ELT-2720: Electrical Project:
Motorcycle Signalling Vest Technical Report

Ben Buckley
Spring, 2022
Vt. Tech. College.

1: Introduction

In 2018, I attended the MoMA exhibition **Items: Is Fashion Modern?** I was not then, nor am I now, a fashionista. I was a motorcycle rider doing good-partner duty to the designer I was dating. I expected to be thoroughly bored, and luckily I was proven wrong. The entire exhibition was fascinating, but for me the star was Asher Levine and Smooth Technology's prototype biker jacket. The designers proposed integrating addressable LED lighting into the jacket itself, with the intention of using it as an extension of the bike's turn and brake signals.

I was struck by the strange lines and the swirling colors, and intrigued by the blend of electronic technology and utilitarian clothing. I was also dismayed at how impractical and delicate the jacket appeared to be, and by the fact that it didn't actually do the thing it was intended to do. As near as I can tell, Levine and Smooth never actually implemented the signalling functionality of their prototype, settling for a very pretty jacket with pretty effects. I promptly forgot about the jacket until a friend and I were joking about having a way to tell other drivers **exactly** what we thought of them. This led to speculating about wearable LED grids and voice-to-text software. I had recently written grid-to-serial display software for the WS2811 RGB LED addressing protocol and idly Googled "wearable LED DIY."

This led me to Enlighted Design, WS2811 "flat" package RGB pixels, and the FastLED library. It was clear that all the pieces were there to make the vest. I decided to use this Projects class as an opportunity to make a functional, practical version of the MoMA prototype. Something much humbler, but that would actually turn my riding gear in to my signalling gear.

The garment that resulted is a fully functional set of animated LED traffic signals integrated into a classic black denim Levi's riding vest.

2: Specifications

Power:

12 V DC (supplied by motorcycle)

4.1 A (all LEDs full bright white)

Microcontroller:

Arduino nano

Display refresh rate:

24 Hz

3: Design

3.1: Overall System

The blinker vest as a complete system consists of three phases of operation: acquire signals, process signals into animation frames, and output frame to grid. As shown in the block diagram in *Figure 1*, signal acquisition (and conditioning) consists of attenuating the inputs from 12V powers signals to 4V logical signals. The microcontroller process the signals and prepares the output in one process, and pushes the assembled grid frames out to the pixels in another. This process is elaborated in the state machine diagram, below right.

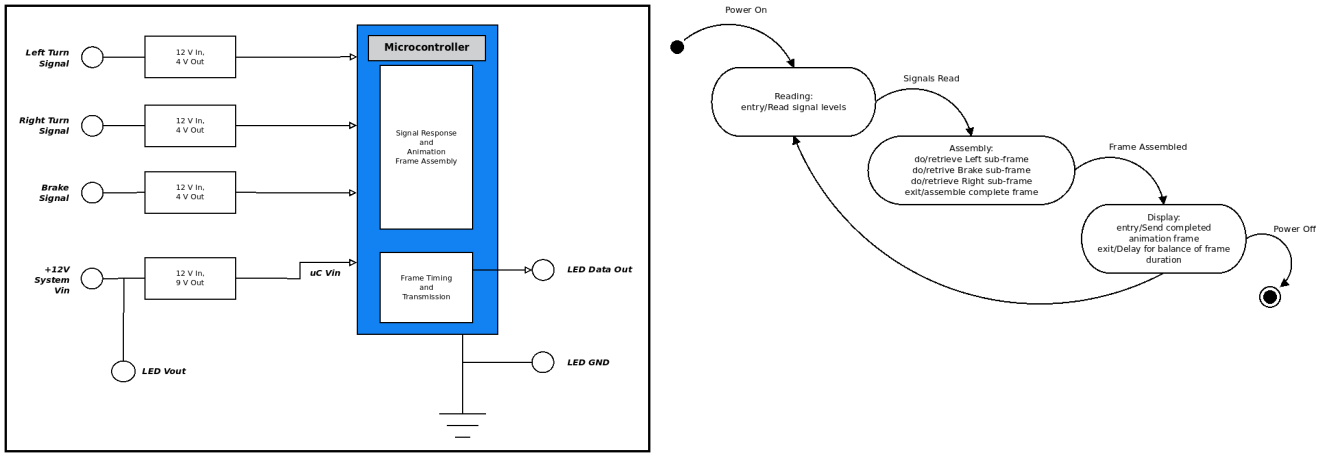

Figure 1: Block diagram of the system functions (left) and pseudo-state machine diagram of the software logic (right)

3.2: Control Circuit

When I first sat down to design the control circuit for the vest, I tried to think of everything. Every possible power source, signal conditioning, voltage limiting, relays to ensure the controller was up before allowing signals, etc. Luckily for me, I have good teachers. Between the advise of my friend Joshua A. Newman and Professor Al-Bahri, I was convinced to simplify my circuit as much as possible. Ultimately, it takes the form of four voltage dividers, four fuses for current limitation, the microcontroller and not much else. *Figure 2*, below is the final control circuit schematic for the blinker vest controls. I delve into specific design choices below.

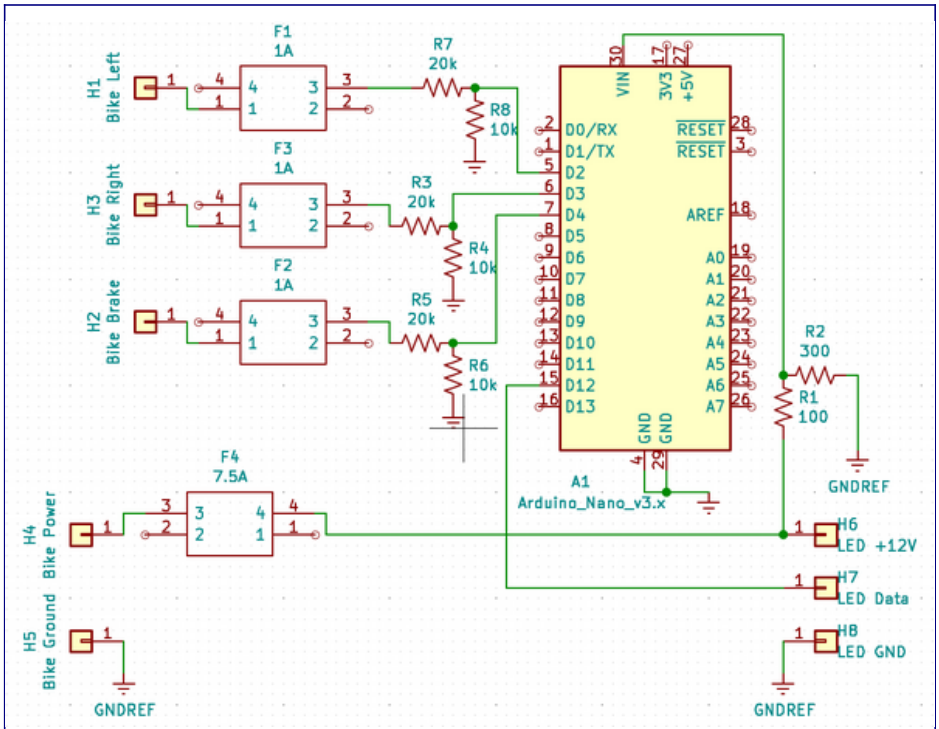

Figure 2: Final control circuit schematic

3.2.1: Nano Power Requirements

The Nano has an on-board 5V regulator that can be supplied at the V_{in} pin. According to the official documentation, this should be supplied with 7 - 12V (nominal) power.

Settling on 9 volts as a target V_{in}, I settled on a 1:3 voltage divider. By experimenting on a breadboard, I settled on 100 and 300 ohms; higher resistances starved the Nano of current and kept it from operating fully.

3.2.2: Signal Requirements

The ultimate power source is the motorcycles' nominally 12V DC power system. Since automotive power has a reputation for voltage swings, I aimed at voltage dividers that would give me acceptable levels anywhere from 10 to 15 volts on the input side. The Nano's digital pins read HIGH on 3.0 to 5.0 volts, so I set my sights right down the middle for 4V with a 2:1 voltage divider. The Nano's voltage-sensing digital pins allow a maximum current of 40 mA. I used resistors in the 10 kohm range to present the pins with less than a milliamp of current.

The 20 kohm resistors that ground the voltage divider also serve as pull-down resistors for the digital pins, preventing them from triggering on stray voltages.

3.2.3: LED Power and Data

The LED array runs at 12V, so I provided a pass-through power branch from the bike's power, as well as a terminal for ground and one for the data stream.

3.2.4: Pin Selection

Pin selection was primarily governed by the design of the software side of the controller. The turn signals are handled with hardware interrupts, so they need to be tied to pins D2 and D3.

The other pins were chosen for physical convenience during the physical construction of the controller.

3.2.5: Current Limitation

The vast majority of the current through this controller goes to the LED array. According to the manufacturer's datasheet, each pixel draws 60 mA, 20 mA for each of the red, green, and blue LEDs. I settled on a 4 x 17 grid, giving me a potential draw of 4.08 A for the lights alone. Keeping the idea of expanding to a larger or denser grid, I doubled that and came up with 8 amps, then settled for a 7.5 A fuse to keep the board from setting my vest on fire.

For the signal lines, I expect neglible current, so I sourced the smallest fuses I could find in the same package, 2A.

4: Hardware

4.1: Testing the Design

4.1.1: Voltage Dividers

With the control circuitry designed, the first order of hardware business was testing the voltage dividers to make sure they provide the voltages and currents I wanted. My very first builds used 10 and 30 kohm resistor for the Nano power supply voltage divider, which starved the board and kept it from powering up. Through experimentation, I landed on 100 and 300 as the largest viable combination.

Because of the potential for wide voltage swings on the bike's side of the circuitry, I had to make sure the 1:2 voltage divider would provide a logical HIGH at both ends of the expected voltage range. Testing this range is demonstrated in *Figure 3*, below. The Nano reads high between 3.0 and 5.0 volts, so the 10:20 kohm divider is spot on.

I wrote a very simple sketch for the Nano to light the extra on-board LED if a signal was detected at the digital pin. This is how I discovered that the by default the Nano's pins are floating and need to be tied low to ground to detect a positive HIGH signal.

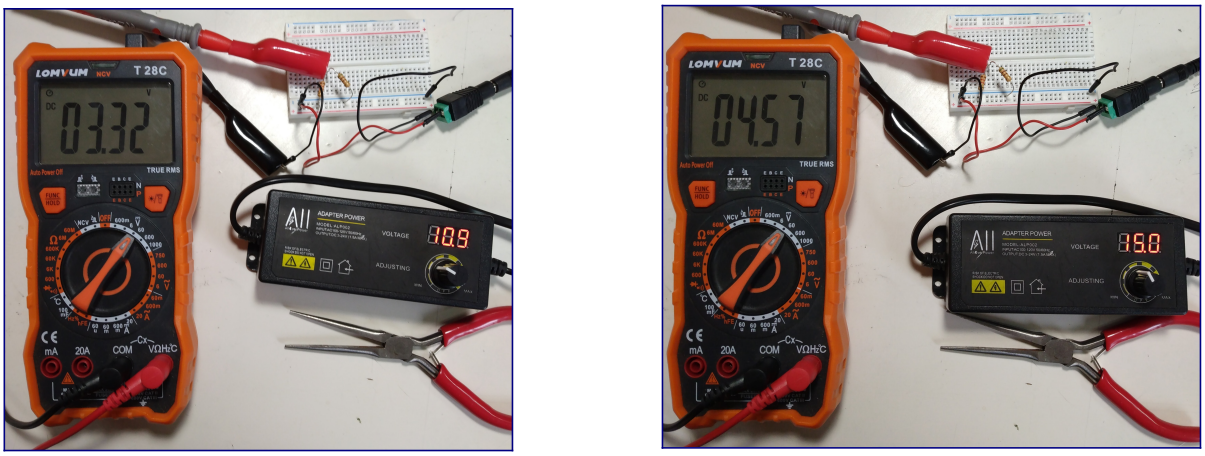

Figure 3: Testing the signal conditioning voltage divider. V_{sig} for V_{in} = 11V, 15V, and 12V (target)

4.1.2: Driving the LEDs

The voltage divider in the lower right-hand corner provided a 5V rail. I was in the process of sourcing more LED pixels and 5V is more common than 12V. In the end, I did not use 5V power, so this power source doesn't appear in any other materials.

I wrote a more elaborated version of the first sketch, for this, but it was lost when I botched a git merge. It's sole purpose was to drive the single pixel to respond to a low signal with Red and a high signal with Blue.

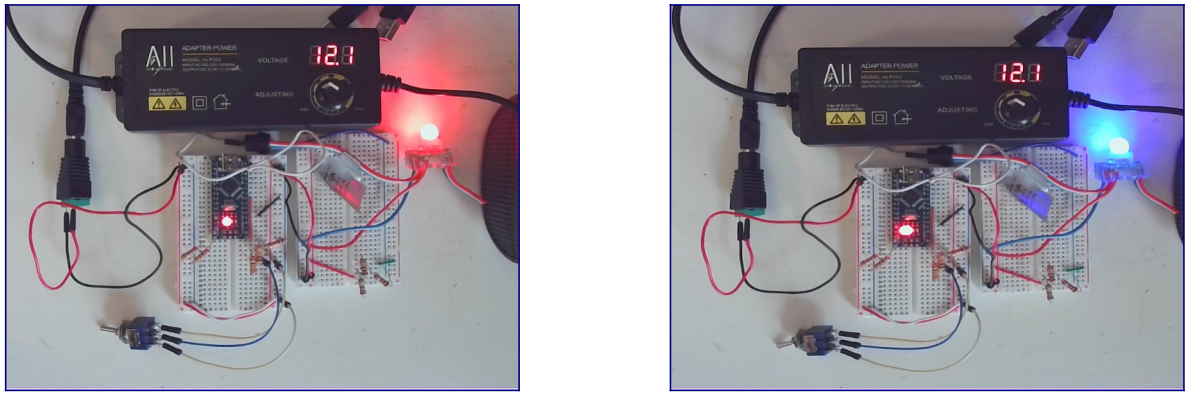

Figure 4: Prototype demonstrating all three basic functions of the system: input, processing, and output

4.1.3: The Final Prototype

Figure 5 shows the complete, final prototype. The three pushbutton switches simulate the signals pulled from the motorcycle's traffic signals. Not pictured is the output to the LED pixel grid, which consists of power and ground leads from the breadboard's rails and a data lead from pin D12 of the Nano.

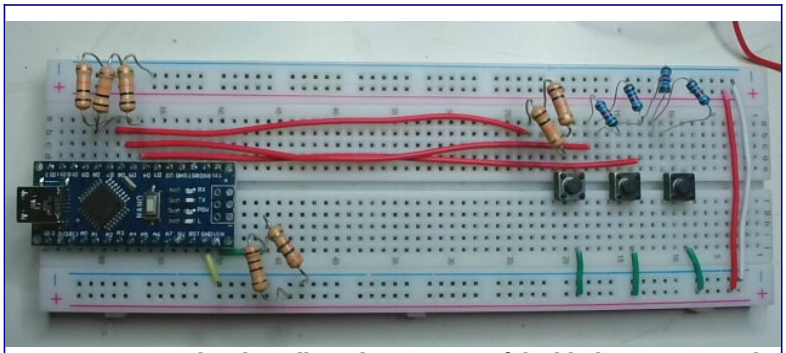

Figure 5: Complete breadboard prototype of the blinker vest control circuit

4.2: PCB Design and Build

Figure 6 contains the final schematic for the PCB of the control circuit, derived from the circuit schematic feature in *Figure 2*, above. *Figure 7* shows the finished, physical board, bottom and top. I go on below to describe the most important design considerations that went into this schematic.

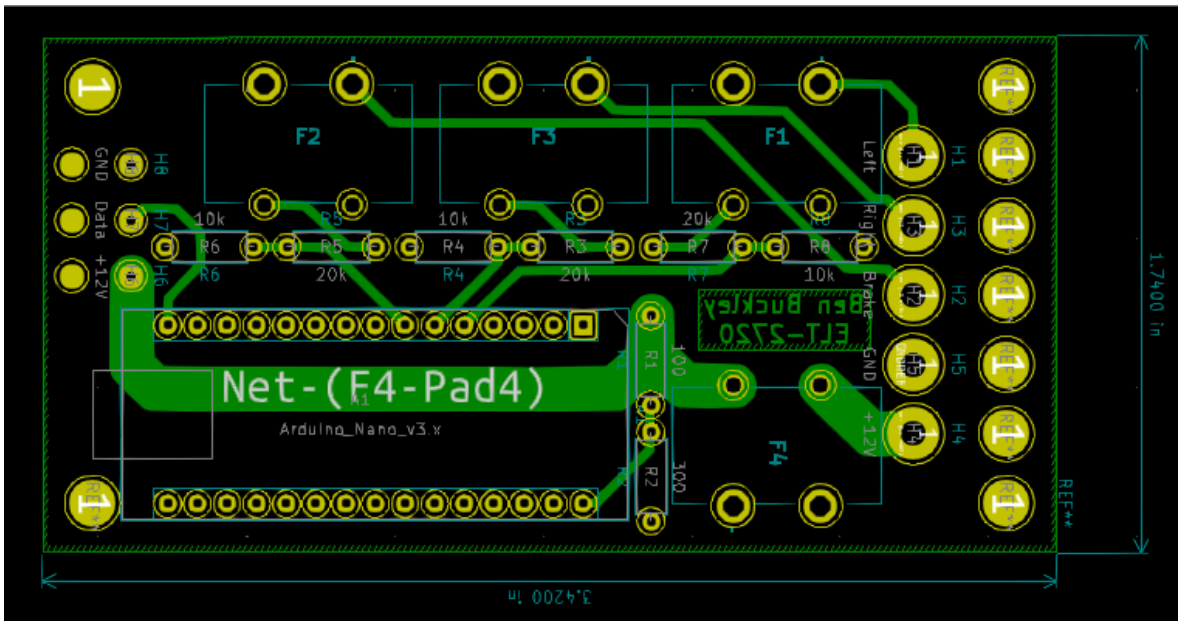

Figure 6: Final PCB schematic for the Blinker Vest control circuit

Figure 7: Finished control circuit PCB, bottom (left) and top (right)

4.2.1: PCB Design Considerations

The layout of the PCB was determined by following factors:

- **USB Port Availability**: As long as the project is in active development, I'll need access to the Nano's USB port. Since the USB port sits high on the microcontroller, some low-lying components can be between it and the edge of the board, but it largely defines one corner of the board. The digital pins have to be accessible to the signal conditioning elements, so that edge of the Nano goes to the middle of the board.
- **In one side, out the other**: The attachment points for the board leads define either "end" of the board and the minimum width of its narrow dimension. The bike-side leads are laid out to correspond to the physicsal order of the digital pins. Since D2 and D3 are the two pins that can provide external interrupts, the turn signal have to have the outer position. The brake is next because its trace can come down the inside the shortest distance without crossing the others. Ground comes next because it only has to connect to the ground pour. Power passes under the microcontroller all the way to the LED end of the board, so it's last.
- **Fuse order follows pin order**: Power goes on right, closest to the 150mm power trace and the V_{in} pin of the Nano. The signal fuses go down the left side of the board. Beside the Nano, they're the physically largest components, so they're stack height determines the length of the long dimension of the board.
- **Strain relief holes**: As part of a garment, the board and especially the board leads will be subject to a lot moving and bending. To minimize the chances of a broken joint at the board, the strain relief holes are the end-most elements of the board.
- **Corner mount holes**: With the other dimensions and spacings defined, the 4mm corner mounting holes are the last extensions of the board's footprint.

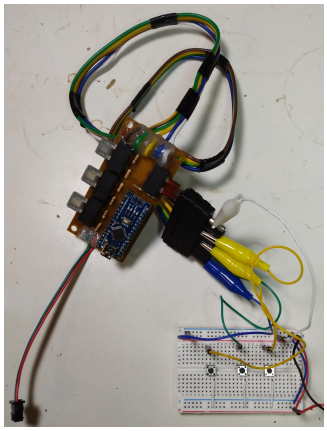

Figure 8: Fully finished control board with board leads, connected to the "motorcycle simulator" used to drive signals and power to the board.

4.3: The Vest

Of course, the control circuit needs a grid of LED pixels to drive, so I built one into a denim vest. *Figure 9* demonstrates the process of laying out the grid on the vest. Some of the assumptions and simplification I made in calculating the grid proved off, so careful inspection will reveal that the vertical spacing is somewhat more compressed under the collar, in the center of the grid. In practice, though, this seems to be offset by the curvature of an actual human back.

Figure 9:Marking the grid and installing the grommets.

Figure 10 shows the inside and outside of the vest with the LED pixels fitted into the grommets. A truly finished product would have all that loose wiring tacked down, and probably have the pixels secured by more than friction and compression, maybe a silicon epoxy.

Figure 10: The square-base LED pixels fit snugly into the grommets.

4.4: The Control Board Enclosure

4.4.1: Design

To build an enclosure for the control PCB, I used TinkerCAD and a 3D printer. The design process began with creating a volume that would fully contain the PCB and the first inch or two of the board leads. (*Figure 11, left*). Converting that volume to a hole and surrounding it with a solid rectangle produced a box that would fit the PCB. (*Figure 11, center*)I split this box down the middle laterally and transferred some of the perimeter from one half of the shell to the other to make fitting tabs that hold allow the wiring in and out of the bottom half of the enclosure (*Figure 11, right*). I added elaborations to the tabs and a couple of pip fittings to the rim of the shell in order to prevent slipping once the box is closed (*Figure 12*, left). Finally, I aligned through-holes for closure screws with a countersink for the screw heads (*Figure 12,* right)

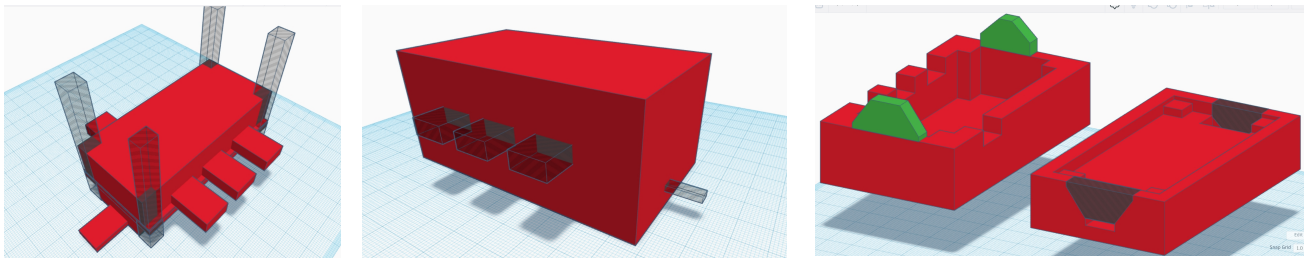

Figure 11: Figure 11: (Left) The negative of the negative volume of the enclosure. The holes at corners will translate into pillars of material for the screw holes to pass through. (Center) The shell built around the negative volume. (Right) Enclosure shell split, alignment tabs cut out and transferred

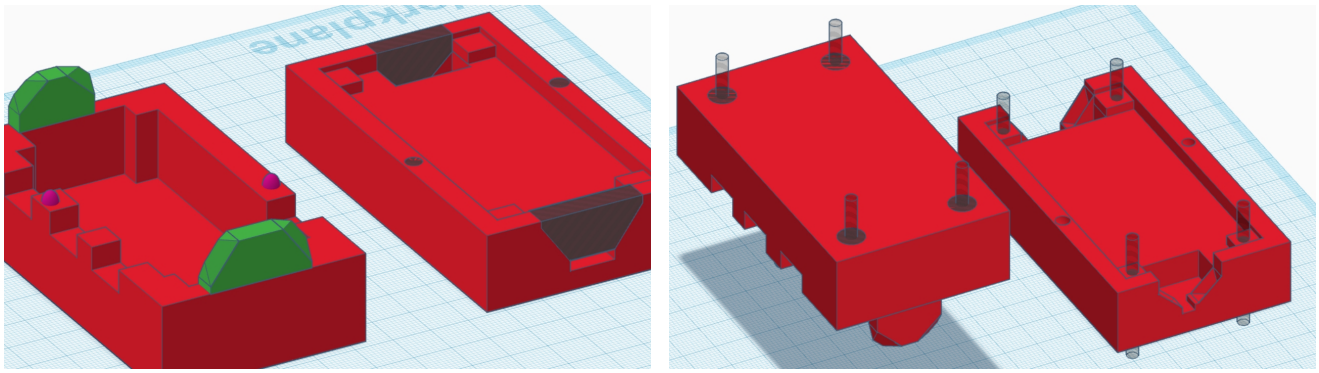

Figure 12: (Left) Adding alignment pips and bevels to the enclosure interfaces. (Right) Adding screw holes and countersinks.

4.4.2: Printing and Error Correction

I printed the enclosure using an Ender 3 Pro 3D printer; the print ran flawlessly. During the design of the enclosure, though, I worked with several assumptions that I didn't back up with measurements. As a result, the spacing between the fuse-holes was incorrect and the board cut-out was too tight. To correct for this, I used the chisel tip on my soldering iron--set to the lowest possible temperature-- to widen the holes and the cutout. I was able to the "squish" excess plastic to the edges of the form,

where it would form a bur. When this thin bur cooled it re-glassed and became brittle, allowing me to scrape it away with a razorblade. The result was a fairly clean set of corrections to the enclosure that allow the board to set snugly inside and the enclosure to close.

I also had to reattach one of the wire-passage tabs which snapped. Luckily PLA likes cyanoacrylate glue and the repair came off nicely.

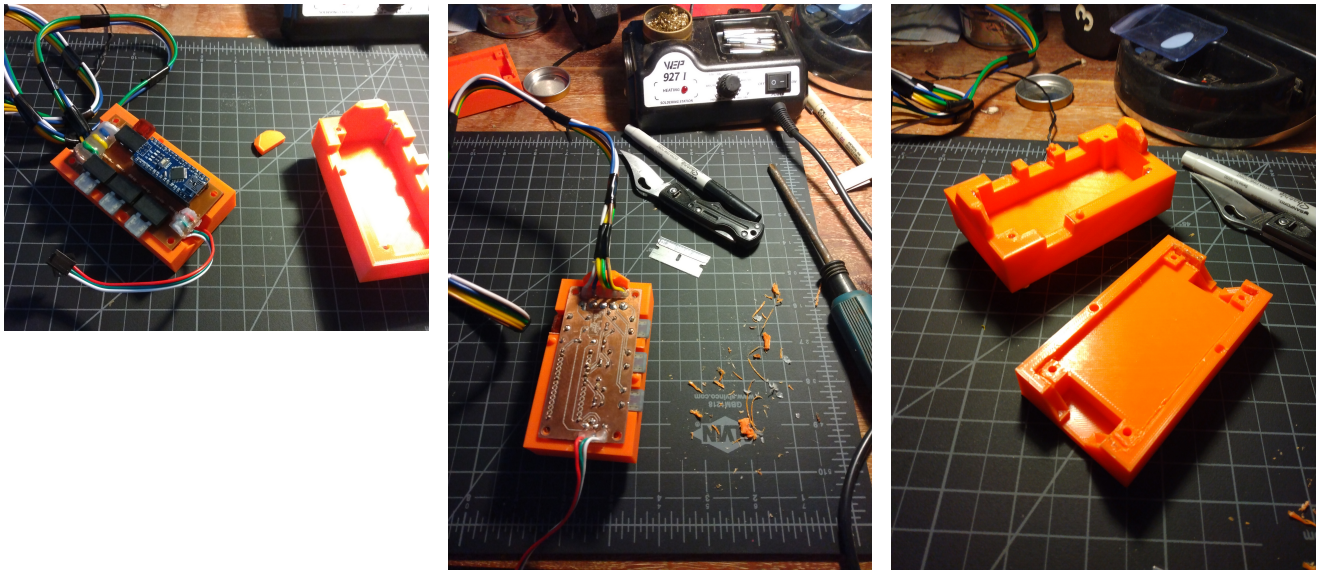

Figure 13: (Left) Damage and fitment issues; (Center) Fuse holes corrected; (Right) The wire tab re-attached and the board area widened

5. Software

Figure 14 lays out the states and state transitions of the microcontroller firmware. This state machine diagram is an elaboration of *Figure 1*, with signal handling and the timing interrupt separated out to represent the way I use ISRs to structure the assembly and timing of the outputs. In short, the turn signals are handled by External Hardware Interrupts that simply read a signal and set a state variable. These ISRs have no other interaction with the codebase. The Arduino loop() does (almost) nothing, with a Timer Interrupt calling output assembly routines and a display output function at a frequency set by macro at compile time. The design goal driving this structure was a desire to maximize the time available to do animation calculations by minimizing the execution time of every other part of the code while using an interrupt timer to create a stable, reliable time-base.

Below I walk through the code to examine the implementation of this design. It needs to be noted that this code hasn't undergone a final cleaning, so there are artifacts like unused variables and unnecessary logic that hasn't been cleaned out yet.

All of the code used here is original, with two exceptions. The first is the Timer Interrupt intialization, which is borrowed from an online tutorial and modified to suit my needs. The other includes all the calls to FastLED.h, a library of LED addressing protocols and fast mathematical functions useful for animating pixel grids.

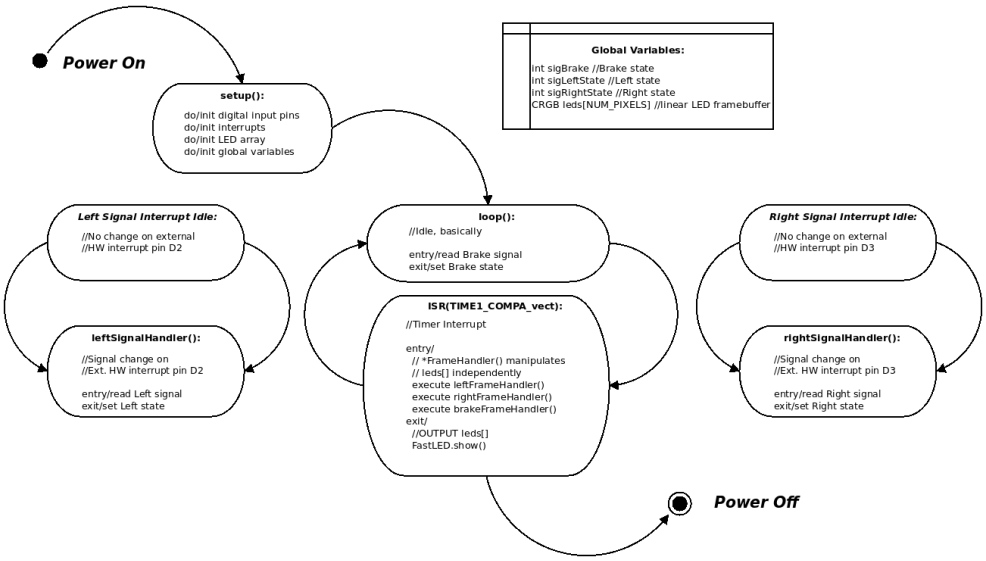

Figure 14: State machine diagram of the blinker vest firmware demonstrating the use of interrupts and global variables to minize CPU time.

5.1: Code Walkthrough

5.1.1: Macros and setup()

When writing C, I always attempt to parameterize my code as much as possible. This replaces numbers with names, making the code more flexible and readable. Here are the macros I use in the blinker vest firmware. This section allows me to easily change fundamental parameters of the system in a single location.

```
  8 //Brake Timer Interrupt Macros
  9 #define CLOCKSPEED 16000000
 10 #define TIMER_FREQUENCY 24
 11 #define CMR_RESOLUTION 1024
 12
 13 //Pin Definition Macros
 14 #define PIN_LEFT_SIG 2
 15 #define PIN_BRAKE_SIG 4
 16 #define PIN_RIGHT_SIG 3
 17 #define PIN_DATA 12
 18
 19 //LED PIXEL Macros
 20 #define LED_COLS 17
 21 #define LED_ROWS 4
...
 24
 25 //State Macros
 26 #define OFF 0
 27 #define ON 1
```

The Arduino-required setup() function provides for initializations. Here, the most interesting things are the calls to setupInterruptTimer() and the FastLED. cli() disables all interrupts; Line 192 contains the call to the timer interrupt setup code, which I'll look at later.; attachInterrupt() calls register the singal-

state handlers for the turn signal with the pins on the board, and set them to trigger on any signal-state change. sei() renables interrups now that I've added my own. Finally, line 199 is the initialization of the array of LED pixels.

```
181 void setup() {
182     signal = 0;
183     NUM_PIXELS = LED_ROWS * LED_COLS;
184 
185     //pinMode(PIN_LED,OUTPUT);
186     pinMode(PIN_LEFT_SIG,INPUT);
187     pinMode(PIN_BRAKE_SIG,INPUT);
188     pinMode(PIN_RIGHT_SIG,INPUT);
189     //digitalWrite(PIN_LED,LOW);
190 
191     cli();  //Disable all interrupts while setting up interrupts
192     setupInterrupt_Timer();
193     //Enable hardware interrupt on D2 for left turn signal
194  attachInterrupt(digitalPinToInterrupt(PIN_LEFT_SIG),leftSignalHandler,CHANGE);
195     //Enable hardware interrupt on D3 for right turn signal
196attachInterrupt(digitalPinToInterrupt(PIN_RIGHT_SIG),rightSignalHandler,CHANGE);
197     sei();  //Re-enable interrupts
198     
199     FastLED.addLeds<WS2811, PIN_DATA>(leds, NUM_PIXELS);
200 }
```

5.1.2: (Nearly) Empty Loop()

```
201 void loop() {
202 
203     //One var for each signal... replace with ext. interrupts?
204     sigBrake = digitalRead(PIN_BRAKE_SIG);
205 
206 }
```

This is the extent of the main loop. If the Nano had another external hardware interrupt, this function would be entirely empy. Of the three signals to the board, the brake is the one that demands the most immediate reponse, as it's most immediately relevant for safety. Thus the signal handling is done right here in the main loop, so that whenever the timer ISR fires, the brake state will have the most up-to-date value.

5.1.3: Turn Signal Handlers

```
146 void leftSignalHandler(){
147 
148     int sigLeft;
149     sigLeft = digitalRead(PIN_LEFT_SIG);
151     sigLeftStatePrev = sigLeftState;
152     if (sigLeft) { //If left turn signal is on now, it was off previously
153         // Do all the WAS-OFF stuff
154         sigLeftState = ON;
155         leftFrameCount = 0;
156     }
157     else {
158         // Do all the WAS_ON stuff
159         sigLeftState = OFF;
160     }
```

```
161 }
```
This is the code that executes whenever the Nano detects a change on the left turn signal. The code for the right turn signal is identical except for variable names. As written, there's a lot more going on here than is necessary. Tracking the previous state isn't needed, since there are only two states and we only get here when there's been a change. The left frame count initialization would be better handled in the frame assembly handler, where we could also implement tracking the state history if we need to for our animation routines.

Essentially, these routines **ought** to do nothing but read a pin and set a variable.

5.1.3: Timer Interrupt

The setupInterrupt_Timer() function does exactly what its name implies. It sets up a counter that divides the frequency of the Nano's clock (16 MHz). The heart of this function is setting OCR1A. This is the value of the Compare Match Register, a number of clock ticks at which the timer interrupt fires and executes the matching ISR. Here, I've replaced the example code's 15 Hz with a line that calculates the value of the CMR based on the macros defined in the head of the sketch.

In its current, final configuration, this value is set to produce a 24 Hz frequency for the ISR calls, giving the output display 24 frames/sec, which is high enough to allow for the appearance of the persistence of motion and low enough to give luxurious 40ms+ intervals in which to perform animation and display calculations.

```
52 void setupInterrupt_Timer(){
54
55     //This timer interrupt setup code borrowed from
56     // https://www.instructables.com/Arduino-Timer-Interrupts/
57
58     //set timer1 interrupt at 15Hz
59     TCCR1A = 0;// set entire TCCR1A register to 0
60     TCCR1B = 0;// same for TCCR1B
61     TCNT1  = 0;//initialize counter value to 0
62     // set compare match register for 15hz increments
63     //OCR1A = 1041;// = (16*10^6) / (15*1024) - 1 (must be <65536)
64     OCR1A = (CLOCKSPEED / (TIMER_FREQUENCY * CMR_RESOLUTION)) - 1;
65     // turn on CTC mode
66     TCCR1B |= (1 << WGM12);
67     // Set CS10 and CS12 bits for 1024 prescaler
68     TCCR1B |= (1 << CS12) | (1 << CS10);
69     // enable timer compare interrupt
70     TIMSK1 |= (1 << OCIE1A);
```

The ISR that handles the interrupt is very simple. In its current version, it includes calls to frame handler functions and executes the brake signal handling in-line. Ideally, the brake-signal handling would have its own functions for style's sake. Regardless, I consider the brake signal the most important of the three for safety, thus I designed this firmware so that the brake output is always written to the framebuffer last and the signal state is updated in every So, 24 times every second, the ISR executes, each handler fills in a portion of the output grid buffer matrix, and finally, it calls FastLED.show() to push the framebuffer to the display grid.

```
119 ISR(TIMER1_COMPA_vect){
120
```

```
121     //Handle left signal
122     leftFrameHandler();
123     //Right turn handling
124     rightFrameHandler();
125 
126 //##### The block between these comments should be extracted to a handler 
function
127 //##### ... but because it's the brake, it should always go last before 
FastLED.show()
128 
129     #define BRAKE_OFFSET 6
130     #define BRAKE_WIDTH 5
131 
132     //Quick-n-dirty brake signal handling
133     if (sigBrake) {
134        frameFill(BRAKE_OFFSET,BRAKE_WIDTH,0,255,255);
135     }
136     else {
137        frameFill(BRAKE_OFFSET, BRAKE_WIDTH,0,255,100);
138     }
139 
140 //##### That's the end of the brake handler.
141 
142     //Send the resulting frame to LED data out
143     FastLED.show();
144 }
```

5.1.4: frameFill()

frameFill() is the only routine that actually manipulates the framebuffer. In its present state, it fills a rectangular block within the framebuffer to a single color, saturation, and brightness. That's all that's needed to provide for the most basic animation consisting of blocks of color that change according to the input signals. It takes as arguments the horizontal position at which to begin filling, the width of the rectangle to fill, and the hue, saturation, and brightness to set them too.

The trick here is that the output isn't in actual fact a grid. WS2811 addresses individual elements in one dimension only; in order to make a grid with them (and avoid wildly complicated parallel output strategies), the string of pixels has to be "snaked" back and forth. Thus, the addressing within any given row of the grid proceeds in the opposite counting direction as the rows immediately above and below. To accommodate for this, frameHandler uses modular division by 2 to determine the "polarity" of the row-under-examination and set the start and stop points appropriately. Once the beginning and of a row are determined, a second loop fills the framebuffer appropriately.

```
 72 void frameFill(int offset, int width, int hue, int sat, int bright) {
 73 
 74     int startCell,
 75         endCell;
 76 
 77     for (int row = 0; row < LED_ROWS; row++) {
 78         if (!(row % 2)) startCell = (row * LED_COLS) + offset;
 79         else startCell = (((row + 1) * LED_COLS) - 1) - (offset + (width -1));
 80 
 81         endCell = startCell + width;
 82 
 83         for (int cell = startCell; cell < endCell; cell++) {
 84             leds[cell] = CHSV(hue,sat,bright);
```

```
 85         }
 86     }
 87 }
```

In order to generalize this approach, I would implement a framebufferGrid global variable that could be manipulated by the frame handlers. This grid buffer could then be traversed by frameFill() and translated onto the linear framebuffer.

5.1.5: Frame Handling

The frame handler functions are the heart of the output process. while fillFrame() does the grunt work of filling the frame buffer, the frame handlers do they more elevated work of deciding what goes into the framebuffer. These two functions are the linchpin that connects the signal handling with the output. This is where all of the animation happens, and that's by design. This architecture allows the animations for each signal to be changed independently in a single block of code. At present, the frame handlers are simple and symmetrical.

```
 89 void leftFrameHandler() {
 90 
 91     #define LEFT_OFFSET 0
 92     #define LEFT_WIDTH 6
 93     #define DUTY_CYCLE (TIMER_FREQUENCY/2)
 94         
 95     if (sigLeftState) {
 96         if (leftFrameCount == TIMER_FREQUENCY) leftFrameCount = 0;
 97         if (leftFrameCount == 0) frameFill(LEFT_OFFSET,LEFT_WIDTH,0,0,0);
 98         if (leftFrameCount > DUTY_CYCLE)
frameFill(LEFT_OFFSET,LEFT_WIDTH,40,255,255);
 99         leftFrameCount++;
100     }
101     else frameFill(LEFT_OFFSET,LEFT_WIDTH,40,255,100);
102 }
```

To begin with, I used macros to define a few values that describe the frame and the animation. The goal is to define a left rectangle within the grid, and when the turn signal is present, flash it between bright yellow and no light at 1 Hz with a 50% duty cycle. In other words, a basic blinker. TIMER_FREQUENCY is available because it was defined in the head in order to calculate the CMR value for the interrupt timer, so it's easy enough to divide in half for the DUTY_CYCLE. (This consideration partially governed the choice of an even number for the framerate.)

The logic of the animation is as follows:

- If there's a signal:
    - If we've seen 24 frames (one second), start over at 0.
    - If we're starting a new second, turn the frame black.
    - If we're entering the back half of the second, light up the frame.
- Without a signal:

- Fill with dimmer amber.

This setup immitates the behavior of a modern LED car turn signal, including going from partially lit to unlit before achieving full brightness, an effect that helps catch the eye of other drivers.

6. Conclusions

In the design and planning for this project, I set my self an imminently achievable goal. I knew going in that this project was on the simple side. That was a strategic choice on my part, facing a heavy semester and full work schedule outside of my coursework. It seemed to me that a simpler project had a better chance of success within the constraints and would allow more attention to go into solving implementation problems than churning through the design. As it happens, I was right.

I did create a Gantt chart for project, at the very beginning; it's below in *Figure 15*. While I honestly never referred back to the chart during the rest of the project, the intial exercise of thinking through all the different steps between design and implementation was enormously helpful. It helped me to build a model of the entire project process, to think through the relative weights of different tasks, and to anticipate potential complexities so I could simplify around them.

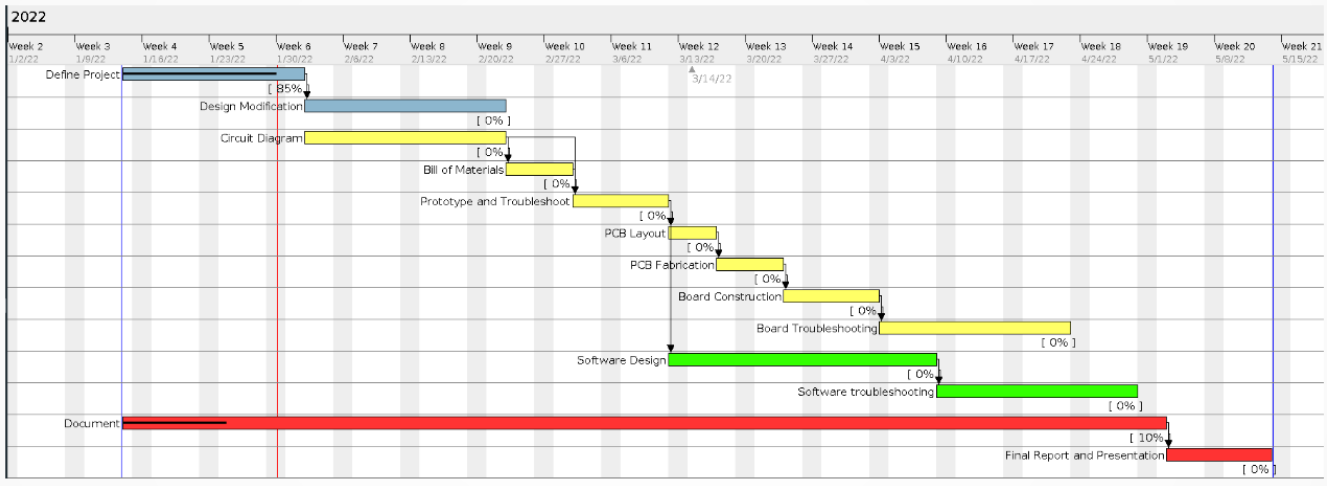

Figure 15: Gantt chart planning the tasks, dependencies, order and timing of the Blinker Vest project

Most of the challenges I faced along the way stemmed from learning new production systems throughout the process. This project was accomplished almost entirely with free/libre open-source software, using the lowest level tools available. This meant teaching myself a new circuit cad package (KiCAD)--including how to produce Gerber files from a schematic, the Arduino command-line development environment, and the use of the Jekyll static-website builder for the project's website. Here, however, is where my emphasis on simplicity truly paid off. By keeping the system itself simple, I afforded myself plenty of time to learn the tools and techniques necessary to fully and successfully execute the project.

I'm very proud of this traffic-signalling vest. I had an idea and executed it all the way to fully-functional physical system. If I were to do it again, or to expand on the idea, here are some possibilities:

- Reducing the size of the board and enclosure:

  Since the board and enclosure are designed to live in the vest's inner pocket, the smaller and less obtrusive, the better. Removing the headers under the microcontroller would go a long way to reducing the overall package's thickness, as would removing excess material from the underside of the box. Doing away with the fuses—possibly replacing them PTCs—on the board in favor of one on the external power line would also eliminate much of the board's footprint. Combined with a switch to SMD components, I could make this whole thing much smaller.

- Increasing the density/Reducing the bulk of the grid:

  The LED pixels I used are positively enormous for a garment; they couldn't physically be closer than they are. The pixels and the grommets add weight and waste display real estate. Other, smaller, WS2811 packages are available that fit in a smaller area. Switching the physical lighting elements would allow for a more flexible, comfortable vest as well as a denser grid producing more light and allowing higher display resolutions. These changes would incur proportional changes in cost, labor (assembly), materials (waterproofing) and power, but they would produce a finer product.

- Wireless signalling, battery power:

  Physically separating the controller from the signal source and giving it an independent power supply would improve the overall user experience by eliminating bulky, restrictive cabling. It would also present the possibility of multiple possible controllers for the vest, allowing it to adapt to say, bicycling or costuming more readily.

Having finally finished building this vest, I can reflect on my conclusions from the work. They're mostly about the planning and execution of projects. First in my mind is the value of good up front planning and systems thinking. I spent a lot of time and notebook paper imagining out the needs for each piece of the project, how they needed to interface, and what tasks and order of operations would get there. Turning this into a structured plan through proposals, presentations, and timing diagrams had me fully prepared to tackle each individual task in a timely, efficient way.

Along with this thought is the value of simplicity. Simplicity here could mean using only voltage dividers through passive resistors for voltage regulation, or it could mean relegating all the logic of animation to a microcontroller, where high-level programming erases much of the complexity of the underlying logic. In each case "best result for least effort" was guideline that helped keep the project well on the rails and allowed me to see it to completion.

Most importantly, though, I can't wait to take it out riding!

References

NB: The works cited below consists reference documents that informed my work on this project. Not all of them have been mentioned directly in the text above. They appear in the order in which I used them while building the vest.

ATmega48A/PA/88A/PA/168A/PA/328/P Datasheet. Microchip Technology Inc., 2018. Accessed at http://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061A.pdf

FastLED Documentation. FastLED Project. Accessed at https://github.com/FastLED/FastLED/wiki/

"Fusing Guide." 12 Volt Planet, 2022. Accessed at https://www.12voltplanet.co.uk/fuses-guide-uses.html

Hoskins, Matthew. "Arduino from the Command Line: Break Free from the GUI with Git and Vim!" Linux Journal, July 16, 2019. Accessed at https://www.linuxjournal.com/content/arduino-command-line-break-free-gui-git-and-vim

Rene_Poschl. "How to connect a wire to the PCB?" KiCad Info, Dec. 12, 2021. Accessed at https://forum.kicad.info/t/how-to-connect-a-wire-to-the-pcb/18262

Hoang, Khoi. "TimerInterrupt," Arduino Reference, 2022. Accessed at https://www.arduino.cc/reference/en/libraries/timerinterrupt/

Ghassaei, Amanda. "Arduino Timer Interrupts," Instructables Circuits. Accessed at https://www.instructables.com/Arduino-Timer-Interrupts/

"attachInterrupt()," Arduino Reference, May 21, 2021. Accessed at https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/

Reilly, Nash. "Using Interrupts on Arduino," All About Circuits, Aug. 12, 2015. Accessed at https://www.allaboutcircuits.com/technical-articles/using-interrupts-on-arduino/