A detailed inspection of the wavy patterns presented by the OptoGlitch

Patrick Pedersen

---

0. Expectations from the reader

For obvious reasons, I cannot cover all subjects addressed throughout this paper. The reader is expected to have a pre-established understanding in the following subjects:

- Photoresistors
- Pulse Width Modulation
- Oscilloscopes, particularly their trigger function

1. Introduction

It has been more than half a year ago since I have published the code for the OptoGlitch onto GitHub. The very idea and concept of the OptoGlitch however, likely arose perhaps even more than a year ago. When I first had the idea to parse Image data analogously through an optocoupler circuit, I had no expectations for what the results would look like. What can be said is that I was anticipating VHS cassette like glitches. When I had finally built the very first prototype for the OptoGlitch, I was overwhelmingly surprised with the results that it delivered. Although vastly different from VHS glitches, I was still happy with the unique set of “wavy” artifacts that arose. At the time of writing, I would even say that I much prefer that those artifacts to those of a glitchy VHS cassette, as that makes the OptoGlitch a unique device itself.

The most notable glitch artifacts that the OptoGlitch produces are:

- “Wave” like patterns
- Color shifts
- Shifts in brightness.

While the wave patterns are to be discussed in much further detail throughout this document, the exact cause of the color shifts is yet to be discovered.

Nonetheless, the shift in brightness can obviously be attributed to a change in the surrounding lighting throughout the parsing process.

Take the following excerpt from an animated gif of the Pokemon Emerald “Corrupted save file” screen, which has been parsed through the OptoGlitch:

Original

Fig. 1.1

Parsed

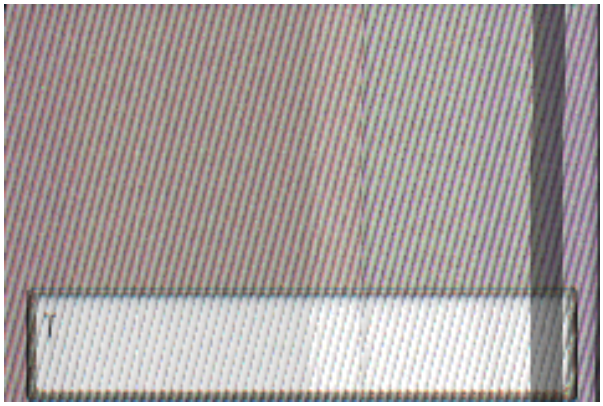


Fig. 1.2

Across the entire image we can see a repetitive pattern of diagonal stripes. Were the resolution of the source image greater, than those would form wave like patterns. To the right of the parsed image we can also see changes in brightness and colour. Most notably, along the very right we see a dark stripe, implying that the lighting has become dimmer at that point in time. Perhaps it was my monitor, which during the parsing process was positioned just next to the device, that had temporarily entered its automatic “sleep mode”, only to be awakened again just moments later.

In the proceeding glitch piece I have parsed a high resolution photo that I had taken at the mountains of Val Thorens, France. Due to the images high resolution, the wavy pattern becomes particularly noticeable here:

Original

Fig. 2.1

Parsed

Fig. 2.2

Another sample that demonstrates the artifacts of OptoGlitch excellently, is an excerpt that I took from a vintage US Department Of Defense educational film from 1963. Similar to the “Corrupted save file” screen, I have also parsed a GIF through the OptoCoupler here. The following frame of the GIF turned out most interesting, as it captured a point in time where the lighting in the room has changed in a way that caused the background to shift its color to red, as well as inducing a significant amplification in the wavy pattern:

Original

Fig. 3.1

Parsed

Fig. 3.2

The next frame:

Original

Fig. 3.3

Parsed

Fig. 3.4

2. Hypothesis

For a prolonged time I could only speculate what is causing the wave like patterns. Speculations ranged from AC powered lighting being the cause, which could be quickly disproven by the fact that the artifacts were still present even under no artificial lighting (besides the LED), to IR interference, all the way to more plausible speculations regarding the pulse width modulation (PWM) used by the Arduino to drive the LED. As it turns out, the very latter speculation turned out to be correct, as will be discussed later.

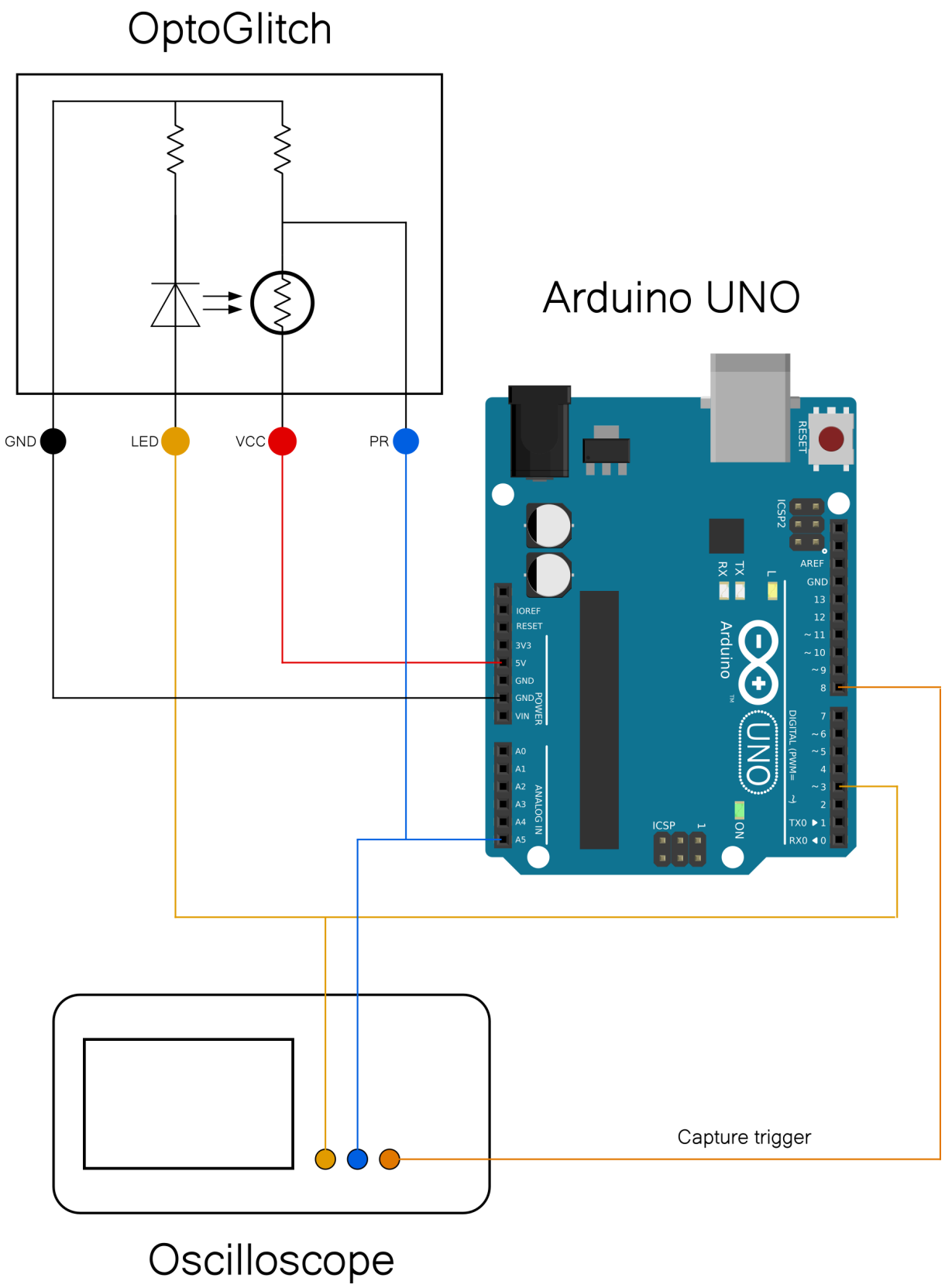
Saving up for an oscilloscope gave me the possibility to analyse the electrical signals that the Arduino was emitting and receiving, and allowed me to discover the secrets behind the unique wavy glitch artifacts that we get to see in parsed images.

3. The setup

The setup involved an Arduino UNO connected to the OptoGlitch, along with an oscilloscope probing the LED and photoresistor voltages, as well as a trigger pin on the Arduino that would output a short signal on whenever the device captured the photoresistor input. This signal acted as a trigger for the oscilloscope which allowed me to see at which exact point in time the Arduino captured the photoresistor input.

Any software calibration in the OptoGlitch software has been disabled for this procedure, as any alteration of the received values within the software could be deduced from the results received with no calibration enabled.

During my experiment, I had assigned the 8th pin as the trigger pin, however, any other available GPIO pin would work as well.



Oscilloscope GND intentionally left out

Fig. 4

In practice, the setup looked a little messier:

Fig. 5

Throughout the parsing process, one could observe similar signals on the oscilloscope, to those:

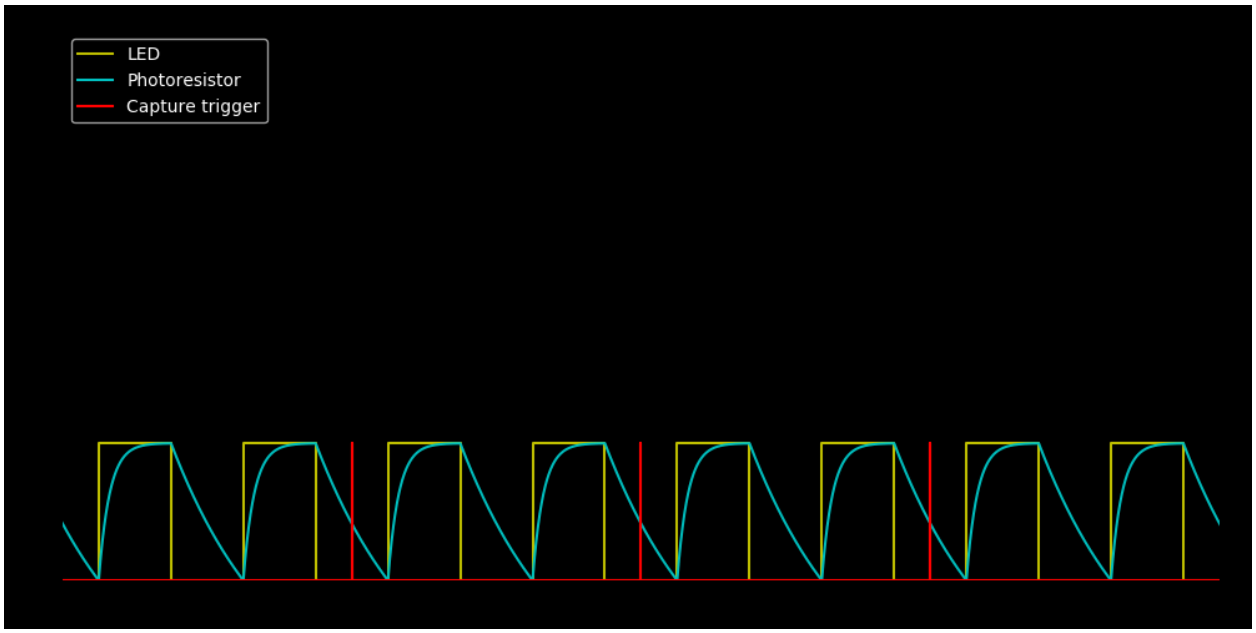


Fig. 6

Since the exact voltages of the signals are irrelevant for an understanding of the glitch artifacts, the photoresistor signal will be displayed above the LED output signal for the sake of clarity:

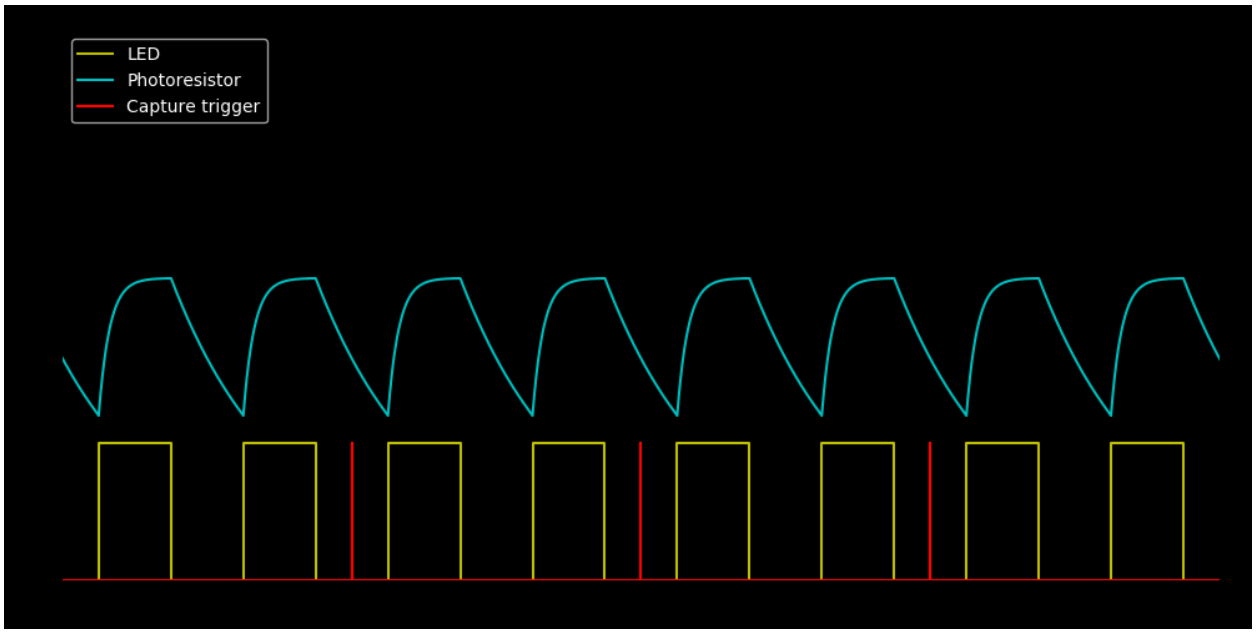


Fig. 7

For comparison, a picture of the oscilloscope during an image parse:

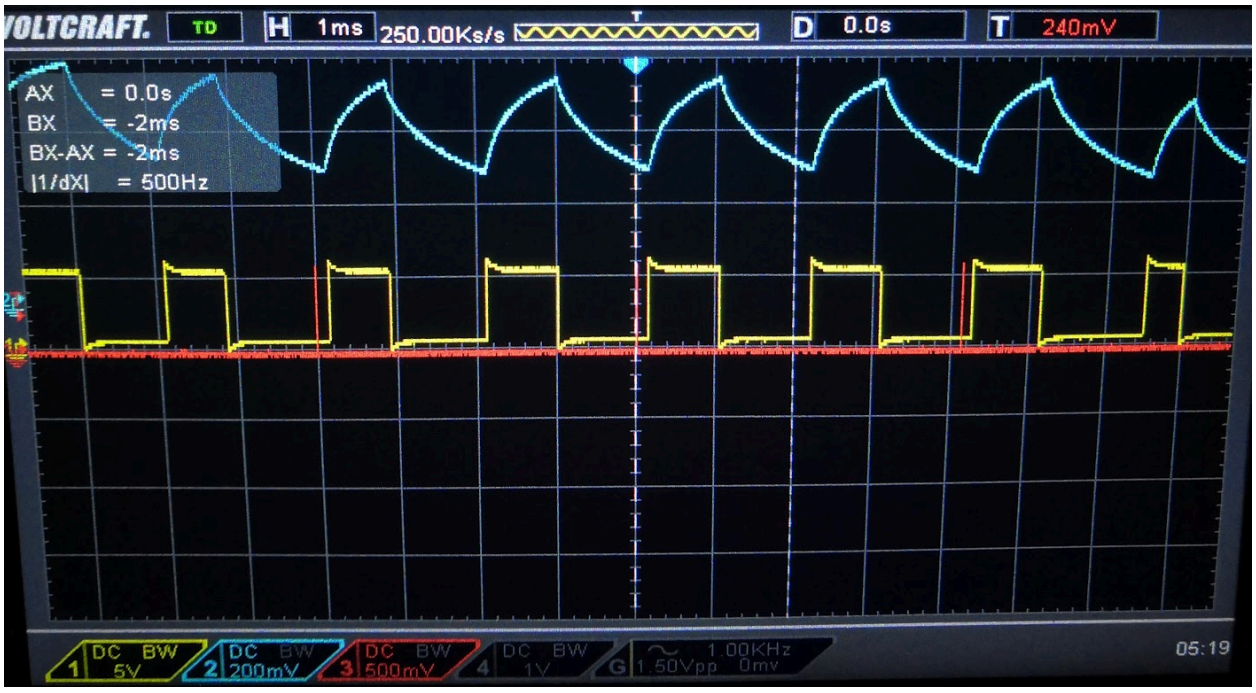


Fig. 8

Due to the fact that the oscilloscope trigger is set to the capture signal (red/orange), we may choose either of the three displayed “peaks” as reference points on the photoresistor signal (cyan) that tell us the exact voltage, and thus value, that the photoresistor delivered when it was captured by the Arduino.

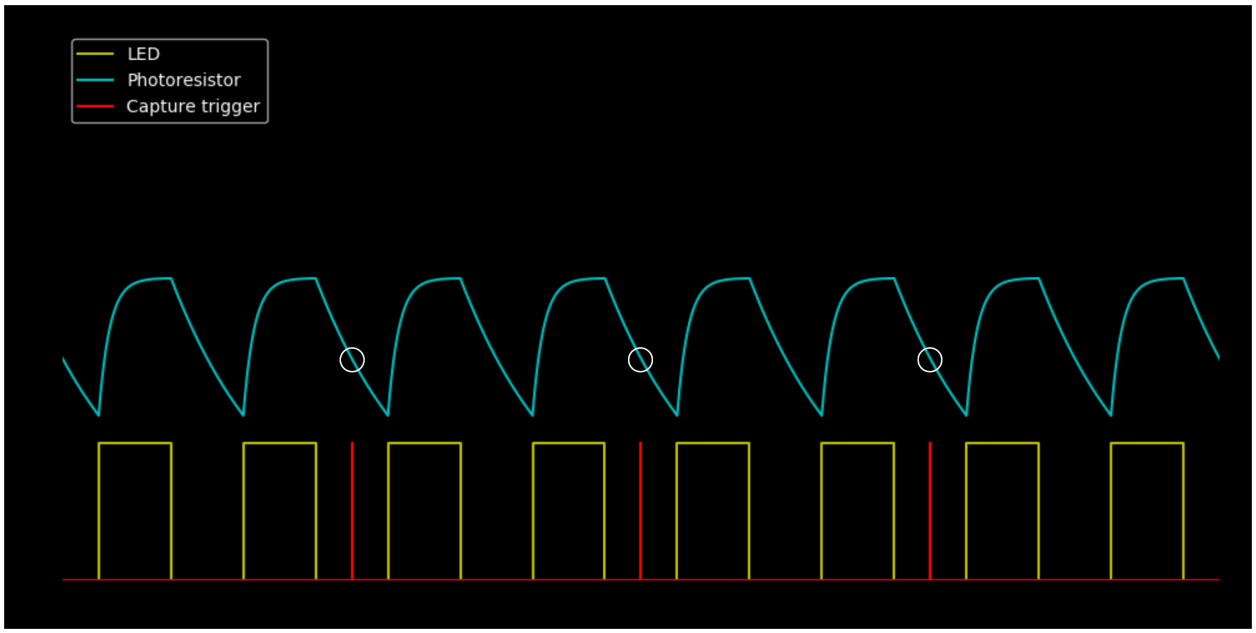


Fig. 9

Readers that are already familiar with the Arduino platform may think of the white points on the photoresistor signal as the values returned by the `analogRead(...)` function. Further, since a total of three peaks are visible, with correct timing, those *may* make up an entire pixel, as each RGB value is parsed through the optocoupler separately. However, I place emphasis on the *“may”*, as incorrect timing could result in an offset of one or two color values, meaning we would only see two pixels being partially parsed :

Correctly timed

Peak 1 (left)	Peak 2 (center)	Peak 3 (right)
R	G	B

Incorrectly timed (Offset by one color value)

Peak 1 (left)	Peak 2 (center)	Peak 3 (right)
G	B	R
	End of previous pixel	Start of next pixel

Incorrectly timed (Offset by two color values)

Peak 1 (left)	Peak 2 (center)	Peak 3 (right)
B	R	G
End of previous pixel	Start of next pixel	

The only reliable way to know whether all three peaks are readings of the same pixel would be to implement another trigger signal that would peak at the start of every pixel, rather than every RGB value.

Finally, it shall also be said that in the simplified diagram above, the LED signal has been simplified to a square wave with a 50% duty cycle. Under ideal circumstances, a wave as such would translate into grey colour, since all three color values would share the same value. In reality, where parsed images typically consist of more than just white, grey and black pixels, the duty cycle of the LED signal can drastically change after every photoresistor capture . In fact, in Figure 8 we can even observe a change in duty cycle after the latest (most right) capture.

4. Understanding the wavy pattern

To understand what causes the wavy artifacts in our parsed images, we must dive into the Arduino's `analogWrite` function, and at the very least, establish a superficial understanding of the AVR PWM timers.

Unlike implied by the name, the `analogWrite` function does not generate a real analog signal. Instead, it generates a pseudo analog signal by generating a PWM signal. For this PWM signal to remain constant at all times, it must run independently, or *asynchronously*, of the main program. Were this not the case, then the duty cycle of the signal could vary depending on the code that is being executed, and the LED would flicker in all sorts of random ways. For the PWM signal to run independently of the main program, the `analogWrite` function makes use of the 8-bit AVR PWM timers. I could go into allot more detail about those timers and highly encourage readers to read *"MICROCONTROLLER TUTORIAL II - TIMERS" by the Indian Institute of Technology Kanpur [1]*, however for the sake of understanding the wavy artifacts, we must only know that these timers run asynchronously of the main program and can be programmed to run at a specific set of frequencies. On the Arduino AVR boards (UNO, Leonardo, Micro etc.) the frequency of the PWM signals generated by the `analogWrite` function lies at around 490 Hz on all PWM compatible pins, with the exception of pin 5 and 6 which produce a PWM signal with a frequency of approx. 980 Hz.

Since the PWM signal runs asynchronously to the main program, any decimal difference in the frequency of the PWM signal driving the LED (and thus the photoresistor signal), and the frequency of the capture signal will cause the peaks of the PWM signal and the capture signal to be "out of sync".

In more mathematical terms, let f_L be the frequency of the PWM signal driving the LED, $f_P$the frequency of the photoresistor signal and $f_C$the frequency of the capture signal. Since the photoresistor signal is directly dependant on the LED signal, we may assume that:

$$f_L = f_P$$

The peaks of the PWM signal and capture signal will be out of sync whenever:

$$f_C \neq n \times f_L = n \times f_P \text{where } n \in \mathbb{N}$$

Shall this be true, then the peaks of the photoresistor signal can be observed to be shifting to the left or right of the capture signal with time. A shift to the right of capture signal can be observed if:

$$f_C > f_L = f_P$$

and to the left if:

$$f_C < f_L = f_P$$

On our oscilloscope we can clearly see that the frequency of the capture signal is lower than that of the PWM signal. We can observe a shift to the left of the capture signal:

$$t = 0$$

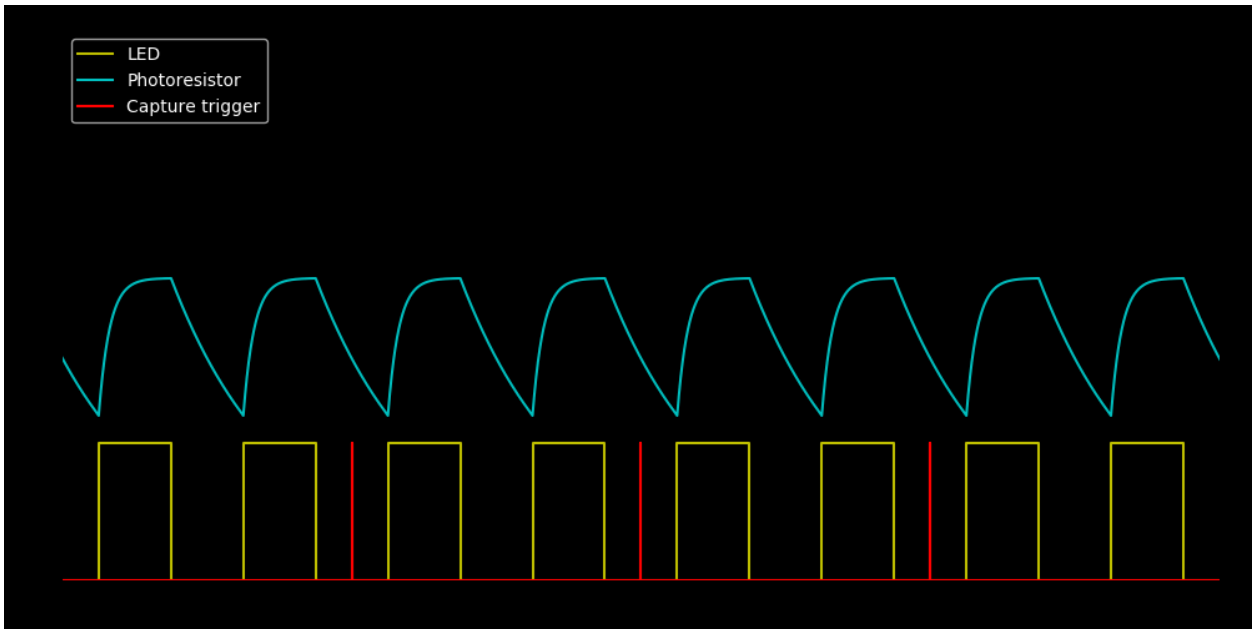


Fig. 10

$0 < t < d$ where d is the lowest common denominator of both frequencies:

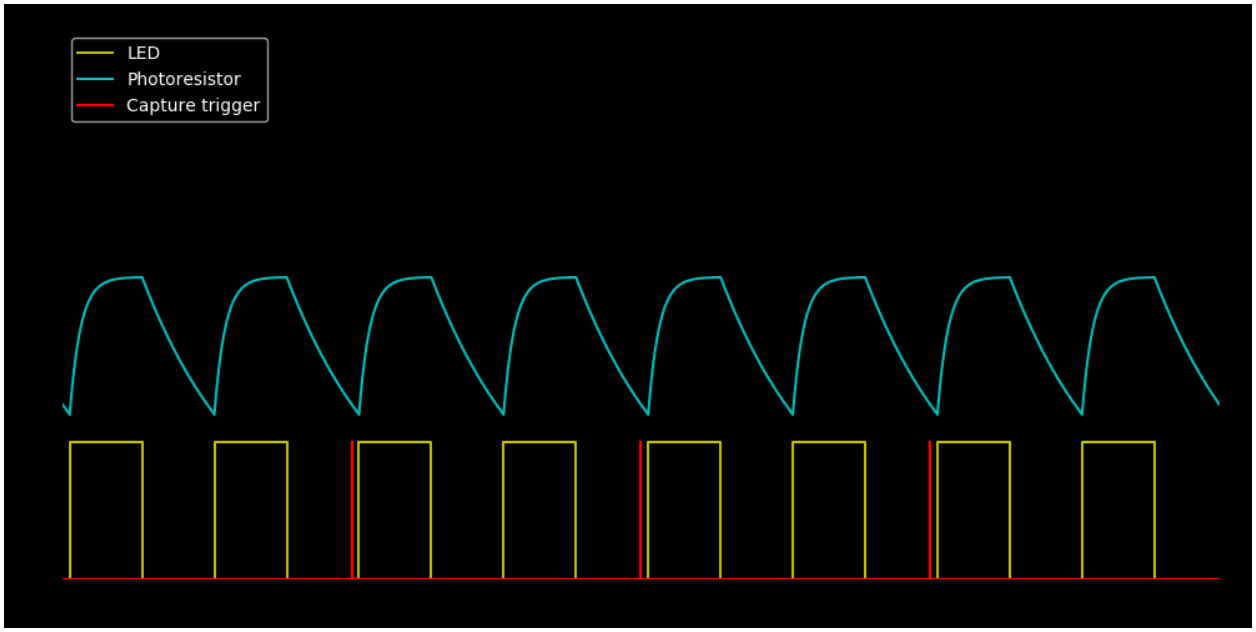


Fig. 11

$0 << t < d$ where d is the lowest common denominator of both frequencies:

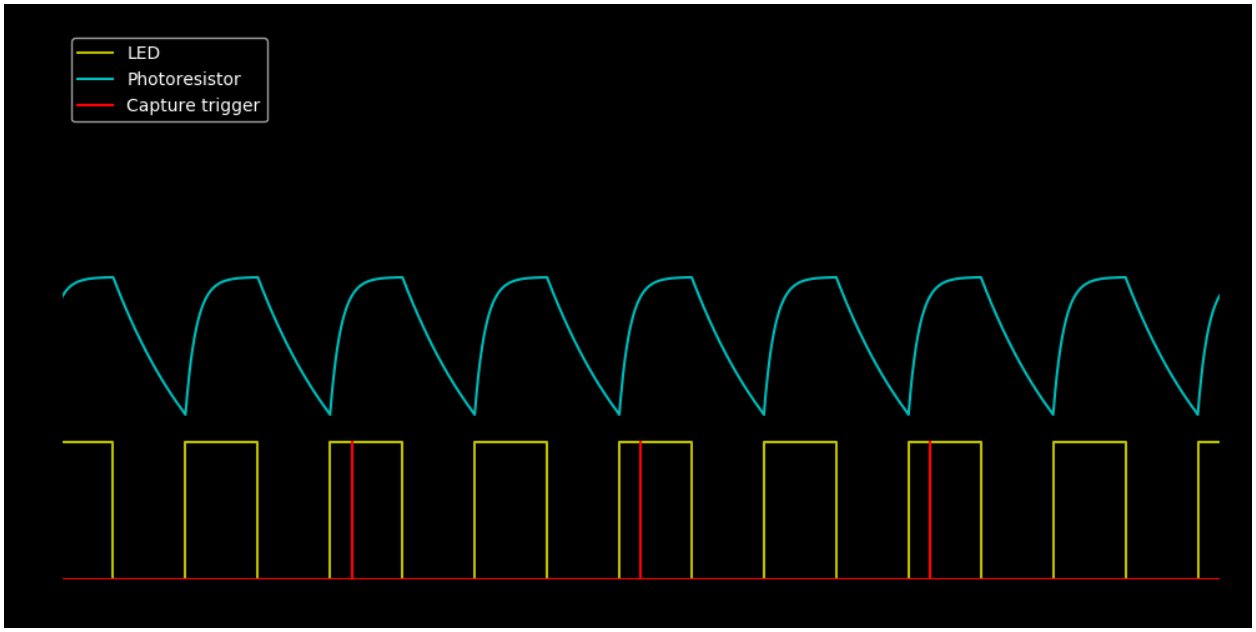


Fig. 12

The amount by which the peaks of the PWM and photoresistor signal shift to the left depends on the frequency and the least common denominator d of the capture signal and PWM/photoresistor signal.

The reason for the wavy artifacts should start becoming obvious. As the photoresistor signal shifts leftwards, the analogRead value "slides" across the sawtooth shaped photoresistor signal. During the HIGH duty cycle, the brightness of the pixel increases, however, as soon as the LOW duty cycle begins, the brightness of the pixel begins to drop gradiently.

After exactly:

$$t = d$$

where d is the lowest common denominator of both frequencies, the photoresistor signal will have shifted by the entire period of the capture signal, and the process repeats itself:

$$t = n \times d \text{ where } n \in \mathbb{N} \text{ and } d \in \mathbb{R}:$$

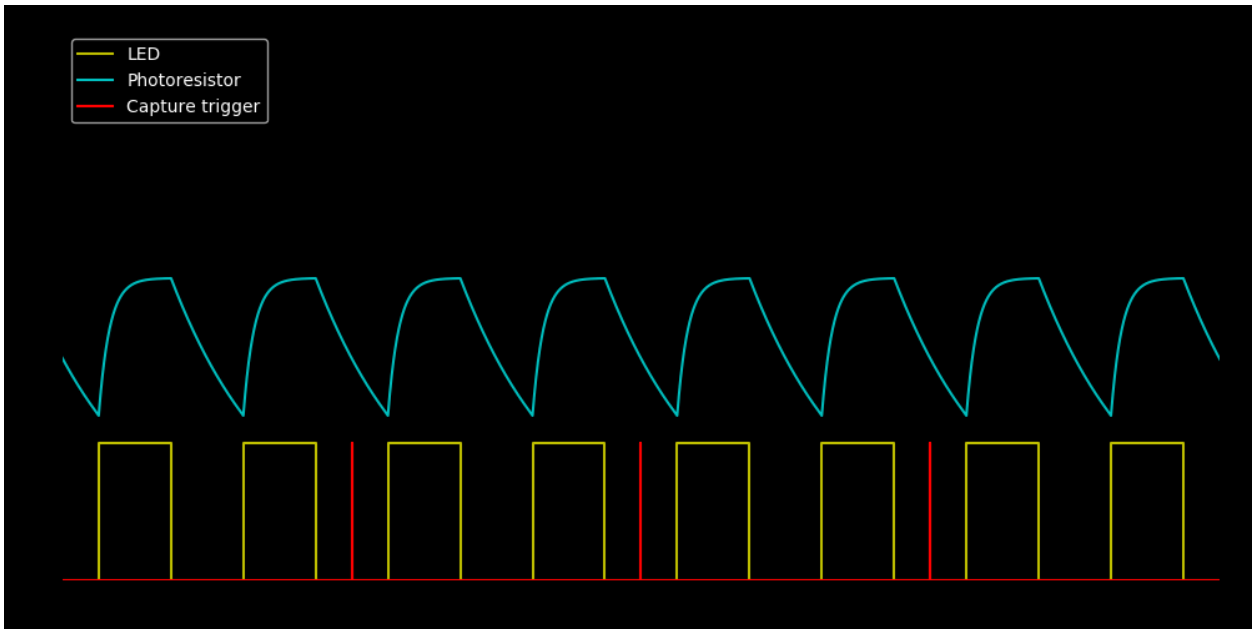


Fig. 13

One may speculate that the color shift may also be attributed to this phenomena. I render this as highly unlikely as the amount by which the PWM and photoresistor signals shift with every capture, is likely too small to produce any visible effects on the color of a pixel.

5. Testing my assumptions

From the information that I have gathered through this observation, I quickly figured I could test my assumptions by filtering the photoresistor signal through a decoupling capacitor. From the few spare capacitors that I had, I simply chose the one with highest capacitance (and thus reactance) and placed it in parallel with the photoresistor output and ground. On the oscilloscope, we would now see a constant voltage instead of the previously altering sawtooth like signal. This meant, no matter when the photoresistor signal was being captured, the captured value should have always remained the same for any set of equally colored pixels.

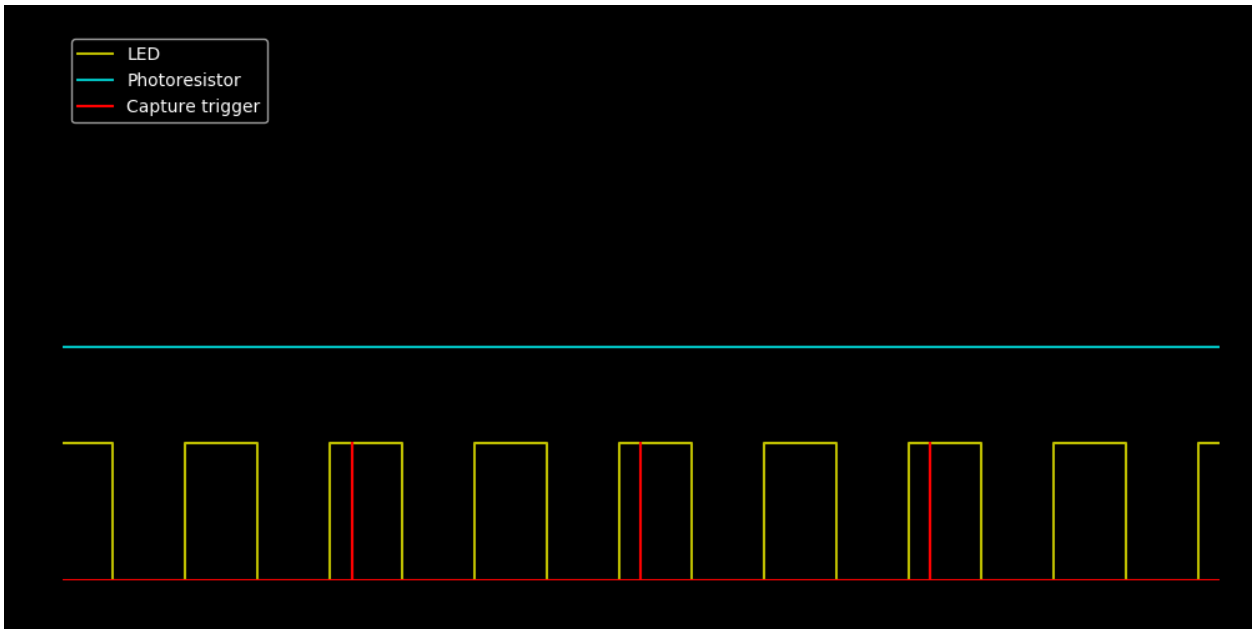


Fig. 14

To further emphasize on the changes, I have decided to add the decoupling capacitor midway of an image parse. That way, one half of the image was parsed without the decoupling capacitor, and was expected to produce the wavy artifacts, and the other part of the image was parsed with the decoupling capacitor, and was expected to be absent of the wavy artifacts.

The result confirmed my assumptions:

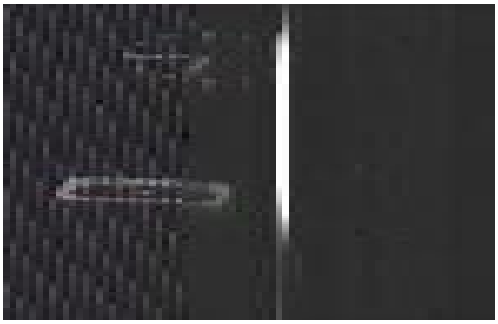


Fig. 15

On the left side, where the wavy artifacts are visible, no decoupling capacitor had been used. On the right side of the parsed image, where the wavy artifacts disappear, the decoupling capacitor has been added.

6. What's next?

At this point I have yet to understand what causes a shift in color/hue. Further, I wish to test whether driving the LED on pin 5 and pin 6 produce different artifacts, as the frequency of the PWM signal is higher on those pins.

7. Conclusion

In the end, the unique wavy artifacts presented by the OptoGlitch can be attributed to the very fact that the ATMega chip on the Arduino generates PWM signals independently of the main program. I would sincerely like to thank everyone person that took his or her time to read this paper. For feedback or any further questions, by no means hesitate to contact me via the following email:

ctx.xda@gmail.com

I would especially like to thank the Hackaday Blog team for publishing an article about the OptoGlitch as well as the *"Glitch Artists Collective"* Discord server for finding great interest in my work. I would also like to thank discord user "Pandela", also known as "*g-l-i-t-c-h-o-r-s-e*", for introducing me to the "Glitch Artists Collective" Discord server in the first place.

8. Appendix

Links:

[1] Indian Institute of Technology Kanpur, MICROCONTROLLER TUTORIAL II - TIMERS

http://students.iitk.ac.in/eclub/assets/lectures/embedded/Embedded-Old-2.pdf

References:

Fig. 3.1
Fig. 3.2
Fig. 3.3
Fig. 3.4

US Department Of Defense, 1963, MAN AND SAFETY - SUPERVISION, Educational film by the US Department Of Defense, Internet Archive, Accessed March, 2019

https://archive.org/details/gov.dod.dimoc.27623, Time: 1m 40s

Fig. 4

Philipp Henkel, 9 November 2018, Arduino Uno Microcontroller Board, A Scalable Vector Graphic by Philipp Henkel, Wikipedia, Accessed April, 2019

https://commons.wikimedia.org/wiki/File:ArduinoUno.svg

Fig. 15

Bell Telephone Laboratories, The Incredible Machine, 1968, Educational film by the Bell Telephone Laboratories, YouTube, Accessed April, 2019

https://youtu.be/iwVu2BWLZqA?t=408