Kai Hirota

SID: u7233149

Networked Music Sequencer

Design

The program is an implementation of the subset of the MIDI protocol, using three communication lines: control, clock, and data lines. Together, they enable a sender to transmit MIDI commands and parameters to a receiver. Each message transmission is a fixed 96 bit transmission, consisting of three 4-byte words - a command followed by MIDI note number and velocity parameters. This implementation of the MIDI protocol is only partial because this program only accepts two commands - note on, and note off. The sender uses the tim7 timer to control the length of interval between each message transmission, allowing each note to be played for any duration with precision limit of 0.01 second. The sender uses the three lines to send each 12 byte MIDI message, 1 bit at a time. Once transmission is complete, the receiver validates the received data and discards the data if it is corrupt. If data was successfully received, the receiver processes the MIDI message. The receiver will then perform velocity to amplitude, and MIDI note number to frequency conversion. Finally, the receiver plays or stops a note at specified frequency and amplitude. The MIDI note number to frequency conversion is done using 32-bit floating point registers and the floating point unit.

Implementation

MIDI message

Each MIDI message consists of three 4-byte words - command, MIDI note number, and velocity. Currently, the program only accepts two commands, which are note on and note off. If the first word is `0x80`, note on, and if the first word is `0x90`, then note off. MIDI commands are 8 bits long, and commands always begin with 1, like `0b1xxxxxxx`, and data always begin with 0, which is why both the MIDI note numbers and velocity range from 0 to 127.

Transmission and Control Flow

1. When the sender needs to send a message, it sends 1 on control line to trigger the control line interrupt on the receiver.
2. Receiver enables the clock line interrupt when rising edge is detected on control line, and disables the clock line interrupt when falling edge is detected on control line.
3. While the control line is on, the sender sets the bit on the data line, and then sends a 1 on the clock line. This triggers clock line interrupt on the receiver, which will then record the bit on the data line. This is repeated until all 96 bits are sent, 1 bit at a time.
4. When the sender has finished sending data, it clears the control line, which triggers the falling edge control line interrupt handler. In this interrupt, the receiver validates that all 96 bits were sent. If data was received correctly, the receiver processes the MIDI message before finishing the interrupt handler.

Flexible Interval

The sender uses the tim7 timer to control the interval between each message transmission. The tim7 interrupt handler is set to lowest priority so that the control line and clock line interrupt always trigger with higher priority. This ensures that every time the clock line is on, the receiver always records the data line bit immediately. Each call to the tim7 interrupt handler loads the next MIDI message to be sent, and also updates the duration of the next interval by resetting the current value reigster and writing to the reload value register.

MIDI note-to-frequency, and velocity-to-amplitude conversion

MIDI note is converted to frequency with precision up to 0.01Hz. It uses 32-bit floating point registers to perform accurate conversion. Since the MIDI format does not specify how velocity should be converted to amplitude, the program uses a simple conversion:

$$a = (a_{\max} // 127) \times v \tag{1}$$

where

- a: Amplitude
- $a_{\max}$: Maximum amplitude value allowed (`0×7FFF`)
- v: MIDI velocity parameter (0-127)
- // indicates use of integer division

Analysis

The program sends, receives, and executes MIDI messages by effectively using three communication lines and interrupt priorities. It is able to perform accurate MIDI note to frequency conversion and velocity to amplitude conversion.

The receiver, if at any time corrupted data is received, will turn on the red LED light to indicate that something has gone wrong. To make it easier to detect such errors, the red LED stays on even if the subsequent MIDI messages are transmitted correctly.

One improvement that should be implemented is to use `lrdh` and `strh` so that each of the three components of the MIDI message only requires 16 bit half-words. This will make the transmission faster, the input data size can be reduced, and most of all, faster transmission means less bits are needed, minimizing risk of data corruption. Since MIDI commands and parameters are only 8 bits long, even more space can be saved depending on the implementation. For example, using only 8 bits will allow an entire MIDI message to be stored in one 32 bit register, but this will also require more processing in order to unwrap the message. If even more space saving and faster transmission is desired, for example, the user can create a mapping between MIDI commands and bits. For example, the first bit can map to note on, or off depending on 0 or 1. Whatever improvement of modification is needed, the program is written with high modularization and with different levels of abstraction to make it easy to add or change features, and to understand dependencies between functions.

One shortcoming of this program is being a partial implementation that only accepts two MIDI commands: note on and note off. Since the program cannot play different notes simultaneously, the design could have included a feature to turn off the previous note whenever and new note on command is received. However, that undermines the purpose of having note off commands in MIDI protocol. Therefore, the input music had to be slightly modified when being converted to MIDI messages, by inserting a note off command after every note on command. This way, the receiver turns off the previous note and immediately plays the next note, and everything is done in correct MIDI format, using commands and parameters.