ECE381 – Lab 3 – PWMs for Brightness and Sound

Introduction:

Pulse-Width Modulation (PWM) is a common method for generating "analog" voltages using digital signals. PWMs use repeated square waves with varying duty cycles to control the average voltage. Most microcontrollers contain built-in hardware to generate PWMs, typically involving a clocked counter and digital comparator. We will use PWMs to generate light & sound effects to create a knockoff of the old Milton Bradley™ Simon handheld game. Four push-buttons corresponding to each of the four colored LEDs are used as input to attempt to play back a "randomly" generated LED pattern, where the LEDs get progressively dimmer as the game progresses.

Objectives:

- Learn how PWMs create "analog" voltages by using different duty cycles to create different average voltages
- Learn about PWMs and how they can be used to generate waveforms with desired frequencies & duty cycles
- Learn about pseudo-randomness on microcontrollers and how it can be challenging to generate sequences that appear to be random

Background:

PWMs:

In this lab, we are going to learn about Pulse-Width Modulation (PWM), and how PWMs are used to control many different things with a digital signal that approximates an analog signal (in fact, the Arduino's *analogWrite()* actually outputs a PWM not a real analog value!). PWMs use the *width* of a pulse train to *modulate* the behavior of what it drives. Changing the width of a pulse train changes the average DC voltage, which often has a similar effect as setting a Digital-to-Analog Converter (DAC) voltage. Most microcontrollers will have some kind of built-in hardware to handle PWMs, and this built in hardware is typically coupled to the timer hardware, as PWM cycle frequency typically requires the same level of control of timing and repetition that timer hardware provides.

How they work will go somewhat like this. Suppose we have the timer clock tick rate set to 10 kHz. Since we know the timer ticks 10,000 times-per-second (10 kHz), we would know that when the *COUNTER* register equals 9999, one whole second would have elapsed (and we could reset it to 0 to restart the process if we set the *PERIOD* Register to the same 9999). Now, suppose that if the *COUNTER* register was less than 5000, we left the LED on, then when it turned 5001,we turned it off.

Our LED would be on for half a second then off for half a second. This would be a PWM square wave with a 1 Hz frequency and 50% duty cycle! The comparison value (5000 in this case) can actually be stored in a 2nd register called the *COMPARE* Register.

This is all well and good, except our LED noticeably turns off for half a second! If what we really want to do is reduce the brightness, we need to change how often the on/off cycle repeats (frequency) and correspondingly the duty cycle (on/off ratio). Increasing the frequency of the cycle makes the repetition occur more often, and just like increasing the frequency of say, your computer monitor, the higher the frequency of repetition, the "smoother" the LED on/off transitions will be. Make it high enough, and it will surpass our human eye's ability to determine that it is off, thereby making the LED appear to be on at all times, even though it's actually turning off lots of times per second! (This phenomenon is called "Persistence of Vision"). Now, changing the duty cycle (ratio of time LED is on vs. off) will alter the brightness. How? A digital I/O pin can only produce GND or 5V you say. Well, by adjusting the duty cycle, we adjust the *average* amount of energy input to the LED, and thus, ultimately, the average number of photons the LED produces. How much goes like this:

$$v_{avg} = DC*v_{max} + (1-DC)*v_{min}$$

where *DC* is the duty cycle, v_{max} is V_{DD}, and v_{min} is GND or 0V for our digital GPIO pins. So by reducing the duty cycle, we reduce the average voltage and make the LED dimmer, and by increasing the duty cycle, we increase the average voltage and make it brighter.

Suppose rather than waiting until 9999 to reset the *COUNTER* register back to 0, we only wait until the *COUNTER* register reaches 99 before resetting it to 0. We do this by setting the *PERIOD* register to 99. We now would have changed the *frequency* from 1 Hz (10000/10000) to (10000/100 = 100 Hz). Unfortunately, if we didn't change our threshold from 5000, our LED would never turn off (as it would *always* be less than 5000), so we would need to adjust that too. Suppose you set the *COMPARE* register to 75 instead. Now since *COMPARE*, at 75, will be larger than the *COUNTER* register for the first 75 out of 100 total ticks, the LED will be on for 75 out of 100 ticks (or a 75% duty cycle or another way of saying it is on 3 times as long as off and thus brighter). If instead our *COMPARE* register was 25, now the LED would only be on or the first 25 out of 100 ticks (it would be off for 3 times as long as it is on, and thus dimmer). Pretty cool, eh? In general, the combination of input clock, Prescaler, *PERIOD*, and *COMPARE* give control to the rate and resolution of the output PWM! Using the previous example at 100 Hz repetition, we would only be able to generate whole number changes in the duty cycle, as there are only 100 ticks per cycle. If we go back to 1 Hz repetition as above, we could actually get duty cycle accuracy down to 0.01% (if instead of 5000, we went with 5001, our output duty cycle would be 50.01%). If we wanted both higher repetition, and higher duty cycle, we would have to increase our input clock to compensate (a 1MHz input clock with PERIOD = 9999 would give us 100 Hz repetition with 0.01% DC accuracy).

Finally, is this really going to be a linear brightness? NO! The number of photons an LED produces is non-linear with Voltage/Temperature/Wavelength/etc., so you won't notice it getting that much dimmer until you likely get in the single digits for duty cycle. Do we care for this lab? NO! We're here to learn about PWMs, not sell LED dimmers. There's a side note here about how LEDs operate that illustrate that PWMs are not "true" analog voltages. If the actual analog voltage across the LED was only 50 mV(the rough equivalent of a 1% duty cycle with a 5V supply), the LED would be off since the LED voltage would be less than the forward voltage required to bias the diode. However, since a PWM isn't actually producing 50mV, but producing 5V, just for a brief period of time, the LED *will* actually be on. It will just be noticeably dim (as it's only on 1% of the time) but it will be on, compared to a DAC that directly outputs 50mV!

PWMs are not just for LEDs! We can even use them to generate music, like many of the old retro video games. Pulsing a speaker on/off can generate sounds at the frequencies of the PWM! (to be fair, a square wave in time is a sinc in frequency, so it's not a pure tone. The dominant harmonic is at the PWM frequency, but there are side harmonics present too!). We will take advantage of this to generate the audio for our Simon clone in basically the same way Milton Bradley did all those years ago, using a 1-bit PWM speaker driver (see Figure 1 below).

Random Numbers on Microcontrollers

Sometimes you want to be able to generate a "random" number. Since all processors are deterministic (they can only execute a pre-defined sequence of opcodes), true randomness is actually quite difficult to achieve on any CPU. Most computers skirt the issue by having some pool of entropy maintained by the operating system, whose source is a combination of non-deterministic things it can access (various component temperatures, disk access times/rates/block IDs/etc, mouse movements, keyboard keys, other environmental noise, ...). The OS uses this pool to "seed" a sequence generator whose output produces what appears to be a sequence of random bytes. As an example, on UNIX systems, the /dev/random device contains the output of this sequence (Windows uses the CryptoAPI). Note, this sequence only *appears* random, but the generator is actually using an algorithm to create the sequence, and thus, if the seed is known, the entire sequence can be generated deterministically. Such types of generators are called *Pseudo-Random* as the output appears random, but really isn't.

In the C standard library, there is a `rand()` function that is in stdlib.h. It functions similarly to the OS pseudo-random generator, in that it uses a seed number (0 by default, though it can be set by the `srand()` function) to generate the sequence, and each successive call of the `rand()` function produces the next output of the sequence. Unfortunately, unless there is a good source of non-determinism available to seed it, calling the `rand()` function will produce the exact same sequence of numbers. Such is typically the case in microcontrollers with no OS available to pool non-deterministic sources into a source of entropy. Thus, alternative methods are needed, or some partially non-deterministic way of seeding it (ie. reading a floating ADC pin, measuring the time for a button press, repeatedly calling `rand()` in a polling loop, sampling a temperature value, etc.) has to be used. (Side Note: The original Simon just incremented a variable from 1 to 4 in the polling loop, it didn't even use rand()!)

One way to skirt the issue would be to do it in hardware. A common way to generate a pseudo-random sequence is through a device called a *Linear-Feedback Shift Register (LFSR)*. A LFSR is a shift register whose next input bit is a function of some combination, typically using XOR gates, of current bits in the register. Which bits are used (they are sometimes called the *taps* since you are "tapping" that bit's output in the register) and how they are XORed are often expressed as a polynomial. See example below (From Wikipedia article on LFSRs):

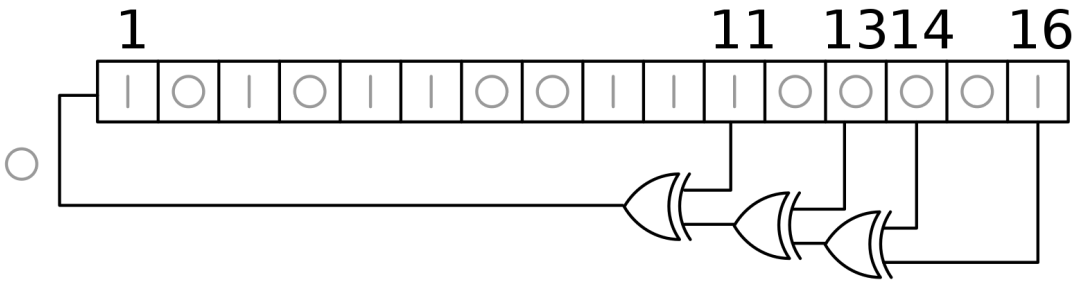


Figure 1: Linear Feedback Shift Register for 16-bit number

The polynomial for this example is:
$x^{16} + x^{14} + x^{13} + x^{11} + 1$
since the 16th, 14th, 13th, 11th, and bits are tapped to produce the next input, and 1 to indicate the input is the 1st bit. The current *state* of the generator is 0xACE1. On the next clock cycle, the value, in binary, would be 0101011001110000 or, in hex 0x5670. Therefore, reading the register at different points in time will produce, what appears to be, the "random" sequence of values. Seeding an LFSR is merely setting the value of the initial state. The input bit to position 1 can also be seen as a "random" bitstream when looking on the outside. Just note, though, that if you know the current state and polynomial, the output sequence is entirely deterministic, meaning I could tell you which number comes next (or in 10 or 17 or 100 iterations, it doesn't matter), hence the name *pseudo-random* as it only appears random if all you see is the output and the rest is concealed. There are other downsides, too. Eventually, based on the size of the register, the sequence will repeat, which is called the *period* of the generator. The bigger the register in bits, more generally, the longer the output "random" sequence before this occurs, to a maximum of 2^{N-1} values, where N is the register size in bits. The period is also based on the polynomial, and not all polynomials are equal. Some will repeat the sequence before all 2^{N-1} values have been output. Thus, there are some predefined polynomials that are commonly used to ensure the max. period is reached. LFSRs can also be implemented relatively easily in software too.

On the PSoC, there is a module that can generate a PRS in hardware. It takes as input a clock, and as output produces the bitstream. If you go to configure it, you can punch in the resolution, in bits, of the sequence, and it will produce the optimal polynomial for that size by default. It also has an API that

allows you to _Read() the current value (or state) of the generator, to _WriteSeed() if you want to reseed it, or even to _WritePolynomial() if you want to change it from the pre-generated one. Pretty nifty, though you would still have to make sure that you don't call the _Read() function at the exact same point in time in your code after calling _Start()!

Regardless of how you generate the sequence (hardware/software), if you need to generate numbers within a specific range, you use modular arithmetic! The general form is:

```
int number_in_range = (rand() % (max - min + 1)) + min
```

where max & min are the largest and smallest values of the range, inclusive. Let's use a 6 sided die for example. The minimum value of the range, min, would be 1, while the maximum value of the range, max, would be 6. 6 - 1 + 1 = 6, so we would be taking rand() % 6. This returns a value between 0 and 5, since % means mod, which is the remainder of division. Since dividing by 6 results in 6 possible remainders (0 for evenly dividing, 1-5 for not), then for a given uniformly distributed random number (which is just a fancy way of saying all numbers are equally likely from calling rand()), any of those remainders is equally likely, so that gives us a random number between 0 and 5! Since a die doesn't have a zero side, we add the minimum value (1) back to give us a range from 1 to 6, where each of those values now has a roughly 1/6 chance of happening, the same as a physical die.

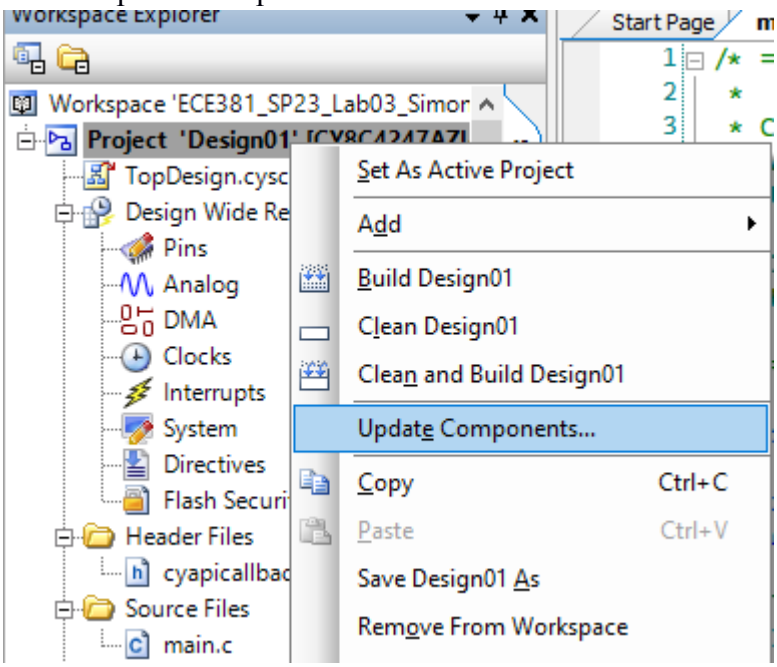
Simon Game

We will use what we just learned to make a clone of the classic Simon game. We will use the PRS module to generate a random sequence of four colors (Red, Blue, Orange, Green). For each color in the sequence, we use a PWM to play a predefined note for that color at the correct frequency (50% duty cycle is fine for this). To make the game more rewarding, we will also use a PWM to slowly decrease the brightness of the LEDs as the sequence gets longer. To do so, will decrease the PWM duty cycle by 5% for each color in the sequence (starting at 100%), until the minimum brightness of 5% during round 20 (the final round).

Configuration:

- Download the template project from the lab directory.

- If you have missing components, right-click on the project name in Workspace pane and select “Update Components”. Hit next a few times and it should work.



- Fill in the empty functions, as well as any code wherever you see XXX in the comments

Hardware:

- Build the circuit shown in Figure 2. You may use the speaker included in your kit or the one on the Digi-Designer at your lab station. (NOTE: The resistor values don’t necessarily have to match, smaller resistors mean more current, which means louder sound!)
- The long lead of the LED is the anode, the short lead is the cathode. The resistors for the LEDs are current limiting resistors, and may not be strictly necessary. They are there to limit the brightness of the LEDs.

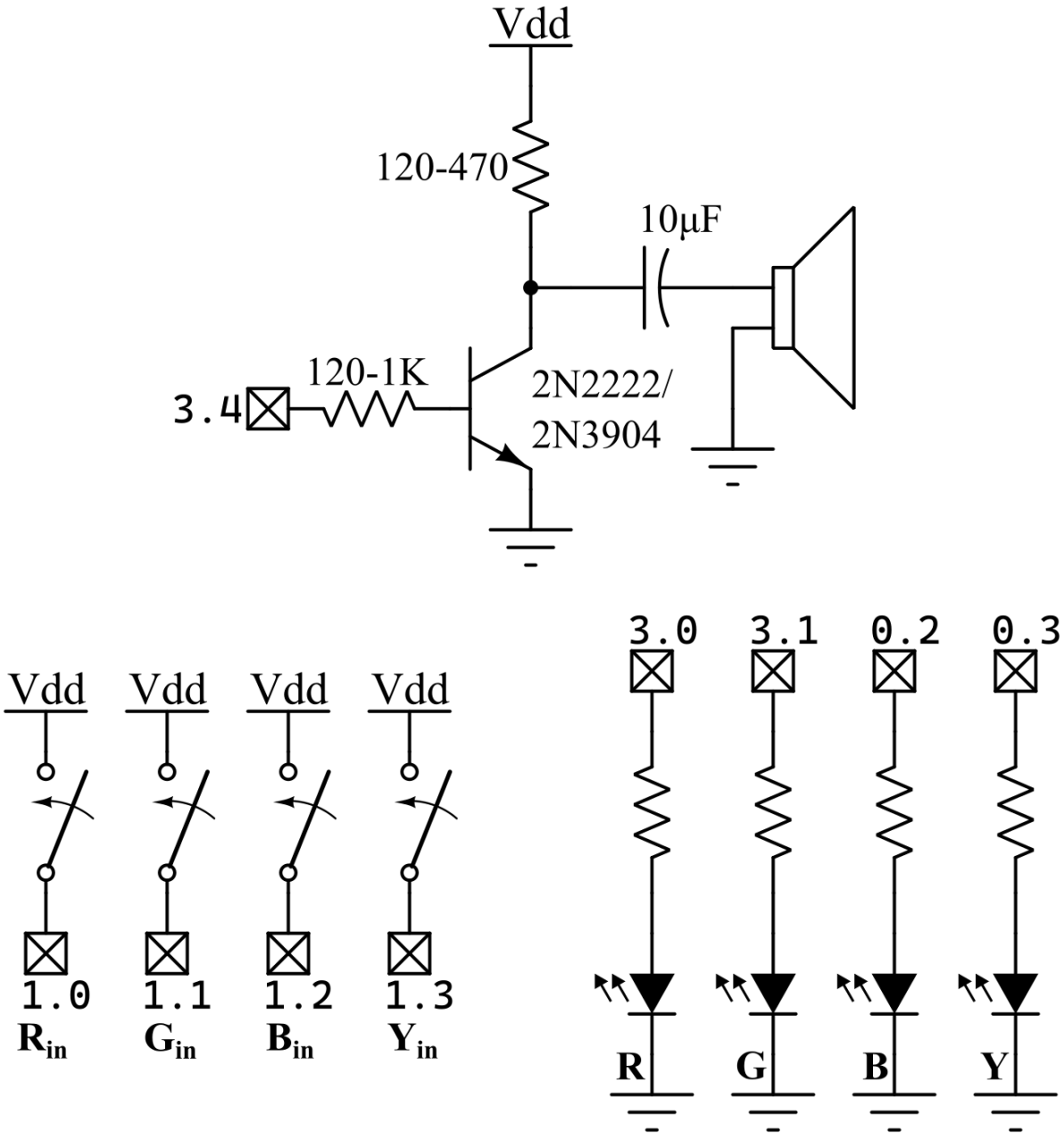


Figure 2: Lab03 Schematic

Software Requirements

- On startup, wait for the user to press any of the 4 buttons to begin the game
    - Make sure to start the PRS before this so that it will generate good random sequences
- Once pressed, play a 600 Hz tone and blink all LEDs 3 times, spaced 500ms apart
    - NOTE: To minimize internal resources, only a single PWM is utilized for all 4 LEDs. A demultiplexer, controlled by the LED_SEL register, is used to choose which of the four

colors is output. Calling `LED_SEL_Write(num)`, where *num* is the LED you wish to select, chooses the color. You technically can't actually display all four LEDs at once, since there is only one PWM and demux output. But, if you cycle through all four de-mux outputs quickly enough, it will appear as if they are all on at the same time!

- Use the PRS to randomly generate a sequence of 20 LED colors and store them in an array.
- Illuminate the LED and play the corresponding tone on the speaker using the PWM:
    - Green: 415 Hz
    - Red: 310 Hz
    - Orange: 252 Hz (Simon originally had yellow, but this is close enough!)
    - Blue: 209 Hz
- The playback duration for "Easy" Simon is 0.42 seconds ("Medium" is 0.32 seconds, "Hard" is 0.22 seconds but we will only do Easy here to keep things simple)
- Playback each LED in the sequence up to the current round number
- Wait up to 3s for the user to press one of the four buttons
    - IF the correct button is guessed:
        - Keep looping to detect the next button press
        - If the sequence for the round is entered correctly, wait 0.8s, then playback the next round by adding one more color to the sequence
    - IF 3s elapses OR the wrong button is pressed:
        - Turn on all LEDs
        - Play back the losing tone, which is 42 Hz, for 1.5 seconds
        - Turn off all LEDs
        - I'll allow you to just wait here until Reset is pressed to start over, or you can go back to the original loop of waiting for any of the four presses to start again
    - To wait for 3 seconds, you could use a Timer interrupt or you could just add a predefined delay in a polling loop that accumulates to 3s