count=0;
    state = BEAT12;
    if (beatgo0[1] == 1) {           //Chan0 - turn off Beat 3-2
        putchar(noteoff0);    putchar(notes[scale][i0]+offset0);
    }
    putchar(0x00);
}
//=====
if (beatgo1[0] == 1) {           //Chan1 - turn off Beat [2-1]
    putchar(noteoff1);    putchar(notes[scale][i1]+offset1);
}

putchar(0x00);
}
//=====
if (beatgo0[2] == 1) {           //Chan0 - Play Beat 3-3
    i0 = pitch(i0, nwindow0);
}

```

```

        putchar(noteon0);      putchar(notes[scale][i0]+offset0);
putchar(def_vol0);
}
if (beatgo1[1] == 1) {           //Chan2 - Play Beat [2-2]
    i1 = pitch(i1, nwindow1);
    putchar(noteon1);      putchar(notes[scale][i1]+offset1);
putchar(def_vol1);
}
PORTB = 0xFB;
}
break;
case BEAT12:
if(count++ == 5){
//printf(" BEAT12 ");
count=0;
state = BEAT13;
if (beatgo0[2] == 1) {           //Chan0 - turn off Beat 3-3
    putchar(noteoff0);      putchar(notes[scale][i0]+offset0);
putchar(0x00);
}
=====
if (beatgo0[3] == 1) {           //Chan0 - Play Beat 3-4
    i0 = pitch(i0, nwindow0);
    putchar(noteon0);      putchar(notes[scale][i0]+offset0);
putchar(def_vol0);  BEAT12_played = 1;
}
beat_found0 = 0;      //look for new rhythm 4-x
PORTB = 0xF7;
}
break;
case BEAT13://----- 3/4 Measure -----
if(count++ == 5){
//printf(" BEAT13 ");
//beat_found2 = 0;      //look for new rhythm ((1-x))
count=0;
state = BEAT14;
if (BEAT12_played == 1){ //Chan0 - turn off Beat 3-4
    putchar(noteoff0);      putchar(notes[scale][i0]+offset0);
putchar(0x00); BEAT12_played = 0;
}
if (beatgo1[1] == 1) {           //Chan1 - turn off Beat [2-2]
    putchar(noteoff1);      putchar(notes[scale][i1]+offset1);
putchar(0x00);
}
if (beatgo2[2] == 1) {           //Chan2 - turn off Beat ((1-3))
    putchar(noteoff2);      putchar(notes[scale][i2]+offset2);
putchar(0x00);
}

```

```

        }
//=====
        if (beatgo0[0] == 1) {          //Chan0 - Play Beat 4-1
            i0 = pitch(i0, nwindow0);
            putchar(noteon0);    putchar(notes[scale][i0]+offset0);
putchar(def_vol0);
        }
        if (beatgo1[2] == 1) {          //Chan2 - Play Beat [2-3]
            i1 = pitch(i1, nwindow1);
            putchar(noteon1);    putchar(notes[scale][i1]+offset1);
putchar(def_vol1);
        }
        if (beatgo2[3] == 1) {          //Chan3 - Play Beat ((1-4))
            i2 = pitch(i2, nwindow2);
            putchar(noteon2);    putchar(notes[scale][i2]+offset2);
putchar(def_vol2); hugeBEAT4_played = 1;
        }
        beat_found2 = 0;      //look for new rhythm ((1-x))
PORTB = 0xEF;
        }
        break;
case BEAT14:
    if(count++ == 5){
//        printf(" BEAT14 ");
        count=0;
        state = BEAT15;
        if (beatgo0[0] == 1) {          //Chan0 - turn off Beat 4-1
            putchar(noteoff0);    putchar(notes[scale][i0]+offset0);
putchar(0x00);
        }
//=====
        if (beatgo0[1] == 1) {          //Chan0 - Play Beat 4-2
            i0 = pitch(i0, nwindow0);
            putchar(noteon0);    putchar(notes[scale][i0]+offset0);
putchar(def_vol0);
        }
        PORTB = 0xDF;
        }
        break;
case BEAT15:
    if(count++ == 5){
//        printf(" BEAT15 ");
        //beat_found1 = 0;      //look for new rhythm [1-x]
        count=0;
        state = BEAT16;
        if (beatgo0[1] == 1) {          //Chan0 - turn off Beat 4-2

```

```

        putchar(noteoff0);    putchar(notes[scale][i0]+offset0);
putchar(0x00);
}
if (beatgo1[2] == 1) {      //Chan1 - turn off Beat [2-3]
    putchar(noteoff1);    putchar(notes[scale][i1]+offset1);
putchar(0x00);
}
=====

if (beatgo0[2] == 1) {      //Chan0 - Play Beat 4-3
    i0 = pitch(i0, nwindow0);
    putchar(noteon0);    putchar(notes[scale][i0]+offset0);
putchar(def_vol0);
}
if (beatgo1[3] == 1) {      //Chan1 - Play Beat [2-4]
    i1 = pitch(i1, nwindow1);
    putchar(noteon1);    putchar(notes[scale][i1]+offset1);
putchar(def_vol1);    bigBEAT8_played = 1;
}
beat_found1 = 0;      //look for new rhythm [1-x]
PORTB = 0xBF;
}
break;

case BEAT16:
if(count++ == 5){
//printf(" BEAT16 ");
count=0;
state = BEAT1;
if (beatgo0[2] == 1) {      //Chan0 - turn off Beat 4-3
    putchar(noteoff0);    putchar(notes[scale][i0]+offset0);
putchar(0x00);
}
=====
if (beatgo0[3] == 1) {      //Chan0 - Play Beat 4-4
    i0 = pitch(i0, nwindow0);

    putchar(noteon0);
putchar(notes[scale][i0]+offset0);    putchar(def_vol0);    BEAT16_played = 1;
}
beat_found0 = 0;      //look for new rhythm 1-x
PORTB = 0x7F;
}
break;----- Measure Complete-----
end
end//----- end Playnotes
//----- INITIALISE -----
void initialize(void)
begin

```

```

//PORT SET-UP's
DDRA = 0xF5;          //Sensors 5 and 6 - [Pin0-Out Pin1-In] [Pin2-Out Pin3-In]
DDRB = 0xFF;          // LED's OUT
DDRC = 0x55;          //Sensors 1,2,3,4 - [Pin0-Out Pin1-In] [Pin2-Out Pin3-In]
                      [Pin4-Out Pin5-In] [Pin6-Out Pin7-In]
PORTB = 0x0F;

//SET UP TIMER 0      all regs are 8-bits!!!
TIMSK = 0x01;          //turn on timer 0 OVERFLOW interupt
TCCR0 = 0x03;          //prescale to clk/64
reload = 255-8;         // t0 interrupt @ .128 msec
                       //TEMPO: 16th = 128 usec x 150 x 5 =~ .1 s = 150 bpm
TCNT0 = reload;

//SET UP USART
UCSRB = 0x18;          //USART receive | transmit enabled
UCSRC = 0x00;          //USART Asynchronous mode (bit6 = 0)
UBRRH = 0x00;
UBRRL = 7;              //31.25 kBaud for 4MHz... for actual MIDI OUT
//                         //9600 baud for 4MHz... for debug
UBRRL = 25;

//INITIALIZE Variables
count = 0;              //counter for state machine -> tempo
time0 = 150;
state = BEAT1;
d_state = RESET;
bbb = 0;
in_dist = 0;
putchar(noteon_perc); putchar(hibongo);    putchar(def_vol0+20);
#asm
    sei
#endasm
end//----- end Initialise

```