Design Document

1. Objective

I implemented a simple serial protocol for sending full packets of data to a receiver, which it could then interpret into a playable sound. This protocol utilized 3 lines: control, clock, and data wires.

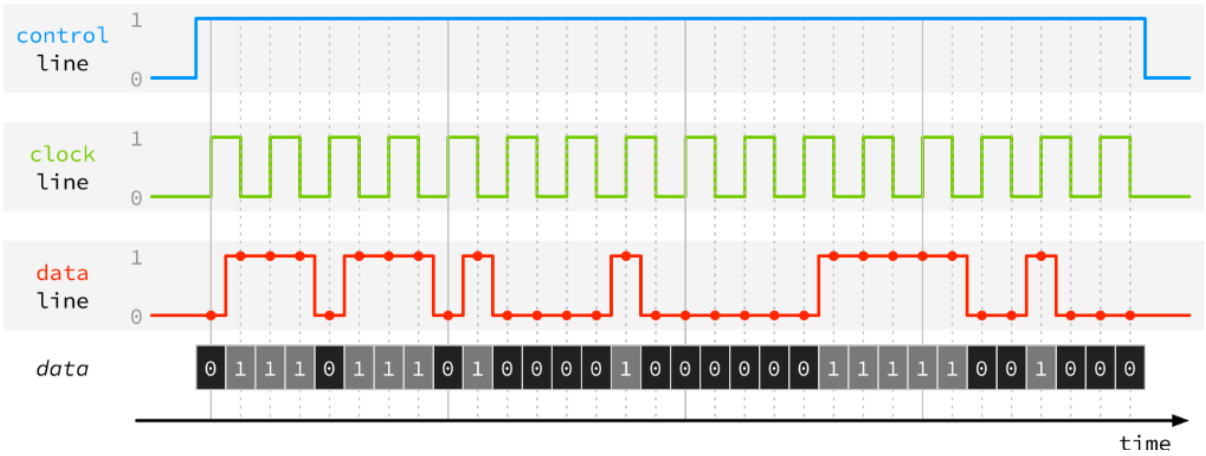

Figure 1: Simple serial protocol implementation (source: assignment specification)

When the sender was about to transmit a packet, it would send a rising edge across the control wire, so that the receiver could prepare to read the data wire. As the connection is synchronous, a clock wire would send a rising edge to indicate when the receiver should read from the data wire. This allows the receiver to compile the packet one bit at a time. Once the sender has finished transmitting all data, it would send a falling edge on the control wire. This would tell the receiver to start playing the sound from the transmitted packet.

2. Design

2.1 Packet Format

Each packet first contains a byte specifying the mode of the sound. A 0 indicates a delay (no sound), and in this case the mode byte is the only part of the packet. A non-zero number indicates a note, and different numbers are used for different waveforms (1 = sawtooth, 2 = sine, 3 = triangle).

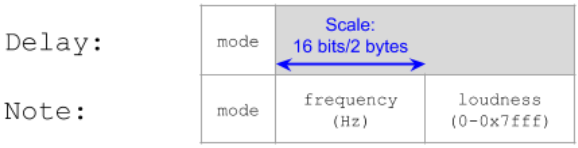

Figure 2: Packet format

For notes, there is another word following the initial mode byte. The first half word is used for the frequency (Hz) and the second half word is used for the loudness (amplitude). The receiver will play whatever sound is was previously sent, so timing is controlled by the sender. Using this packet format, notes can be played with different waveforms, frequencies, and loudness. By manipulating what loudness is played at each specific time, the sender can also create an amplitude envelope.

2.2 Serial Receiver

To implement the receiver, I allocated two full packets (in the format of figure 2) in memory. The first of these packets is used for the sound which is currently being played (`receiver_current`), which is interpreted by the primary loop of the program. This report will focus on the serial protocol rather than the digital sequencer which interprets this packet. Another packet is allocated for the next sound (`receiver_next`) which is compiled bit by bit as data is received. Once the entire packet is received, `receiver_next` replaces `receiver_current` as the sound to be played.

When a rising edge is detected on the control line, the most significant bit in `receiver_status` is set. Unless this bit is set, any rising edge interruptions on the clock line will be discarded. Clock interrupts will otherwise read the current bit in the data line (0 for low, 1 for high), and copy that bit to the respective index in `receiver_next`. The current index is stored in the other 7 bits of `receiver_status`, and incremented every bit read. Finally, when a falling edge is detected on the control line, the most significant bit in `receiver_status` is cleared, and `receiver_next` is copied to `receiver_current`. Note that for packets that indicate a delay (contain only 0s), only the first byte is used to save space, so only that byte is copied across.

2.3 Serial Sender

The sender operates on a timer (`tim7`), and transmits the next packet whenever the timer interrupt is triggered. It first fetches the data it needs to send from memory, then sends a rising edge on the control wire. To transmit each bit, the bit at the current index is read, the data line is set or cleared depending on the bit read, and finally a rising edge is sent on the clock. This is repeated until all bits of the packet are sent, then a falling edge is sent on the control wire. Note that for sending a delay, only a byte is transmitted.

3. Improvements

There are several potential improvements which can be made to this simple serial protocol. There is currently no means of error detection or correction. To help detect whether there is an error, a specific bit can be allocated within the sent packet (known as a parity bit). This bit can be set to 1 if the packet is equal to an even number, or 0 if an odd number. The receiver could then check that this bit matches. If it does not, this indicates that the packet has not been correctly received. While not full proof, as incorrect packets could still match or the error detection bit itself might not be received correctly, this would reduce the chance of errors. The receiver could then discard a packet with an error, or if another line were connected to allow the receiver to communicate back to the sender, it could request that the packet be resent.

The protocol could also be extended to allow harmony. This could be implemented in a similar method to the MIDI protocol, where different ‘instruments’ can be specified and manipulated independently. Additional waveforms could also be added, as there is further capacity in the mode byte (see figure 2).