Polyphonic Sound Generation

ECE 362
https://engineering.purdue.edu/ee362/

© Rick
Generation of a wave table

- To generate a single cycle of a sine wave, use a program like this:

```c
// genwave.c
#include <stdio.h>
#include <math.h>
define N 1024
int main(void) {
    int x;
    printf("const short int wavetable[%d] = {\n", N);
    for(x=0; x<N; x++) {
        int value = 32767 * sin(2 * M_PI * x / N);
        printf("%d, ", value);
        if ((x % 8) == 7) printf("\n");
    }
    printf("};\n");
}
```
Running genwave.c

• Compile and run the genwave program:

  gcc -o genwave genwave.c -lm
  ./genwave > wave.c
The wave table

• The wave table will be a single array
  – Values range from -32767 to 32767 which nearly the maximum range that can be specified with a short int (AKA int16_t if you #include <stdint.h>).
  – The array is declared with “const” so that the array is placed in the Flash ROM rather than in RAM.
Sine wave output

• Try using the wave table to generate a signal using the DAC output.
  – The wave table contains values between -32767 and 32767, but the DAC can, at best, output values between 0 and 4095.
    • It is important to leave the sine wave in a range centered on zero so that we can later add multiple samples together.
    • To output to the DAC take each sample, divide it by 16, and add 2048.
  – A single traversal of the entire wave table will result in a single cycle of a sine wave.
  – Repeated traversal will show a continuous sine wave.
Determining frequency: simple

- Consider a single cycle of a sine wave with 100K samples, used with a DAC sample frequency of 100K/sec.

- This would produce a 1Hz output.
  - Setting the DAC rate faster would produce a higher frequency.
  - Setting the DAC rate slower would produce a lower frequency.
Limitations

• Adjusting the DAC rate allows very crude refinements to the output frequency.
  – To get a 2Hz wave, set the DAC rate to 200Ksamples/sec.
  – To get a 3Hz wave, set the DAC rate to 300Ksamples/sec.
  – There is no way to use this scheme to produce a 2.5Hz output signal.
  – There’s a limit to how fast we can drive the DAC.
Determining frequency: step size

• What if, instead of taking every sample, we take every other sample of the wave table with a 100K-entry wavetable and a 100Ksample/sec DAC rate?

• This would still produce a sine wave, but
  - now the frequency would be 2Hz.
  - Taking every Nth step produces a wave of N Hz.
Generalized frequency calculation

- With a DAC rate of $F_{DAC}$, a single-cycle wavetable of $N$ samples, and a step size of $S$, a general formula for the frequency, $f$, of the signal output by the DAC is:

$$ f = S \times \frac{F_{DAC}}{N} $$
Example for a DAC ISR

- Construct an ISR to output a signal using a step size like this. The ‘offset’ variable keeps track of the last sample output from the wavetable.

```c
const short int wavetable[N] = { ... };

int offset = 0;
int step = 440;

void ISR(void) {
    offset += step;
    if (offset >= N) // If we go past the end of the array,
        offset -= N; // wrap around with same offset as overshoot.
    int sample = wavetable[offset]; // get a sample
    sample = sample / 16 + 2048; // adjust for the DAC range
    DAC->DHR12R1 = sample;
    // trigger DAC here.
}
```
Limitations of stepping

- It’s still easy to produce an output frequency with an integer multiple of 1Hz.
  - Easy to produce 440 Hz output.
  - Still no way to get 466.164 Hz with this scheme.
    - Why would we want 466.164 Hz?
- We’d like to be able to step by *fractional* amounts.
Musical notes

• The “standard” frequency for the ‘A’ above middle C is 440 Hz.
• Each octave represents a doubling of frequency, so:
  – The ‘A’ below middle C has a frequency 220 Hz.
  – The ‘A’ above middle C has a frequency 880 Hz.
• What about the notes in between?
  – There are 12 notes in an octave.
  – Each note is then $2^{1/12}$ (about 1.05946) higher than the previous one.
  – The A# above middle C is then about $440 \times 1.05946 = 466.164$ Hz.
  – The B above middle C is then about $440 \times 105946^2 = 493.883$ Hz.
  – Middle C is $220 \times 105946^3 = 261.626$ Hz (three steps above 220).
  – See page 520 of your text for a handy table.
More limitations of stepping

- Difficult to use a table of 100K samples with only 64Kbytes of Flash ROM.
  - You can do this by storing only \( \frac{1}{4} \) of the wavetable and then flipping it around to produce the other \( \frac{3}{4} \) portions.
    - Your textbook has an example of this on pp 512 – 515.
- Using a step size near the sample size is not going to work.
  - If \( S = N/2 \), you would always output zeros.
  - If \( S = N/4 \), you would output a triangle wave.
  - \( S \) should probably be less than \( N/20 \) to produce anything reasonable.
    - Only possible when we have a large \( N \).
Stepping with fixed-point math

• It’s tempting to use a floating-point step size and offset to refer to wavetable samples.
  – Our microcontroller has no floating-point math hardware, so this can be very slow. Avoid it.

• We can use fixed-point arithmetic to do it (almost) as quickly as integers.
Steps still work with fractions

- If we make fractional steps, we just round down to the next lower integer to get the offset into the wavetable array. E.g., a step of 4.5 would look like:

- Practically, that means taking steps of 4, 5, 4, 5, and so on.
  - This might be quite as clean of a sine wave, but you won’t notice.
  - Other fractional amounts, like 440.164, would work just as well.
Benefit of fractional stepping

- If we can have fractional stepping, we can also use a smaller wave table size. The formula still works. E.g., for $N = 1000$, $F_{DAC} = 100K$, $S=4.66164$

$$f = S \times F_{DAC} / N = 4.66164 \times 100000 / 1000 = 466.164 \text{ Hz}$$
Implementing fractional stepping

- When using fixed-point arithmetic, you generally choose what part of an integer you want to be the whole number and use the other part as a fraction.
  - Let’s use a 32-bit integer to represent a 16-bit whole number and a 16-bit fraction. (The nomenclature for this fixed-point format is “Q16.16”.)
  - To encode a fixed-point number, we specify it with a floating-point number multiplied by the amount we shift for the fractional part. E.g., to encode 466.164 in Q16.16, we say:
    - int step = 466.164 * (1<<16);
Q16.16 examples

• We can imagine there’s a “binary point” between the whole part of the number and the fractional part of the 32-bit number:
  
  2.5 = 0000000000000010.1000000000000000
  
  4.75 = 0000000000000100.1100000000000000
  
  466.164 = 0000000110111110.001010011111011

• To get the whole part, we just shift it right by 16.
Example ISR with fractional steps

// Assume N = 1000 and the DAC rate is 100K/sec...

const short int wavetable[N] = { ... };

int offset = 0;
int step = (4.93883 * (1<<16)); // play the ‘B’ above middle ‘C’ (493.883 Hz)

void ISR(void) {
    offset += step;
    if ((offset>>16) >= N) // If we go past the end of the array,
        offset -= N<<16;  // wrap around with same offset as overshoot.
    int sample = wavetable[offset>>16]; // get a sample
    sample = sample / 16 + 2048;        // adjust for the DAC range
    DAC->DHR12R1 = sample;
    // trigger DAC here.
}
Mixing notes

- Everything our ears hear is the result of constructive and destructive interference between multiple frequency sources.
  - Our ears are insensitive to phase of the frequencies, therefore it’s easy to mix multiple frequencies together.
  - We can take wavetable samples and **ADD** them.
    - Do this by maintaining separate offsets and steps for each note.
    - By stepping through the same wavetable at different rates, you effectively produce two different notes.
  - This works as long as wavetable samples are centered on zero.
• When adding two different audio sources, sometimes the highs or lows of samples will occur at the same time.
  – This will be a larger value than we can produce with the DAC.
  – We don’t want to write a value too large (or negative) to the DAC holding register because it will not represent an audio level that makes sense.  e.g., 4099 == (4096 + 30), when written to the DAC, would be output as a 30.
  – We cut our output amplitude in half by dividing by 32 instead of 16.
  – And we ‘clip’ the values that are too high or too low:
    
    ```c
    if (sample > 4095)
        sample = 4095;
    else if (sample < 0)
        sample = 0;
    ```


ISR with multiple notes

// Assume N = 1000 and the DAC rate is 100K/sec...
const short int wavetable[N] = { ... };

int offset1 = 0;
int offset2 = 0;
int step1 = 2.61626 * (1<<16); // Middle ‘C’ (261.626 Hz)
int step2 = 3.29628 * (1<<16); // The ‘E’ above middle ‘C’ (329.628 Hz)

void ISR(void) {
    offset1 += step1;
    if ((offset1>>16) >= N) // If we go past the end of the array,
        offset1 -= N<<16;  // wrap around with same offset as overshoot.
    offset2 += step2;
    if ((offset2>>16) >= N) // If we go past the end of the array,
        offset2 -= N<<16;  // wrap around with same offset as overshoot.
    int sample = 0;
    sample += wavetable[offset1>>16]; // get sample for tone #1
    sample += wavetable[offset2>>16]; // get sample for tone #1
    sample = sample / 32 + 2048;      // adjust for the DAC range
    if (sample > 4095) sample = 4095; // clip
    else if (sample < 0) sample = 0;  // clip
    DAC->DHR12R1 = sample;            // trigger DAC here.
}
Playing many notes

- Use an array of steps for all possible notes.
- Use an array of offsets for all possible notes.
- Still divide the cumulative amplitude by 32, and still clip it.
  - Probably will not have three or more notes coinciding with a high amplitude, but watch out to make sure.
- When a note starts, set its offset to zero and mix it into the sample on each ISR.
- When a note ends, mark it so it is not mixed in with the sample.
ISR pseudocode

char pressed[90] = { 0 }; // which piano keys are pressed?
int offset[90] = { 0 };
int step[90] = { 0.16352 * (1<<16), // Low ‘C’ (16.352 Hz)

    ... 7902.133 * (1<<16), // High ‘B’ (7902.133 Hz)
};

void ISR(void) {
    int x;
    int sample = 0;
    for(x=0; x<90; x++) {
        if (pressed[x]) {
            offset[x] += step[x];
            if (offset[x] >= N<<16)
                offset[x] -= N<<16;
            sample += wavetable[offset[x]];
        }
    }
    sample = sample / 32 + 2048; // adjust for the DAC range
    if (sample > 4095) sample = 4095; // clip
    else if (sample < 0) sample = 0; // clip
    DAC->DHR12R1 = sample;
    // trigger DAC here.
}
Small suggestion

- When producing multiple notes, the time taken by the ISR will be different depending on:
  - how many notes are generated
  - other calculations you might want to add
- Since the DAC trigger happens at the end of the ISR, it could be slightly delayed sometimes which will lead to an inconsistent sample speed.
- Put the DAC trigger at the very beginning of the ISR, and store the result for the NEXT trigger at the end of the ISR.
ISR pseudocode (trigger at start)

```c
#include <stdio.h>
#include <stdlib.h>

#define N 8192
#define WAVETABLE_SIZE 100

char pressed[90] = { 0 }; // which piano keys are pressed?
int offset[90] = { 0 }, step[90] = { 0 }; // Low 'C' (16.352 Hz)
    7902.133 * (1<<16), // High 'B' (7902.133 Hz)
}

void ISR(void) {
    int x; int sample = 0;
        // You should trigger the DAC here this time.
    for(x=0; x<90; x++) {
        if (pressed[x]) {
            offset[x] += step[x];
            if (offset[x] >= N<<16)
                offset[x] -= N<<16;
            sample += wavetable[offset[x]];
        }
    }
    sample = sample / 32 + 2048;      // adjust for the DAC range
    if (sample > 4095) sample = 4095; // clip
    else if (sample < 0) sample = 0;  // clip
    DAC->DHR12R1 = sample; // only store to the DAC here this time.
}
```
Multiple voices

- If you can store a wave table in 1K samples (2K bytes), you will have room for multiple wave tables.
  - Not all wave tables need to have a sine wave:
    - Make pointier wave shapes for sharper sounds.
      - Like a sawtooth wave: \( \text{wavetable}[x] = \frac{x \times 65535.0}{N} - 32768 \)
    - Add small ‘harmonics’ to a sine wave for a richer sound.
      - e.g., Let each sample be \( \sin(x) + \frac{\sin(2x)}{4} + \frac{\sin(3x)}{8} \)
    - Record your own sound and capture it with the ADC.
  - This lets you assign multiple ‘voices’ per note if you want to.
  - By adding and averaging the samples of multiple wave tables, you can produce many types of sounds. (e.g., what is the average of a sine wave and a square wave if they have the same frequency? How about a sine wave and a sawtooth wave?)
  - Maybe you want to have some wave tables with more samples than other wave tables?
ADSR

- Your textbook describes an ADSR (attack / decay / sustain / release) model for a synthesizer. (pages 521 – 523)
  - If you keep track of the time at which you ‘press’ a note, you can modulate its contribution to a combined sample with its ADSR curve.

- Also easy to create vibrato and echo effects this way.
Pitch Bending

- One of the most interesting things you can do with sound synthesis is pitch ‘bending’.
  - Dynamically change the step size of each note to correspond to a different frequency.
  - By changing each frequency to that of the frequency below it, you can shift your sound output by a half step.
  - By multiplying each step size by $1 + \Delta$, you can shift your sound output by a variable offset.
    - Determine that multiplier by an ADC sample of a sliding potentiometer will let you have trombone-like glissando effect.