Design Document

1. Objective

I aimed to generate an approximation of a sine wave that would change frequency over time. Given a value for a base frequency in Hz (440Hz in my program) and a value for half of the target amplitude (where `0x7fff` would use the full dynamic range, which was used in my program), an approximate sine wave would be generated where the frequency would shift between two octaves above the base frequency ($base \times 2^2$), and two octaves below ($\frac{base}{2^2}$). Initially, every time a full period is completed, the frequency will increase by 1. This would happen until the upper bound (two octaves above base frequency) is reached, in which case the frequency would start to decrease by 1 every period, until the lower bound (two octaves below base frequency) is reached, and so on. This way the sound produced would appear to 'bounce' back and forth between the upper and lower bounds.

2. Design

2.1 Sine Wave Implementation

First, I needed to implement an approximate sine wave. It would only be approximate because there is a limited number of samples (times `BSP_AUDIO_OUT_Play_Sample` is called) every period, dependent on the frequency (for example, a lower frequency would increase the number of samples per period). At 440Hz, this gives us approximately 109 samples per period (with an output sample rate of 48kHz), so the curve of my wave would not be perfectly continuous or 'smooth' like a sine wave. Ultimately, there were also imperfections in my wave due to rounding errors, and a slight 'jump' at the end of each half period.

I chose to use Bhaskara I's sine approximation formula for my implementation. Defined as $\frac{4x(180-x)}{40500-x(180-x)}$, this formula gives an approximation of the first half period of a sine wave. In my implementation, I applied this formula at every sample point. I stored a value in a register (`r4`) which would increment every sample (effectively tracking time), then when using the formula I would convert this value to degrees and substitute it in the place of x. I also needed to scale the result to fit the amplitude, and this was done before the division, otherwise I would only get values between 0 and 1, which would be rounded towards 0 and lose information (when using `udiv`).

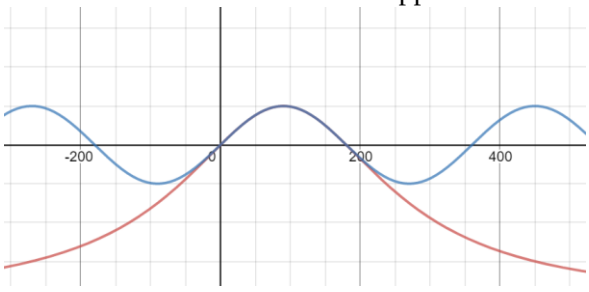

Figure 1: Bhaskra's formula (red) and a sine wave (blue)

So far, I could generate the first half period of an approximate sine wave. For the second half of the period, I would branch to a separate function when `r4` was greater than half of the samples per period (≈54 for 440Hz), reset `r4` to 0, and instead negate the result of Bhaskra's formula. This would effectively generate a complete period of an approximate sine wave, as the second half is the negation of the first. To repeat the period, I would branch back to the function for the first half once `r4` again reached half of the samples per period, and again reset `r4` to 0.

Initially, my wave would get cut off before it had completed an entire half cycle, resulting in a large 'jump' before crossing the vertical axis. To counteract this, I would increment `r4` before the first sample was taken, removing a point at 0, essentially giving the wave a head start and making the wave more even overall. I was also able to pinpoint a large amount of error being generated by the conversion from the counter (`r4`) to degrees. To reduce this error, I performed a logical shift left ($\times 2^2$) on values while I was performing the calculation to ensure less information was lost, then shifting them back once the calculation was done. Ultimately, when compared to a perfect sine wave by sound, my wave does appear slightly 'harsher', though it sounds far closer to a sine wave than a triangle wave.

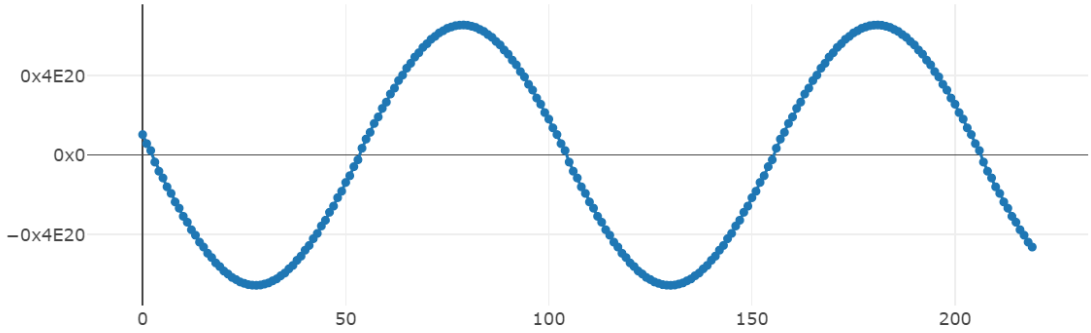

Figure 2: My approximate sine wave (440Hz, amplitude/2 of `0x7fff`)

2.2 Frequency Over Time Implementation

In order to change frequency over time, several values needed to be recalculated. Notably, the samples made per period would have to change (thus changing the length of a period and the frequency), and the factor used to convert the counter (`r4`) to degrees. To achieve this, before every new period the frequency is incremented or decremented, and these values are recalculated. I also chose to set bounds at +/- two octaves from the base frequency to create an audible sound that would modulate between different octaves of a given note (for example, for note A4 the sound would go between 110, 220, 440, 880 and 1760 hertz).

3. Improvements

While my wave makes a reasonable approximation of a sine wave, there are still observable imperfections. Whenever the wave crosses the *x* axis, the distance between sample points is notably larger. This is likely due to my attempt to create a more symmetrical wave by pre-incrementing the time counter (`r4`). A more ideal solution might be found with further work reducing any rounding errors.

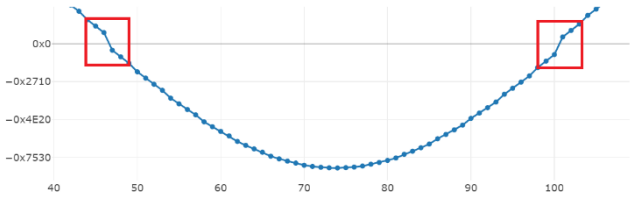

Figure 3: Red squares indicate observable imperfections in my wave (440Hz)

I also considered calculating values needed for one full period initially, storing them in memory, and reading the results from memory to save on having to make calculations for every sample (functionally a lookup table). However this proved to be impractical once I began modulating the frequency, as each period was unique. Nonetheless, a more sophisticated lookup table might be possible, storing values needed for different periods of different frequencies, which could reduce unnecessary calculations in my code.