Programming for Embedded Systems Lecture 5: Building a Basic Sound Synthesizer

Bernhard Firner

February 16, 2015

RUTGERS

What's Next?

- We've learned
  - Digital output
  - Timers
  - Pulse width modulation
- The final project is a music player, so let's talk about music

DAC and Music

- With PWM we can simulate analog output
- This can be used to reconstruct sounds, such as speech or music
- We will eventually play prerecorded samples, but we can also synthesize our own sounds

Simple Sound Synthesizer

- The next project will be a simple sound synthesizer
- What do we need to know?
  - Sound synthesis
  - Basic button input (for control)

What is Sound?

- The human ear is basically a frequency detector
- Sound is pressure waves propagating through air
- Fluid in our ear is vibrated by this pressure
  - This amplifies weak pressure waves that are occurring at even frequencies
- Different nerves vibrate at different frequencies – when these nerves vibrate they tell us that we've heard a certain sound

Sound Synthesis

- Sound is just a pressure wave repeating at some frequency
- The simplest synthetic sound is just a repeated pulse
- The pulse just needs to repeat at a given frequency to produce a sound
- Real sounds are complex and are made of many harmonics
    - Different frequencies mixed together

Single Pulses Aren't Sound

- A single pulse isn't a sound
  - It is just pressure in our ear
  - A single wave isn't amplified by our biology
- The pulse needs to repeat for a human to detect a sound
- The human hearing range is roughly 20 to 20,000Hz
  - Varies between individuals and with age

Sounds Have Names

- Because we do so much with sounds we've named certain frequencies
- The modern musical scale has notes A, B, C, D, E, F, and G
- The notes repeat; there is a G below A and an A after G
- There are also "sharps" and "flats" that increase or decrease a note's frequency
- The 7 notes plus 5 sharps or flats are grouped into sets of 12 called "octaves"

More About Sound Names

- Since the note letter repeat we need to distinguish between a lower “A” and a higher “A” so we give each one a number
- A4 is currently 440Hz (concert pitch)
- Notes are related logarithmically; A3 is half the frequency of A4 while A5 is twice A4’s frequency
- The shift from one note to its higher or lower counterpart is an octave shift

Making a Simple Tone

- We've already made a tone
  - The 1/4 width pulse at 16KHz from last time
- If we just put our output through speakers we'll hear a tone
- If we create a pulse at 440Hz we'll be playing the note A4

A 440Hz Tone

```
int steps = 0;
//Divide the sample frequency by the note frequency
//This is the number of samples between pulses
int a4_interval = 16000 / 440;
while (1) {
    while (update_lock);
    ++step;

    //Create a pulse here -- the interval determines the note
    if (a4_interval == step) {
        buffer = 250;
    }
    else {
        buffer = 0;
    }

    //Make sure we block until the interrupt reads the data
    update_lock = 1;
}
```

Does the Pulse Width Matter?

- The human ear is sensitive to amplitude and frequency
- Changing from a 10% width pulse to a 15% width pulse won't change the frequency, but the total power increases
    - This was what you saw when you used applied the reconstruction filter to the single pulse
- A pulse itself is a very digital sound though
    - People usually find a more "full" output, such as a sine wave, to be more pleasing

Sine Wave Output

- Sine waves sound less abrupt than pulses
- However, an unrectified sine wave output with PWM will have an annoying high frequency component
- Unfortunately, rectifying the signal significantly lowers the output power
- To hear sine wave output from the MSP430 we'll need to use a speaker with an amplifier or add an amplifier ourselves

Choosing a Tone

- Last lab we generated a pulse, a sawtooth, and a sine wave at a desired frequency.
- Tone generation is the same task, we just need to know what the tones are

Multiple Sequential Tones

- A song or scale has multiple tones, so our code changes a bit

1. One variable keeps track of the current note
2. Another variable keeps track of the phase of that note
   - e.g. to play a pulse every 36 sample this counter goes from 0 to 35
   - The period of the note is determined by the current note
3. Another variable keeps track of how long the note is played
   - If a note should last for 1/2 second, with 16,000 samples per second this note lasts for 8000 samples
   - When this counter rolls over you increment the note counter

Generating Multiple Sine Waves

- If we are synthesizing a sine wave there is one more step
- We can't store a sampled sine wave for every single note
    - Not enough memory
- Instead we'll store one large sampled sine wave in an array
- We'll subsample that array to generate other sine waves

Sine Wave Example

- For example, let's save we've stored a 200 sample sine wave

```
unsigned int sine_samples[] = {...};
```

- A4 is 440Hz, and we want to play a sine wave
  - Basically we will compress a 200 sample sine wave into 36 samples
  - We'll do this by skipping some values

Sine Wave Continued

- We are playing 16KHz samples
- The period of A4 is $16,000/440 = 36.\overline{36}$
- $200/36.\overline{36} \approx 2.84$
- So the index in the sine wave should increase by 2.84 every sample

Song Resources on Sakai

- songs.h (on sakai) has several helpful definition
- Notes from C3 to B5
- The notes in the A major scale in an array
- The notes for twinkle twinkle little star with their durations
- The notes for an old irish tune with their durations

Digital Input

- Music players have buttons!
- We'll cover more about input next week, but today we'll do some basics
- Here are the registers we'll use:
  - P1DIR
  - P1IN
  - P1REN
  - P1OUT

Pin Direction

- Previously we've set P1DIR to turn a pin to output mode
- Clearing a bit in P1DIR sets a port 1 pin to input mode
- Reading that bit in P1IN gives either a 0 or 1
  - 0 for no input
  - 1 for input

The P1.3 Button

- There is a button on pin 1.3
- It is attached to ground, so pushing the button sets it low

```
//Set up the P1.3 button as input
P1DIR &= ~BIT3;
```

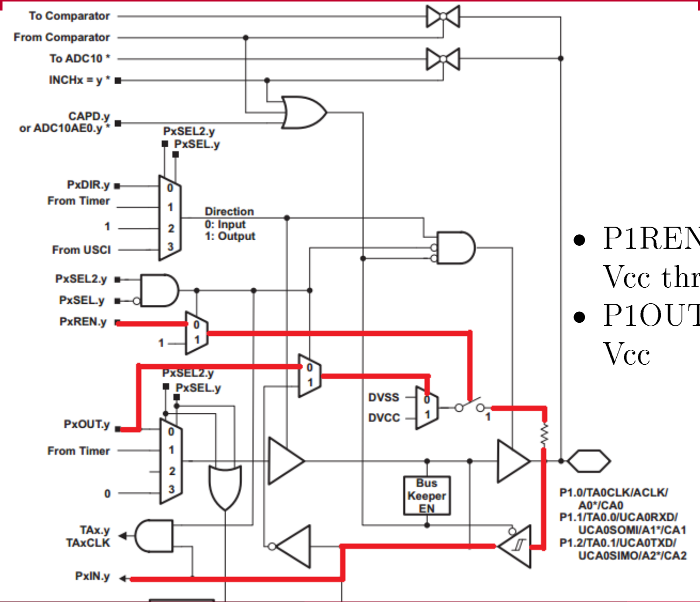
- We check the input state by ANDing P1IN

```
if (P1IN & BIT3) {
//Do something when the pin is high
}
else {
//Do something when the pin is low
}
```

Getting P1.3 High

- Pin 1.3 is pulled low when we push the button, but how does it get high?
- We need to pull it up to Vcc with a *pullup-resistor*
- Pullup and pulldown resistors are enabled with the P1REN register
- Pullup or pulldown is set by P1OUT:
    - P1OUT is 1: pullup
    - P1OUT is 0: pulldown

Port 1 Schematic



- P1REN connects Vss or Vcc through a resistor
- P1OUT chooses Vcc or Vcc

Code Example

```
//Set up the P1.3 button as input
P1DIR &= ~BIT3;
//Enable pullup resistor
P1REN |= BIT3;
P1OUT |= BIT3;
//Set up P1.0 for debugging
P1DIR |= BIT0;
while (1) {
    //Turn on LED1 if the input is high,
    //turn it off otherwise
    if (P1IN & BIT3) {
    P1OUT |= BIT0;
    }
    else {
        P1OUT &= ~BIT0;
    }
}
```