Design Document

1. Objective

I aimed to improve my digital synthesizer by implementing ADSR (attack-decay-sustain-release) envelope functionality, and the capacity to use different waveforms or instruments. Data required to play an individual note would be stored in memory, and then read in sequence by my program, customizing various properties of the note.

An attack-decay-sustain-release envelope describes the change in loudness/volume (amplitude) over the lifetime of a note. Where a normal note (such as those played in part 1) has a constant loudness, an ADSR note is broken up into 4 stages. The first stage is the 'attack', where the note rises from silence to an initial peak loudness. Next, the loudness will 'decay' to lower value of loudness. Once this value is reached, the note will remain constant at this loudness and 'sustain'. Finally, the loudness will fade out back to silence, called the 'release'.

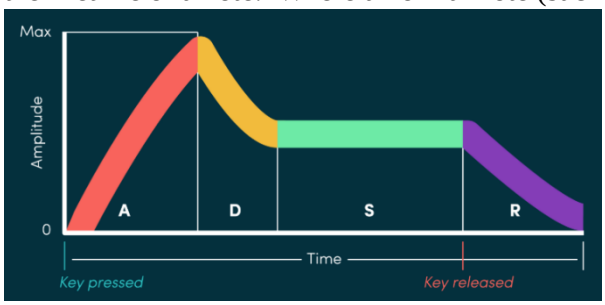

Figure 1: The shape of an ADSR envelope
Source: https://blog.landr.com/adsr-envelopes-infographic/

2. Design

2.1 ADSR Envelope Implementation

In my implementation, I opted to create a linear transition between each of the four stages of the envelope. I revised my original function for playing a note to take in parameters for an initial and final loudness. During setup, a constant is calculated that is added to the current amplitude every cycle, such that the final loudness is reached at the end of the duration of the note. This accounted for cases where the loudness would need to decrease (the calculation allowed for signed values, e.g. using `sdiv`), and where the loudness would stay constant (where initial and final loudness were equal). Different waveforms can still be generated in this method (see 2.2 for memory format), provided the correct parameters are accepted.

The function is called once for each stage for the envelope, with the frequency, respective initial and final loudness, and durations being fed as parameters. To complete the full envelope, a duration for each stage, loudness of the initial peak, and loudness of the sustain stage is required. While the sustain duration is usually determined by the amount of time a key is pressed (such as on a piano), it made more sense to specify it for a synthesizer. The parameters for each full envelope are stored in memory, and are read via the base address for each sound.

Care was taken to ensure that transitions between notes and silent delays were seamless. The initial approach was to disable any sound from being played during silence, however this led to audible imperfections and 'pops'. The next approach was to continue playing a constant sound with the value of the last sample until the next sound, creating a flat wave which was silent. However this would not always transition to the next sound smoothly, so instead the delay function would quickly transition the sample value to 0 before keeping it constant.

2.2 Memory Utilization

Having effective means to store different sequences of notes and silent delays was integral to the flexibility of my program. While a basic format was used for part 1, I sought to produce a wider range of different notes. Note that the first 2 bytes at the start of the song memory address indicates the total number of sounds (including delays) that are in the song.

Preceding every sound is a byte detailing the 'mode'. As indicated in figure 1, this details if the sound is a delay (`0`) or a note (`1`), if the note is normal (`0`) or has an ADSR envelope (`1`), and what the waveform or instrument the note is. Currently, there are only 2 different waveforms implemented - a triangle wave (`0`) and a sine wave (`1`) - but there is capacity for up to 16.

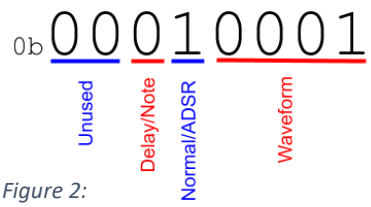

Figure 2: Mode format for an ADSR sine wave

Depending on the mode, a different format and size is used for the sound. For a delay, the only other parameter is duration. For a normal note, the frequency, loudness, and duration must be specified. A note with an ADSR envelope requires the most parameters, taking up a total of 15 bytes (see 2.1 for more details).

Delay:	mode	duration (ms)	Scale: 16 bits/2 bytes					
Note:	mode	frequency (Hz)	loudness (0-0x7fff)	duration (ms)				
Note (ADSR):	mode	frequency (Hz)	sustain loudness (0-0x7fff)	peak loudness (0-0x7fff)	attack duration (ms)	decay duration (ms)	sustain duration (ms)	release duration (ms)

Figure 3: Format for each type of sound (wide cells = 2 bytes, narrow cells = 1 byte)

Since different sounds require different amounts of data, only as much space as is needed is allocated. While this means the mode byte must be read in order to know the size of the current sound and therefore what the address of the next sound is, this is more efficient in terms of storage space.

3. Improvements

The majority of error in my program can be attributed to rounding imperfections. Checks to see if the duration of a note had been reached was done at the end of cycles, which could cause sounds to play for slightly longer than intended, especially for lower frequency waves. This could be solved by checking duration at more frequent intervals.

Additionally, especially low durations (less than 50ms) for attack or release stages of an ADSR envelope could lead to imperfect audio. This was likely due to the loudness being incremented at significant values and exceeding the dynamic range (`0x7fff`), so a resolution could be to check if loudness is within a reasonable range at each step.

Further functionality, such as new instruments, could also be added to the synthesizer. There are several unused bits in the mode byte which could prove useful for customizing the wave further. Different envelopes, such as an inverted envelope, or the option for a smooth curve instead of linear between ADSR stages, could enhance the possibilities of sounds produced by my synthesizer.