Warren Cheng
Rob Hendry
Ashwin Ramachandran

Embedded Systems Spring 2012 Design Document
3/20/12

MIDI-Controlled Synthesizer

I. Introduction

This paper documents the basic design of our MIDI-controlled synthesizer, which will take input stimuli from a MIDI controller (a keyboard) and produce synthesized tones. Our system will also have the capability to further manipulate the synthesized notes by sending the signals through a digital signal processor (DSP). This design essentially has three main components: decoding of the MIDI input according to the MIDI 1.0 protocol, designing hardware to synthesize the notes, and creating a sufficient DSP to manipulate the synthesized signals. The final system will be constructed using a combination of C programming on a NIOS II processor and specialized hardware built using VHDL.

II. MIDI Input Basics and Decoding

Traditionally, MIDI control signals are passed through a specialized MIDI connector. However, since we are implementing our design on the Altera DE-2 board, we will be using a MIDI to RS-232 converter to receive the signals on the board. The MIDI 1.0 Specification lists a recommended circuit for conversion of a MIDI output to a TTL signal through the use of an opto-isolator. This circuitry is necessary because MIDI-signals are transmitted through current loops as opposed to voltage levels. An opto-isolator is used because MIDI currents are provided without a ground reference. In addition, we found the *TerrorMouse* project used a MAX232 IC to convert the TTL levels into RS-232 levels (see figure 1).

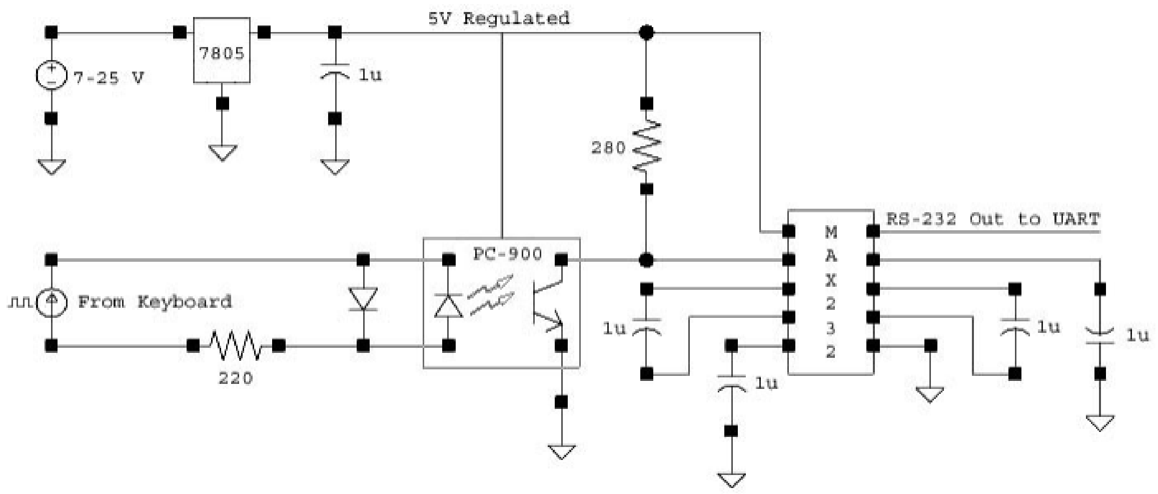
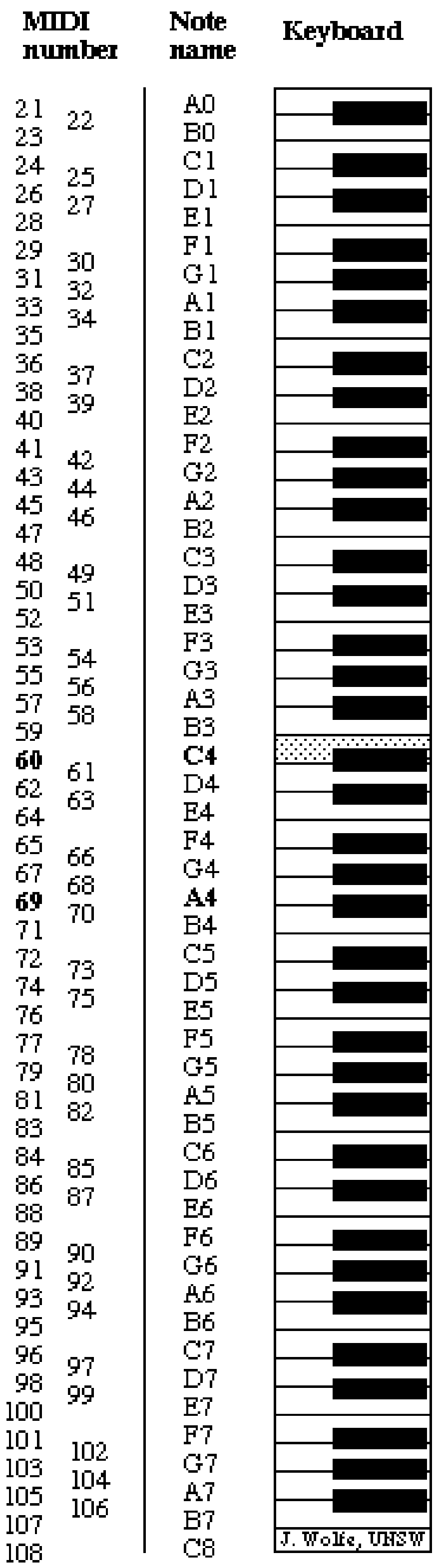


Figure 1. MIDI-in to RS-232 Conversion circuit


Keyboard
J. Wolfe, UNSW

MIDI number		Note name
21	22	A0
23		B0
24	25	C1
26	27	D1
28		E1
29	30	F1
31	32	G1
33	34	A1
35		B1
36	37	C2
38	39	D2
40		E2
41	42	F2
43	44	G2
45	46	A2
47		B2
48	49	C3
50	51	D3
52		E3
53	54	F3
55	56	G3
57	58	A3
59		B3
60	61	**C4**
62	63	D4
64		E4
65	66	F4
67	68	G4
69	70	**A4**
71		B4
72	73	C5
74	75	D5
76		E5
77	78	F5
79	80	G5
81	82	A5
83		B5
84	85	C6
86	87	D6
88		E6
89	90	F6
91	92	G6
93	94	A6
95		B6
96	97	C7
98	99	D7
100		E7
101	102	F7
103	104	G7
105	106	A7
107		B7
108		C8

Figure 3. MIDI signal to note chart

The portion of the circuit in the top left corner of figure 1 provides a regulated 5V supply for the MAX232 chip. The bottom left corner displays the circuit suggested by the MIDI 1.0 protocol for the current loop to voltage level conversion. Finally the MAX232 chip itself converts the input voltages to an RS-232 signal for use on the DE-2 board.

The last piece of the MIDI interface is a MIDI UART, which is responsible for serial to parallel conversion. As each data bit is serially transmitted via RS-232, it will be stored in a shift register, which will finally be read by the processor after an entire 8-bit MIDI message is received. Figure 2 shows a block diagram for the overall MIDI input interface.

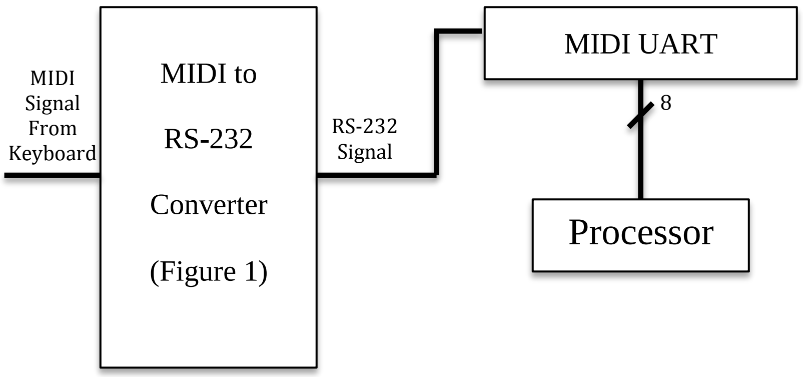

MIDI Signal From Keyboard
MIDI to RS-232 Converter (Figure 1)
RS-232 Signal
MIDI UART
8
Processor

Figure 2. Block diagram for the MIDI input interface

The basic MIDI 1.0 protocol says that a MIDI message begins with a status signal. This 8-bit signal indicates whether the controller wants to turn a note on, off, change control, change the program (change the instrumental sound), and so on. The upper nibble of a status signal indicates the operation, and the lower nibble indicates the channel to perform the operation on. We will be ignoring the lower nibble of the input, thereby enabling only a single channel.

In order to play a note, the note-on status signal (with its corresponding channel) is transmitted, followed by two more bytes. The first byte is the note itself (where the upper nibble is a number from 0-127, and the lower nibble is the channel) and the second is the velocity (or how long the note is held out). Each number between 0 and 127 is mapped to one note according to the MIDI protocol (see figure 2).

The rest of the different statuses operate along a similar principle. For the purposes of our synthesizer, we will only be implementing note on, note off, and change of program events. A note on event will generate a signal. A note off event will stop the sound. Finally, the change of program will signal the DSP to manipulate the sound of the signal. Because of this, the features that are not implemented will be ignored by the system. The MIDI-decoding will take place on the NIOS-II processor and be programmed in C. It will then send output control signals to either the FM-synthesizer logic (to turn the note on or off), or to the DSP for sound manipulation.

III. FM Synthesis

Our synthesizer will use the FM synthesis method for producing tones as introduced by John Chowing. Given a digitally encoded pitch (from the MIDI port), our FM synthesizer will be capable of producing a large variety of timbres by combining carriers, modulators, and envelopes. *Carriers* are simply digital signals that oscillate at some desired frequency, which produces a tone when fed to a DAC and a speaker. Carriers can be thought of as signals which oscillate at the fundamental frequency of the desired pitch or at a harmonic of the fundamental, though this is not necessarily required. *Modulators* refer to oscillating signals which are added to carriers to produce more interesting timbres, adding harmonics to the carrier frequency. The combination of these two signals is the essence of FM synthesis. Figure 1 illustrates the effect of frequency modulation. *Envelopes* are functions of time which control the volume of a modulated frequency. Envelopes are used to create, for example, the attack and release (i.e., the loud start and decaying finish) of a note that is often characteristic of real instruments. Following the terminology found in the user manual of an early digital synthesizer, the Yamaha DX7, we can define a basic FM operator which will serve as a building block for different tones our synthesizer will make.

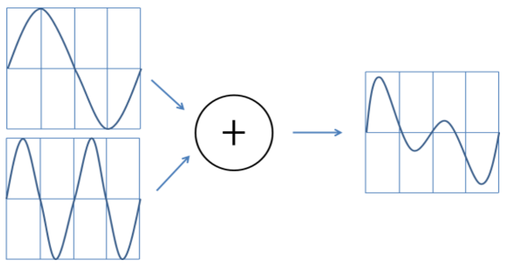


Figure 4. Example of adding two frequencies (e.g., a modulator to a carrier).

FM Operator. An FM operator consists of an FM signal adder and an amplifier. The operator takes as input pitch data (frequency of the carrier), modulation data (digital modulator signal), and envelope data (an envelope "program"), and outputs a synthesized tone. The FM signal adder combines the pitch data and modulation data according to the function:

$$x(t) = \sin\big(\omega_c t + I\sin(\omega_m t)\big)$$

where x(t) is the resulting amplitude at time t (generating a digital signal of the desired tone), ω_c is the frequency of the carrier, ω_m is the frequency of the modulator, and I is the modulation "depth." The output timbre can be varied by changing the modulation depth. An amplifier is placed on the output of this function to control its volume. The amplifier can be programmed with envelope data to create effects over time such as vibrato or decay. Figures 2 and 3 illustrate the basic layout of an FM operator and an example output signal given a decay envelope.

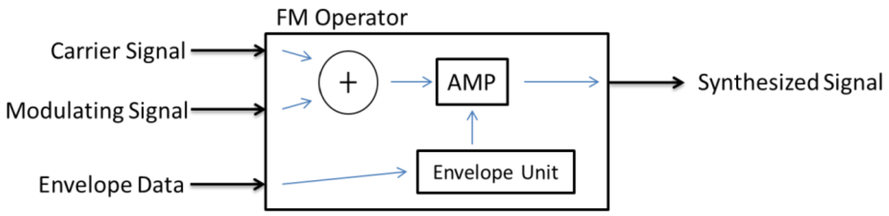


Figure 5. Diagram of FM operator.

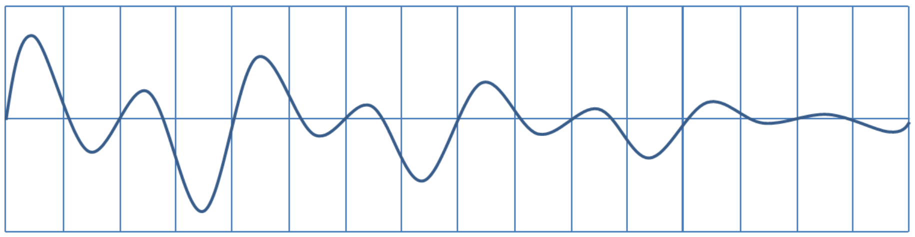


Figure 6. Example output signal of FM operator: the output of figure 1 with a decaying envelope which decreases the amplitude of the signal over time.

Constructing Synthesis Algorithms. To produce complex timbres with many harmonics, we can do three things:

1. Cascade FM operators, so that one operator's output becomes the modulation signal for the next
2. Sum the outputs of parallel FM operators
3. Feedback the output of operators into themselves (or other operators above them in the cascade)

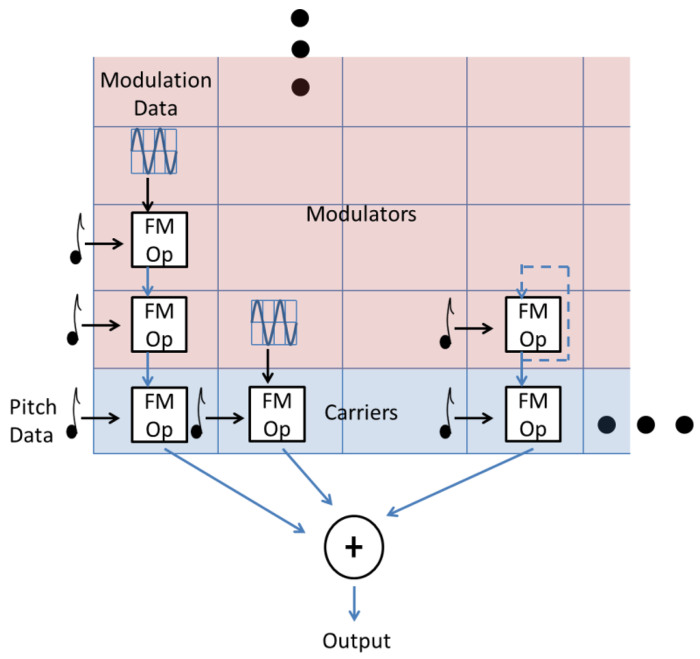


Figure 7 illustrates the degrees of freedom we have when arranging FM operators. We can call the bottom “row” of operators carriers, since they provide the last (and most likely the most influential) pitch before the final signal is synthesized. Not shown in this figure is the envelope input to each FM operator.

ADSR Envelopes. In our synthesizer, each FM operator will be able to dynamically change the volume of its output. When applied to a carrier operator, the envelope can be used to emulate the plucking of a string, or the slowly decaying reverberations of a drum after it has been hit. When applied to a modulator operator, can dynamically change the amount of modulation which is applied to the carrier, providing yet another way to develop different tones. A common envelope function is called an ADSR envelope, and has four parameters: attack time, decay time, sustain time, and release time, as illustrated below. Additionally, we may also parameterize the amplitude of the attack, decay, sustain, and release in more detail. We will use this type of envelope in our FM operators.

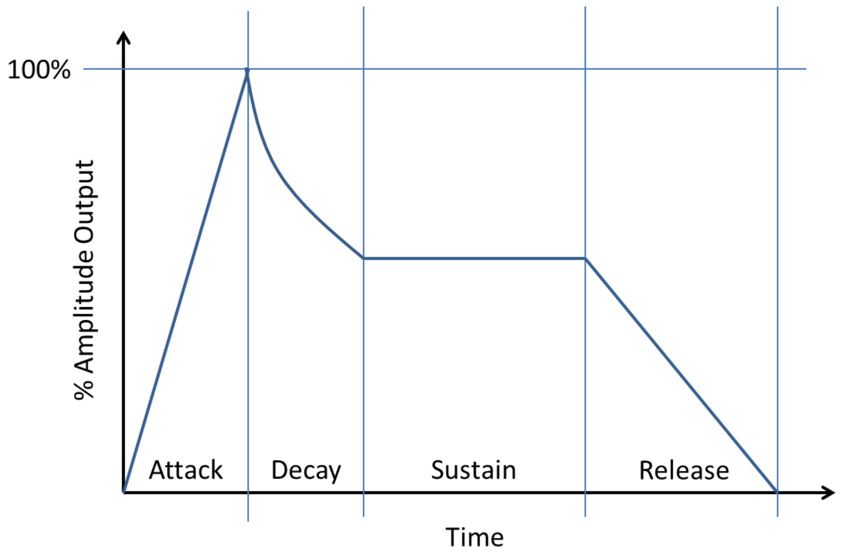


Figure 8 shows a picture of an example envelope. Here, when the note begins it has a quick burst of loudness (like plucking a string on a guitar), then sustains for a while (as in a string resonating), and then a decaying release (like the guitar body resonating for a while after the string stops).

IV. Vocoder

Background

Instead of recreating instruments and plucked strings like in the spring 2004 TerrorMouse project, we decided to try recreating a vocoder using the Altera DE2 board. A vocoder basically is a device which distorts a human speech input which can make it sound very metallic or robotic. It was originally used to encrypt speech for telecommunications, but now used popularly as a musical instrument. Basically the two signals that are required for vocoder operation is a carrier signal which usually comes from a harmonically rich source such as an organ or synthesizer and a modulating signal which the input that comes from a microphone. The way it works it that it decomposes human speech into a number of frequency bands which is later passed through a series of band pass filters. These band pass filters have the ability to suppress some frequencies of human speech while accentuating others. Afterwards, these signals are multiplied by the corresponding frequency bands from a carrier source whose output is summed with other outputs representing other frequency bands.

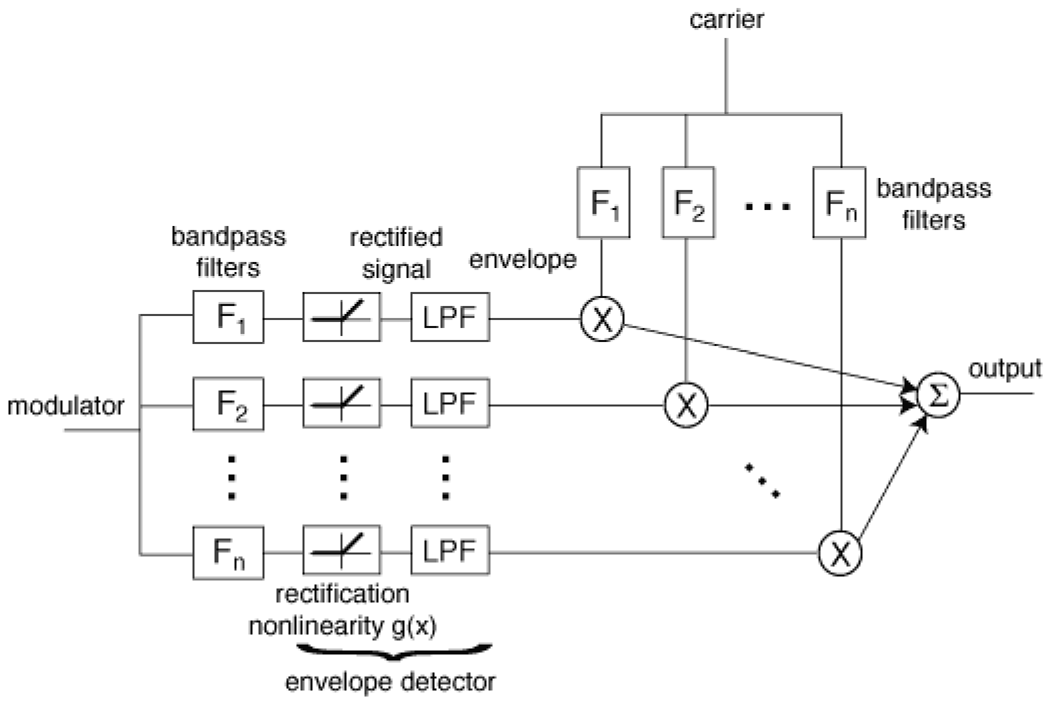


Figure 9 – A diagram representing how a channel vocoder works.

General Design Implementation of Vocoder

In our design we intend on taking in a microphone input handled by the line in jack of the 24 bit audio codec on the DE2 board. Since the sound produced by a human can go as high as 7 kHz, we have decided to set the sampling frequency of the audio codec to be 14 kHz in order to fulfill the Nyquist sampling theorem. This, however, may be too conservative of an estimate and perhaps we may decide to lower the sampling rate later on. The ADC of the codec should produce serial data which is then decomposed into its frequency components using the fast Fourier transform. We intend on implementing the FFT in software and plan on storing the FFT data for a fixed period in SRAM memory. Implementing the FFT in hardware is a possibility, but we are still not quite certain if the pros of this method necessarily outweigh the cons. We might get some sort of improvement in speed, but we're going to have a lot of digital data flowing through our system from the FFT and we might not have enough flip flips to use as buffers. Doing it in software appears to be a "cleaner" approach. Digital band pass filters can also then be implemented in C in order to separate the different bands of spectral components into different channels. Multiplication of the carriers signal components with the modulating voice signals components can easily be accomplished in code by multiplying the corresponding registers that hold the respective data and store the data into a new output register. Each spectral band corresponding to one of the several channels in the vocoder would have an output register which would be summed and stored in another register. We would then need to write a piece of code in software that carries out the inverse FFT in order to recover the modulated signal. This digital time-domain output would then be ideally fed back to the audio codec to be converted into an audio signal and outputted through the line out jack on the board.

Before we actually begin designing our FM synthesizer which will generate waveforms such as a sawtooth wave that will serve as our carrier signal, we will robustly test the effect of a certain carrier on a modulating sound signal in order to determine which carrier will help us to yield the best sounding robotic voices. We will accomplish this by rigorously testing the FM synthesis algorithm as well as its corresponding vocoder result.