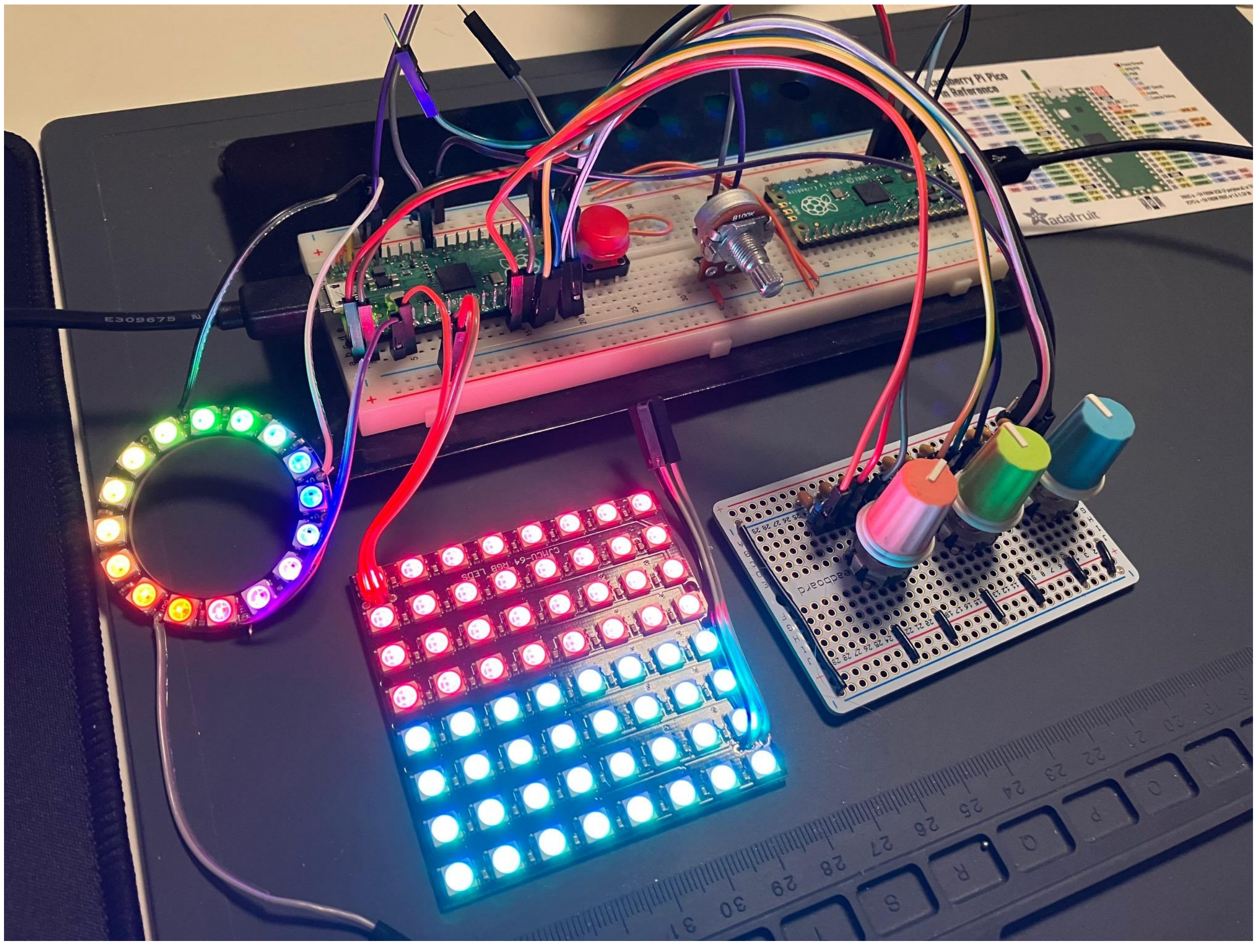


The palLED

A friendly tool for designing color palettes for addressable LEDs using RYB color theory

06/20/2022

Carrie Sundra
Alpenglow Industries
https://www.alpenglowindustries.com
info@alpenglowindustries.com
@alpenglowind on twitter, instagram

Application Description

The palLED is a friendly kit that helps you envision color palettes on addressable RGB LEDs, using RYB color wheel theory. There's a big difference between how colors look on a computer monitor, and how those same RGB codes translate into colors on addressable LEDs. Additionally, traditional color theory uses a color wheel with red, yellow, and blue as primary subtractive colors, whereas LEDs generate light using red, green, and blue in an additive manner. Relating RYB theory to RGB space isn't intuitive, and the palLED lets you see how color schemes created using RYB theory look on RGB LEDs, and also provides you with the RGB settings for each LED.

This website provides an excellent example of the type of color schemes and color mapping that this project aims to achieve with addressable LEDs: https://www.sessions.edu/color-calculator/

If you're looking to develop a specific color scheme to create a certain mood, or if you're developing or re-creating specific brand colors, you can break out your trusty palLED and immediately see how they'll look on your addressable LEDs. You can also add diffusion panels to see how they'll look behind acrylic, silicon, or inside an enclosure.

Video showing the project:
https://www.youtube.com/watch?v=KaCqmaZs_2U

Codebase:
https://github.com/casundra/MakingEmbeddedSystemsClass/tree/main/Final%20Project

Hardware Description

The palLed is currently a desk proof-of-concept which uses solderless and solderable breadboards to plug everything together. See the cover photo for visuals.

Microcontroller & Serial Data I/O

The microcontroller is a Raspberry Pi Pico in a solderless breadboard. It uses the RP2040 microprocessor which is a dual ARM Cortex M0+. The onboard Pico USB connector is used for power, programming, and a serial console (UART over USB). A second RPi Pico is set up as a Picoprobe for debugging.

Color Display

The matrix and ring are collectively called “strips” in this report and the codebase.

LED Matrix

An 8x8 matrix of WS2812B addressable LEDs is plugged into the Pico and powered off USB. https://www.amazon.com/gp/product/B07SQXCM87

LED Ring

A 16 LED ring of WS2812B addressable LEDs is also plugged into the Pico and powered off of USB. This was used: https://www.adafruit.com/product/1463 In retrospect, it would have been a little better to use a ring that has a number of LEDs that’s a multiple of 12, that way colors that are arranged at 120 degrees and at 90 degrees would both be able to be displayed accurately. Twelve would have been a little too few LEDs to show the color wheel’s range well,, but a 24 LED ring like this would be perfect: https://www.adafruit.com/product/1586

User Input

There are currently 4 knobs and 3 buttons available, although only one button is currently implemented.

Knobs / Rotary Encoders (Quadrature)

These following quadrature phase encoders with built-in buttons were used: https://www.newark.com/alps-alpine/ec12d1564404/encoder-vertical-12mm-30det-15ppr/dp/75T6731

The encoders change functionality depending on mode:

- RGB_PICKER: each knob adjusts R, G, and B values of the LED.
- HUE_PICKER: Left knob changes H, or hue, or the color according to its angle on the wheel. Other knobs are ignored but will eventually likely be S and L if they don’t conflict with brightness.
- COMPLEMENTARY: Knobs are currently ignored, in the future they would allow some adjustments of hue and other color display settings.

These are rotary quadrature phase encoders because they need to be agnostic to absolute position, as it would be extremely disruptive to the user if settings were retained between modes since the active color would then change.

They are hooked up to digital i/o pins with interrupts in the codebase so that changes in position can be captured and accumulated. Encoders are basically fancy switches, and they are debounced with an RC filter in hardware.

Knob / Rotary Potentiometer

This is used to adjust overall brightness of the LED strips, and as there are fixed endpoints and the user should understand where the knob is set between them, an analog rotary potentiometer is used.
https://www.digikey.com/en/products/detail/tt-electronics-bi/P160KN-0QC15B100K/2408877

It is hooked up to an analog input and converted to position using the ADC module.

Buttons (Momentary, NO)

Each encoder actually has a button built into the stalk, so it's not a separate piece of hardware. Only one button is currently used; it is the Left encoder and it switches palLED modes. It is hooked up to a digital i/o and it is debounced with an RC filter in hardware.

Software Description

Main Loop

The palLED currently has very basic functionality. There are 3 modes: RGB_PICKER, HUE_PICKER, and COMPLEMENTARY. The main loop checks for button presses that would change the state, goes to the state machine and performs functions related to that state, checks the brightness knob reading, displays patterns on the LED strips, checks the serial Console, and keeps the onboard LED heart beating.

State Machine

The state machine is pretty simple at the moment, future development may break out instructions into a separate "state" library as functionality grows. The RGB_PICKER state checks the status of the encoder knobs and changes RGB values for the active color accordingly. The HUE_PICKER state adjusts hue for the active color based on the left knob. It currently transforms RGB to HSL and back to do this; next version of the code will have active color information stored persistently for both RGB and HSL. COMPLEMENTARY roughly calculates the complement color, and splits the matrix display into half active and half complement. Encoders are ignored.

LED Pattern & Color Generation

This is the meatiest portion of the code and could likely be broken into smaller modules going forward. It handles all colorspace conversions, packs colors into the color buffer for each strip, and handles general data about the strip (# of LEDs, brightness setting).

I severely underestimated the amount of data manipulation needed to display a color at the appropriate position on the color wheel. It turns out that this is not possible to do by just looking RGB data, because RGB is basically a plot of color on a cube. Instead, I had to transform the RGB data to HSL (hue, saturation, luminescence) colorspace, which is cylindrical, where hue is the angle of the color around the color wheel (ultimately in degrees in my code). I used tutorials I found on the internet here: https://www.niwa.nu/2013/05/math-behind-colorspace-conversions-rgb-hsl/

The tutorial went through the steps, but the code is my own.

I also learned that our eyes don't perceive brightness in a linear manner, so had to incorporate gamma adjustment. Basically, we perceive many shades a low brightness, but as brightness is increased, we just perceive "ow, bright". For a brightness knob to exhibit what we think is a linear change in brightness, we actually have to apply an exponential correction. This is done with a lookup table I got from adafruit: https://learn.adafruit.com/led-tricks-gamma-correction/the-quick-fix

It's very fast and convenient, for each brightness 0-255, a corresponding brightness is at that index in the table, which is gamma-corrected.

In order to display a pattern on the matrix, the active color is sent to a function that generates a matrix of the Color structure (just 8-bit .red, .grn, and .blu values), one element for each LED. It then goes to the showIt function which gamma-corrects the brightness, then sends the color to the WS2812 driver which sends it to the LED.

In order to display a pattern on the ring, the active color is transformed from RGB to HSL, the color angle is used to determine which LED should be lit, and then a for loop is used to pack the ring with unlit LEDs at all other positions. It then goes to the same showIt function which gamma-corrects and then sends the color to the WS2812 driver.

LED Display

Fortunately RPi had example code for WS2812 driving through the PIO interface and state machines. I took their example (which was one .c file) and broke it out into a header and driver .c file. They had an example using DMA, but I wasn't able to successfully break it out into its own .c and .h, and since the palLED currently doesn't care if the LED display is blocking (we're not displaying any animations or dithering), I left that for a future improvement. The driver function is called in a while loop that goes through the strip's color buffer (LED by LED).

Brightness Reading

The ADC is polled once per loop and the brightness value is converted from 12 bits to 8 bits and written to the LED strip information.

LED Strip and Color Info Storage

I went around about this a bit and still think it could use improvement. I'm using a Color structure to store RGB info. I then use an array (Color[]) to store info for each LED in the strip. I liked this because I could write very human readable code with ".red" and ".grn", etc. I could likely have used a 2D or 1D array instead. Strip information is stored in a separate structure, and I likely could have combined the 2 structures, but again, I didn't want to be burdened by a lot of this.that.theotherthing style access. I appreciate C++ and classes and class constructors more now. I'd like for this to be easily extensible by someone else, without having to dig all over the code to add constructors or cases for a new LED strip.

Interrupts

Both phases for each encoder and one encoder button (at the moment) are attached to GPIO interrupts. The RP2040 has one main GPIO interrupt that covers all of them, so the interrupt routine also has to determine who called it. In the case of the encoder phases, a simple function is quickly called that determines the direction of travel (CW / CCW) and either increments or decrements the encoder count. This was a bit tricky - generally, I'd set up the pins to interrupt on CHANGE (either RISING or FALLING) in order to catch both edges of one phase. This is needed to sense each detent, and increment the count once per detent which is a typically expected behavior. However, the Pico surprisingly did not have a CHANGE interrupt, so I'm sensing one phase on rising and the other on falling, and that works. An encoder_count_update flag is also set which the main loop checks.

For the encoder button, the pin is quickly read to verify the press and an update flag is set.

Serial Console

This is Elecia's Consolinator code that is entirely ported into this project. In the future, some of the example functions could be cleaned up, and I could make better use of the existing printf functionality and likely get rid of the Consolinator functions that print specific formats. I believe there's a bit of duplication. Additional commands of "print" and "test" were written, which respectively print out all LED data, and do a print test for printing LED data which was useful for testing new colorspace conversion code, as I could easily adjust it to print only a single LED's worth and also include print debugging for intermediate variable in the functions I was developing.

Onboard LED Heartbeat

Pretty standard, keeps track of the time it last blinked and turns the LED on or off according to the current time. Would like to eventually explore using the built-in Pico event timers for this.

Licenses

All underlying RPi Pico SDK code is under a BSD-3 license which has the additional clause that you can't use Raspberry Pi's or contributors' names to endorse or promote your products without permission. The promote is interesting, of course people say all the time that an RPi Pico is used in their device and there seems to be no problem with that. I suspect it's mainly the endorsement that they're concerned about.

Elecia's Consolinator code is under an MIT License.

All code I've written is also under an MIT License.

Both BSD-3 and MIT licenses allow for commercial use, distribution, and modification. All code used should be fine if we develop and manufacture a kit out of the palLED. We will of course keep all original license text within the codebase.

Firmware Update

A UF2 bootloader is built into the RP2040 platform and was the primary method used to program/reprogram the Pico during code development. For a user to flash code, they need to push the BOOTSEL button while applying power to the Pico. The Pico then goes into bootloader mode and mounts as a USB flash drive. Then the user drags and drops the Application.UF2 file that we'd provide onto the flash drive, and the new application code in the UF2 is loaded. The user would need access to the BOOTSEL button or it would need to be remoted.

The Application.UF2 file is generated every time the code is compiled and can be found at:
C:\Users\<your username>\Documents\<your repo folder>\.pio\build\raspberry-pi-pico

Architecture Diagrams

State Table

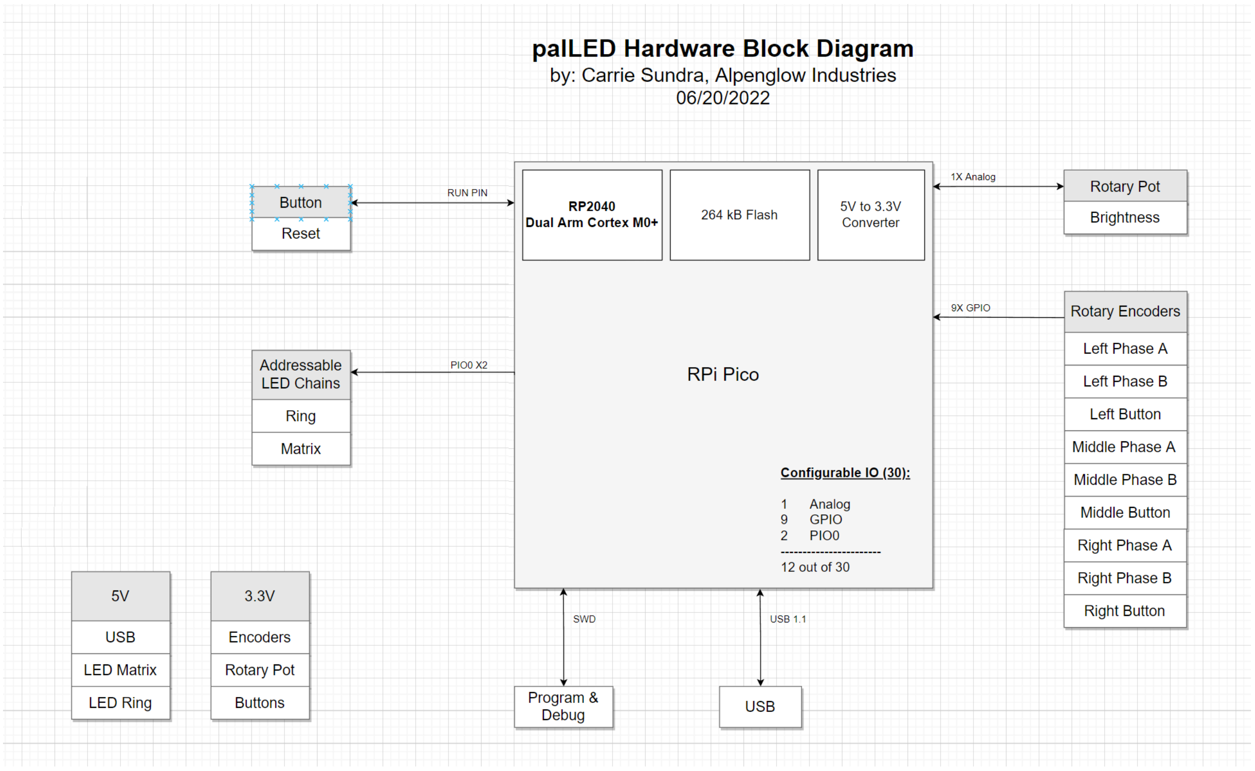
https://docs.google.com/spreadsheets/d/1Bwn0tV4N7vu0LyWxg6XoQYavriNirxIgOVEE_6TinkA/edit?usp=sharing

Better viewed through the link above

State #	**State Name**	**No. Colors**	**Action**	**Left Knob**	**Middle Knob**	**Right Knob**	**Brightness Knob**	**Mode Button**	**Serial In**	**Serial Out**	**LED Ring**	**LED Matrix**
0	RGB_PICKER	1	adjusts RGB values in activeColor depending on encoder movements	activeColor.red	activeColor.grn	activeColor.blu	adjusts overall brightness for both strips	next mode	console commands	activeColor when changed, data via serial command	activeColor	full activeColor
1	HUE_PICKER	1	converts to HSL, adjusts H depending on encoder movements, updates activeColor	hslColor.hue	X	X	adjusts overall brightness for both strips	next mode	console commands	activeColor when changed, data via serial command	activeColor	full activeColor
2	COMPLEMENTARY	2	displays a fixed color wheel, displays activeColor and its complement on the matrix, ignores encoder movements for now	X	X	X	adjusts overall brightness for both strips	next mode	console commands	activeColor when changed (not currently possible), data via serial command	fixed wheel	half activeColor, half complementary color

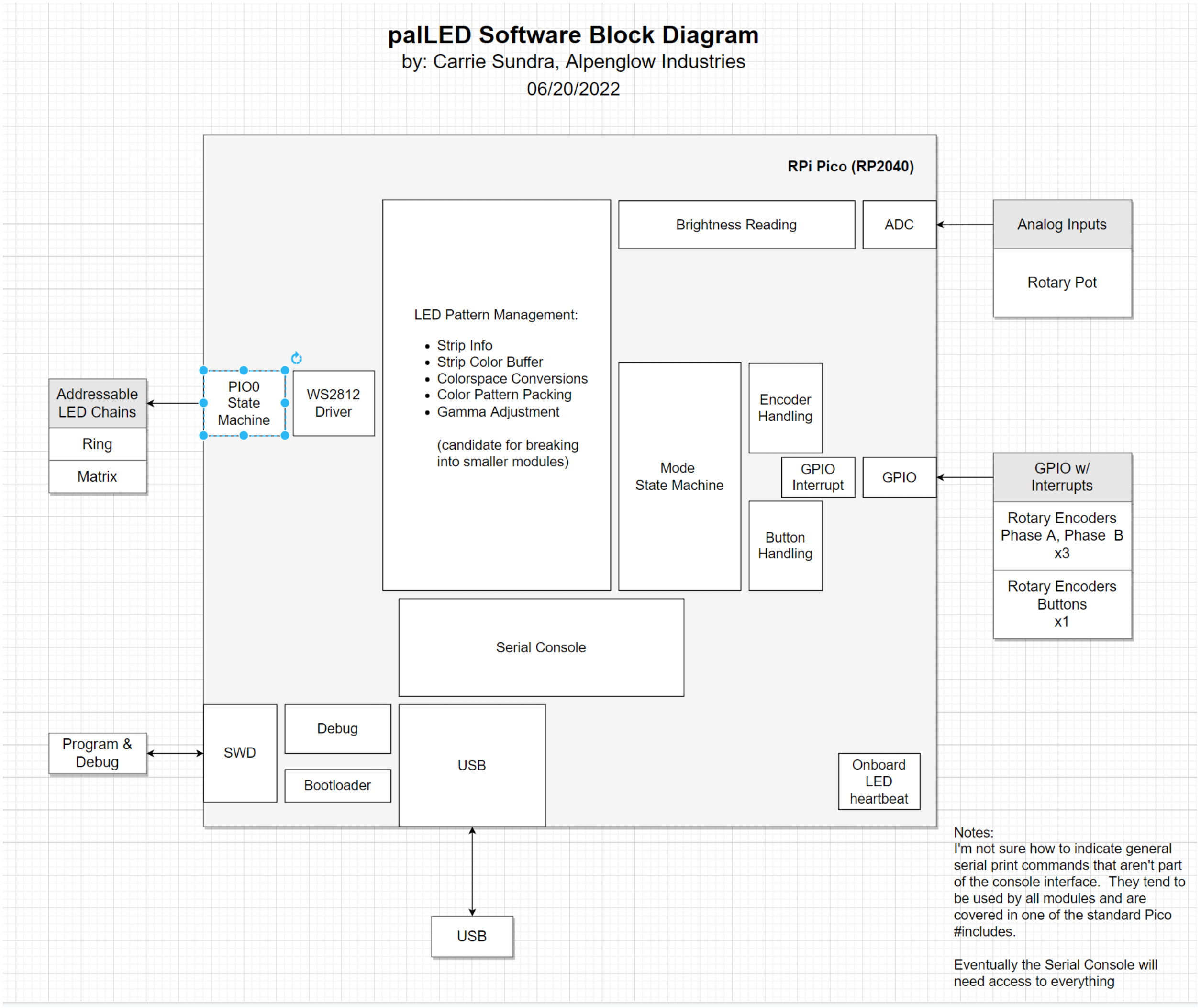
Hardware Block Diagram

https://drive.google.com/file/d/1sUODYfhG_7mndhHARYSBbNzuElzD7EUt/view?usp=sharing



Software Block Diagram

https://drive.google.com/file/d/1tTngIRLrHbXTuAVWzzBSSVFKVUn2dkVf/view?usp=sharing

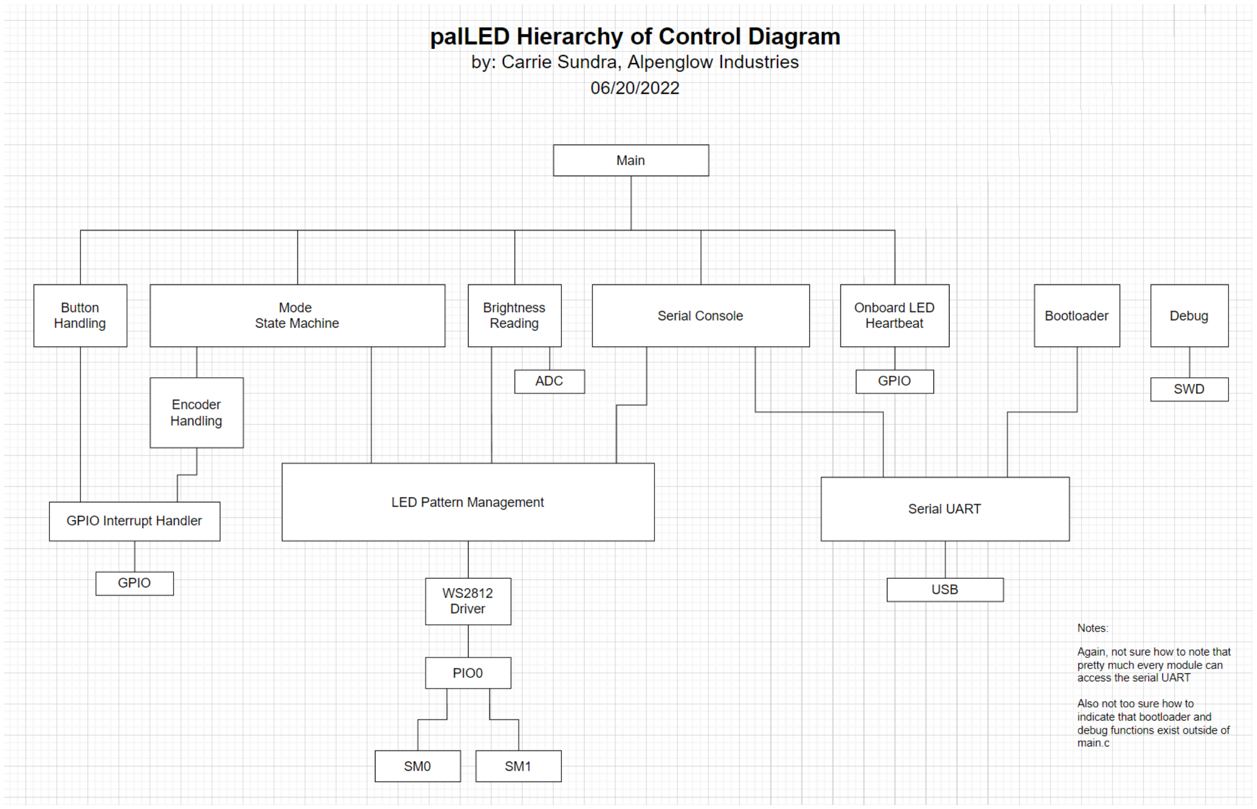


Notes:
I'm not sure how to indicate general serial print commands that aren't part of the console interface. They tend to be used by all modules and are covered in one of the standard Pico #includes.

Eventually the Serial Console will need access to everything

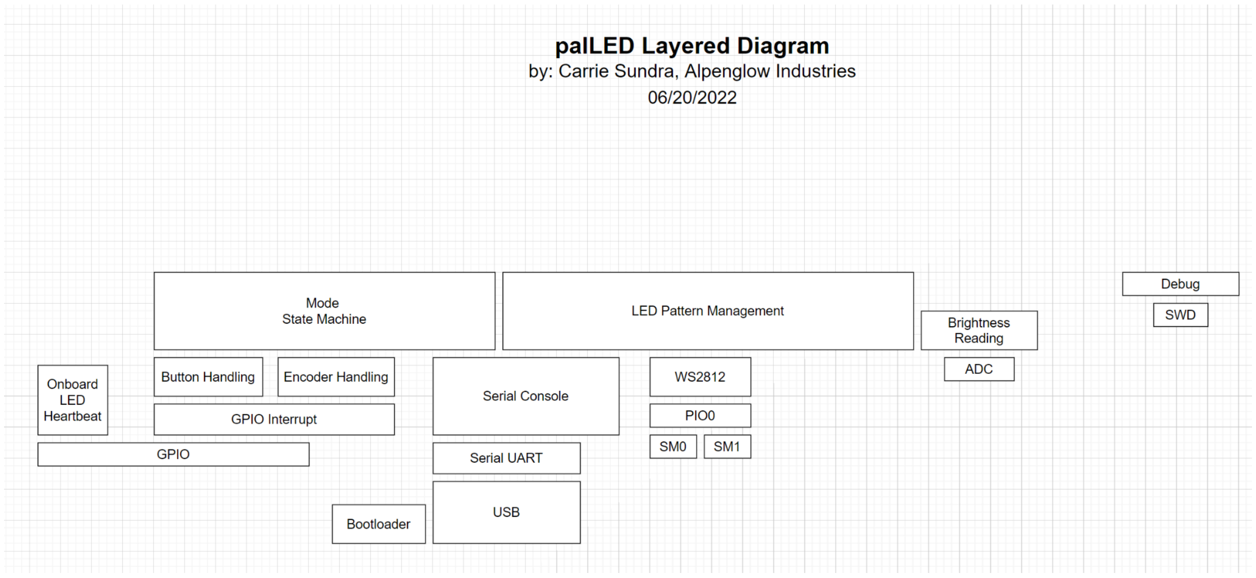
Hierarchy of Control

https://drive.google.com/file/d/1ACcmUYsDEGrGaeHEEaHpJOoxPsTgB8ZX/view?usp=sharing


palLED Hierarchy of Control Diagram
by: Carrie Sundra, Alpenglow Industries
06/20/2022
Main
Button Handling
Mode State Machine
Brightness Reading
Serial Console
Onboard LED Heartbeat
Bootloader
Debug
ADC
GPIO
SWD
Encoder Handling
GPIO Interrupt Handler
GPIO
LED Pattern Management
Serial UART
USB
WS2812 Driver
PIO0
SM0
SM1
Notes:
Again, not sure how to note that pretty much every module can access the serial UART
Also not too sure how to indicate that bootloader and debug functions exist outside of main.c

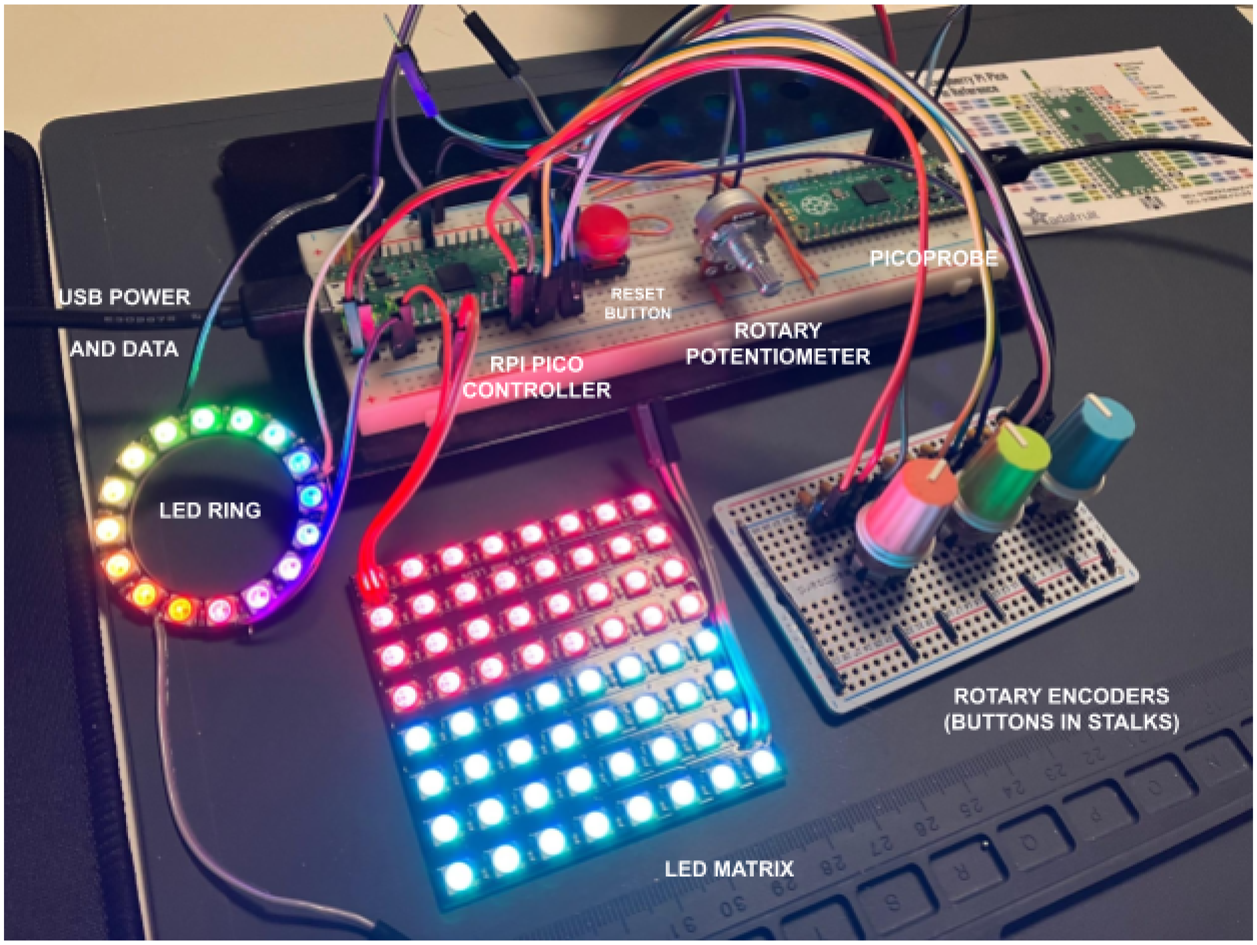
Layered Diagram

https://drive.google.com/file/d/1zXk3KvTLun8SN9UMwhKB0bu5IH0Dp5zT/view?usp=sharing



Build Instructions

Hardware:



A solderless breadboard was used for the Pico. A pre-existing LED matrix and ring were used, wires were simply soldered to them and they were plugged into the Pico's breadboard. The rotary potentiometer was also plugged straight into the breadboard. The encoders needed some debouncing and had a footprint incompatible with solderless breadboards, so a solderable breadboard was used to fix everything in place. Then jumpers were used to connect the encoder board to the Pico breadboard. Also on the Pico breadboard is an external button connected to the RUN pin on the Pico, allowing a button reset instead of having to unplug/replug USB. A second Pico is also on the breadboard, which can be used as a Picoprobe debugger. They are currently operating from 2 different USB ports, so the only difference between the setup in Getting Started with the Raspberry Pi Pico is that power is not connected together.

Software:

The OS used was Windows. The development environment used was VSCode with Platformio, and the Wizio platform plugin, using the Baremetal framework. This was chosen to minimize toolchain setup, and to avoid having to compile anything for Windows. I also wanted to learn VSCode and Platformio in order to have experience with an IDE that could work with many different microcontroller architectures as well as Arduino ports. The official RPi platform for Platformio only supplied the Arduino framework, and while this does include the RPi SDK and one can directly use SDK functions and external libraries within the setup() loop() structure of Arduino, it's nicer to avoid the Arduino overhead and initial setup. Wizio allowed this and also had some nice built-in features allowing reprogramming of the Pico to happen entirely with one click. Otherwise, the Pico uses a UF2 bootloader and the Arduino framework requires manually setting the Pico to bootloader mode and dragging and dropping a file every time one wants to reprogram the Pico, which a terrible workflow for development. More specific instructions on my setup can be found in the readme here:
https://github.com/casundra/MakingEmbeddedSystemsClass/tree/main/Week3-Homework/RpiPico-Wizio

Debugging:

I did not use the hardware debugger much. I'm still largely unfamiliar with them and am learning how to use them. I also didn't run into any issues that couldn't easily be fixed with printf debugging. Mostly it was testing algorithms for color space transformations, which requires the output of a large chunk of data. I tested the system in pieces as each bit of functionality was developed. There are still some test functions in the different libraries that can be used to verify a specific function, like outputting a serial "Hello World" heartbeat, or periodically printing the brightness setting.

Power:

The system was entirely powered off of USB and runs through the Pico's USB connector. USB power is the intention of the final system as well, although it will need a supply capable of 2A if high brightness is needed and/or multiple strips with many LEDs are used. In the future, power will be separated from the Pico's USB connector so that all LED power doesn't have to run through the Pico board.

Future (The Plan)

The final vision for this project is more ambitious than the proof-of-concept shown in this report. Even the minimum viable product needs a bit of clean-up and enhancements. The code currently has various “To Do:” comments that are mostly about potentially reducing cycles and organizing information differently to be more efficient and intuitive.

MVP (mini-palLED?)

The MVP will add an LCD that will display color and mode information on the device itself. The following will be evaluated:

For Pico, plug-in, via Waveshare:

- 1.8 inch TFT ST7735S driver
- 2.0 inch IPS ST7789VW driver

ST7735S driver on GitHub, and another one, and another one

Wizio ST7789 driver on GitHub

Generic form factor, via AliExpress:

- 1.77 inch TFT ST7735
- 1.8 inch TFT ST7735
- 2.4 inch TFT ST7789V

Looks like there’s decent support for both the ST7789 and ST7735 drivers, will start with Waveshare ones since a Pico plugs right into them. Leaning towards using a TFT because they cost a little less than OLEDs.

The device implementation will only be a “Color Picker” using 2 modes

- RGB mode - encoders control R, G, and B values and the rotary pot controls overall brightness. This mode is useful for reproducing a specific color if RGB values are already known. However, it is not intuitive in terms of being able to “dial in” a color using the color wheel.
- HSL mode - encoders control H, S, and L values and the rotary pot controls overall brightness. This mode is more intuitive for choosing a color if one isn’t already predetermined, as you can slide a color around the color wheel and move the color more towards an adjacent color to “dial it in”.

The user can adjust the following settings, to get a better understanding of how different brightness levels and corrections affect the LED color:

- Gamma correction (brightness): on and off
- Color wheel display: RYB or RGB
- Auxiliary strip type: WS2812, other protocols, number of LEDs in the aux strip

The serial console will be expanded to be able to display a variety of data as well as change settings of the device. It will have the ability to:

- Display active color values whenever it is changed
- Display color values for each LED in each strip
- Change device settings (color wheel, gamma, aux strip type)

Color values will be both RGB and HSL.

The hardware will be minimal with no additional enclosure. It may have a custom PCB, but likely would use plug-in headers so people could supply their own RPi Pico.

The Full palLED

The full final product is a bit more ambitious and implements all the features that allow for more color scheme development based upon RYB color theory. It would incorporate all of the above plus the following.

The modes would be the following, for more information on what they mean, see Appendix A. Original state table can be found here: https://docs.google.com/spreadsheets/d/1t5VnAQuAprSvEfFaMDxI-ScO61hFJ46T6QLDlgbNkkQ/edit#gid=0

- Monochrome (MONO)
- Complementary (COMP)
- Split Complementary/Triad/Adjacent (SPCOMP)
- Split Complementary with Complement/Tetrad/Adjacent with Complement (SPCOMPCOMP)
- Dual Complementary/Tetrad (DUALCOMP)
- Gradient (GRAD)

The serial console would output more data about the color schemes, including how the auxiliary colors are related to the active color in terms of HSL.

There will be more mechanical kit parts, requiring some assembly. It will include:

- Laser-cut enclosure parts (3D models and drawings)
- Assembly Hardware (standoffs, screws, knobs, button caps)
- Custom PCB with RP2040, an LED matrix, an LED ring, all encoders, buttons, LCD, aux LED connectors, and USB port
- An on/off switch might be nice

- Diffuser materials/options
- Possible: analog PWM LEDs that show the same colors for comparison

We'll also need to develop:

- Test fixture and procedure for the PCBA
- Full documentation (all of the above sections of this report, updated, should suffice plus links to software and CAD files)
- A video or series of videos about it
- A marketing plan

The ability to use other addressable LED protocols on auxiliary connectors could be tricky if different pins were needed. Some strips are 4 pins, some are SPI, etc. Fortunately the RPi Pico is extremely flexible, and most peripherals can be mapped to most pins. Implementing a change in protocol may only be a change in the pin/function mapping. Of course, power is more difficult to change, so it may be that different connectors are required for the different strip interfaces.

Adding analog PWM RGB LEDs would be really cool - it would be great to be able to compare and contrast them with addressable LEDs set to the same R, G, and B values. But it takes a LOT more pins, 12 PWM outputs would be needed to drive only 4 LEDs with different colors. There is a specialized IC, the Lumissil IS31FL3242-QFLS4-TR, which is an I2C chip that is capable of driving 4 RGB LEDs. It's interesting, and it's not terribly priced at $1.19 at 100 qty, but it's always a bit nerve-wracking to design such a highly specialized no-substitions-available IC into a product. So perhaps it could be an add-on module, or another different version of the palLED, so if it became unavailable, it wouldn't completely destroy the ability to ship a product and generate income.

Grading Self-Assessment

Project meets minimum requirements: 3

My state machine is a little basic, but I greatly exceeded the number of interrupts required and implemented a couple of additional commands on the serial console. I have the 3 peripherals needed, a video, a link to my codebase, and this report.

Completeness of deliverables: 3

My code is very well commented, I used the final report document as a guide and made sure to address every point, and made a video showing how it all works which also talks about some of the challenges.

Clear intentions and working code: 2

My code works and I think my intentions are clear, but I’m not sure that I’m quite that “professionally polished”.

Reusing code: 3

I did discuss the licenses and how it relates to making this a shipping product. I included all original licenses and added links to parts of code that I wrote using tutorials even if there was no code copied.

Originality and scope of goals 2

I had a lot bigger plans and while I think this project meets the minimum requirements, the scope is less than I would have liked.

Bonus: Power analysis, firmware update, or system profiling: maybe 1

I ran out of time and didn’t do a power analysis or system profiling. I did note several places in the code comments where I thought I could make improvements. Maybe I get a point for firmware update, I did discuss it, but it’s built into the RP2040 so I feel like it’s cheating a little bit.

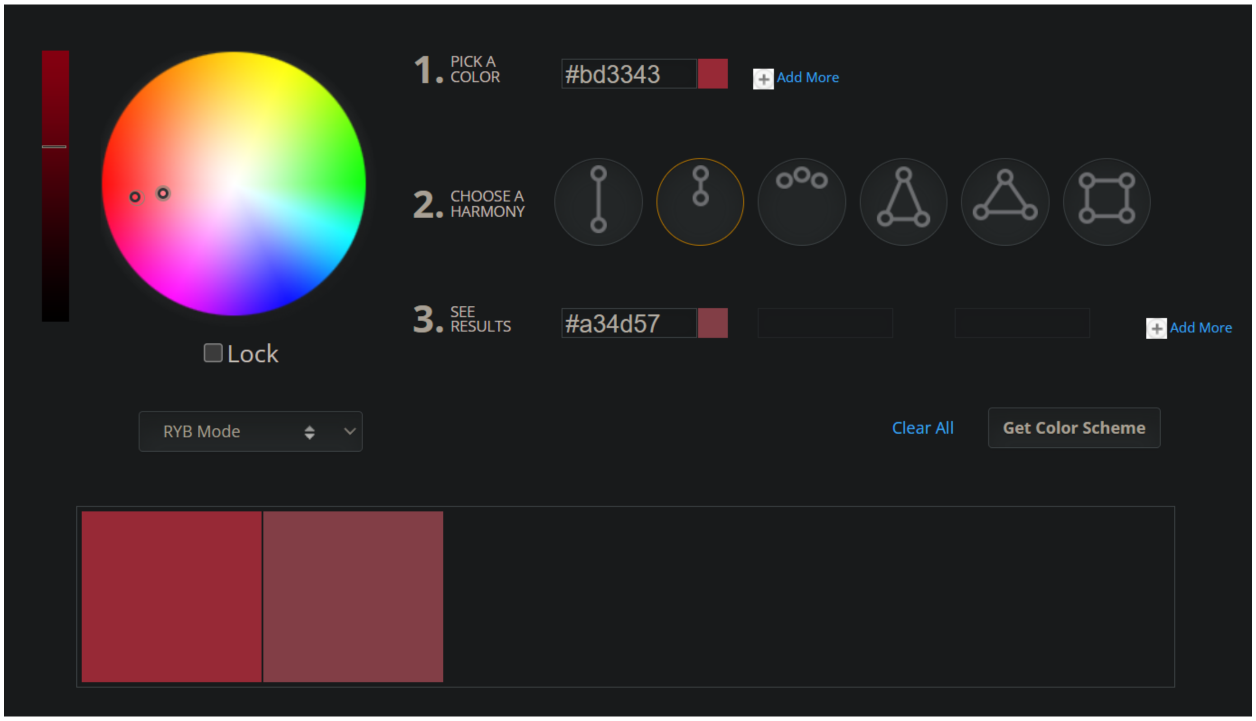
Bonus: Version control history showing development: 1

I’ve used GitHub from the start and I commit frequently and with vigor.

Appendix A

Screenshots are from: https://www.sessions.edu/color-calculator/

Monochrome (MONO)



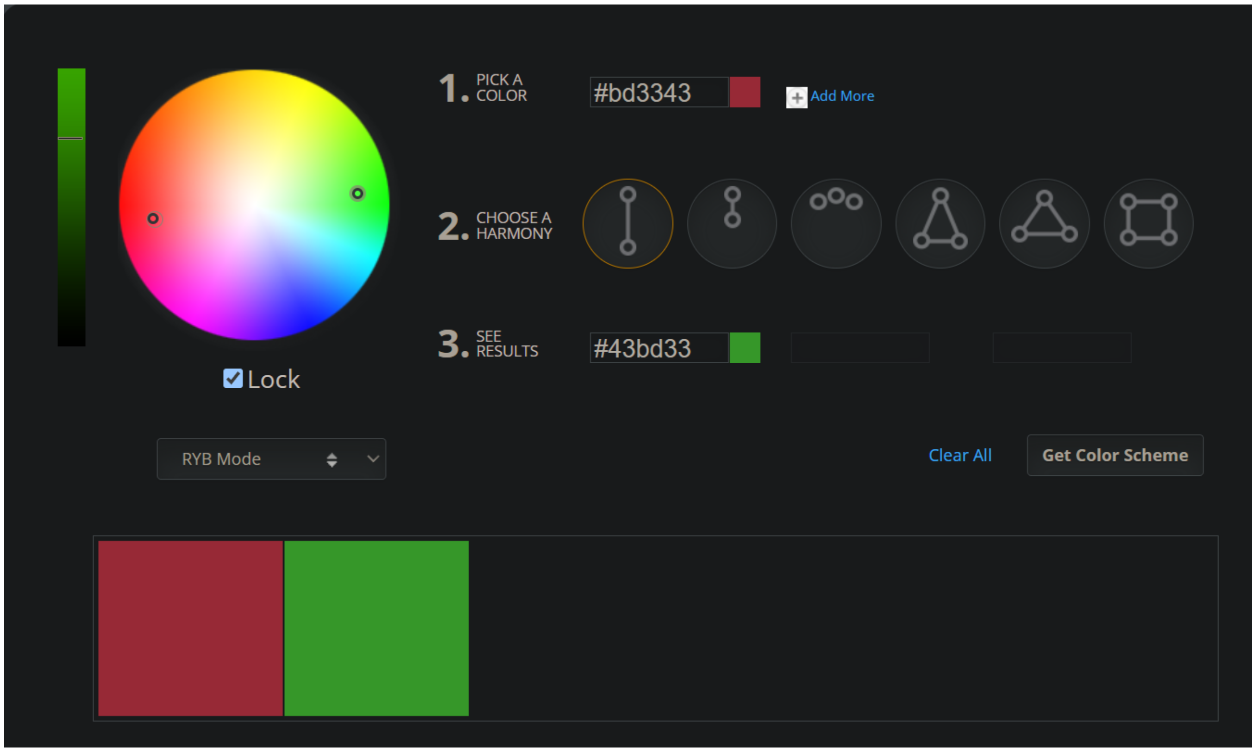
1. PICK A COLOR #bd3343

2. CHOOSE A HARMONY

3. SEE RESULTS #a34d57

A single color (the main color) is displayed on the matrix, with varying brightnesses of that color displayed below it. These accent colors are tints & shades of the main color. The main color can be changed which rotates it around the wheel. The range of monochrome tints can be adjusted to show brightnesses close to or far away from the main color.

Complementary (COMP)

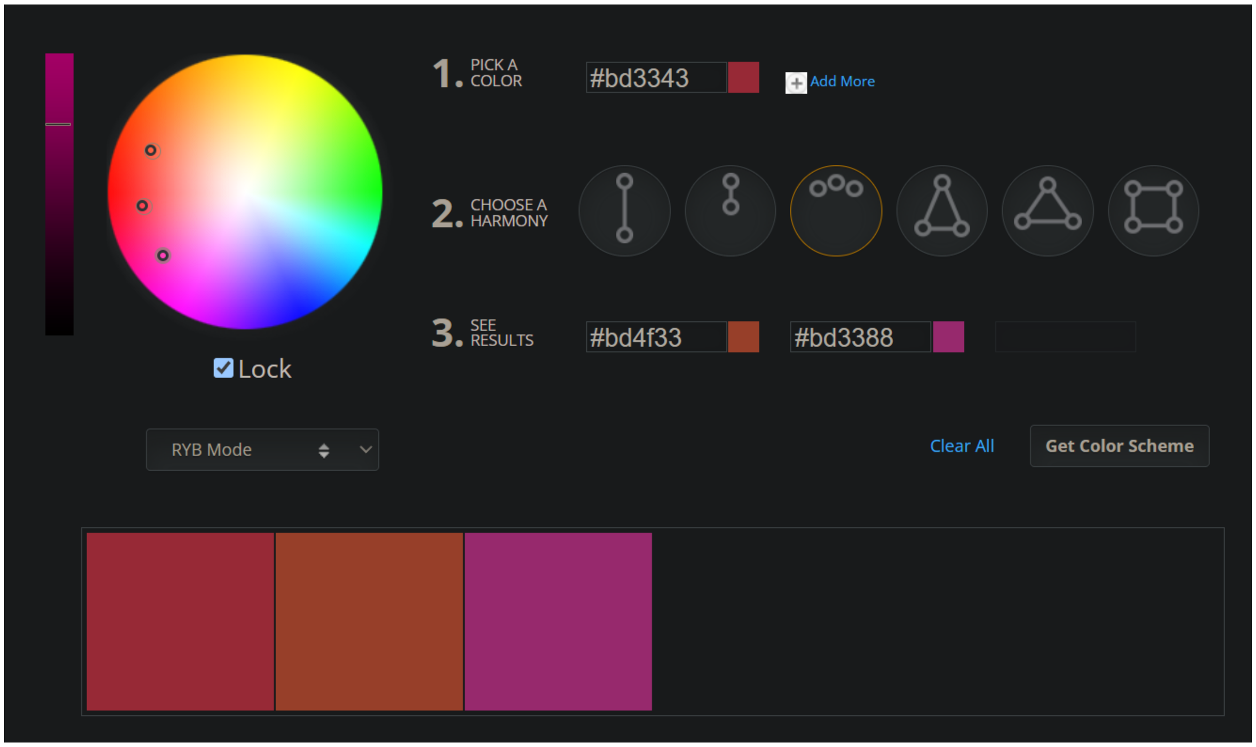


The main color and its complement (180 degrees opposite on the color wheel) are displayed on the matrix, and both have their monochrome tints displayed below.

For this scheme and all below, the main color can be changed which rotates both colors around the wheel. The range of monochrome tints can be adjusted to show brightnesses close to the main color, or farther away from the main color. This persists for all modes except Gradient.

Split Complementary/Triad/Adjacent (SPCOMP)

Adjacent:



1. PICK A COLOR #bd3343 Add More

2. CHOOSE A HARMONY

3. SEE RESULTS #bd4f33 #bd3388

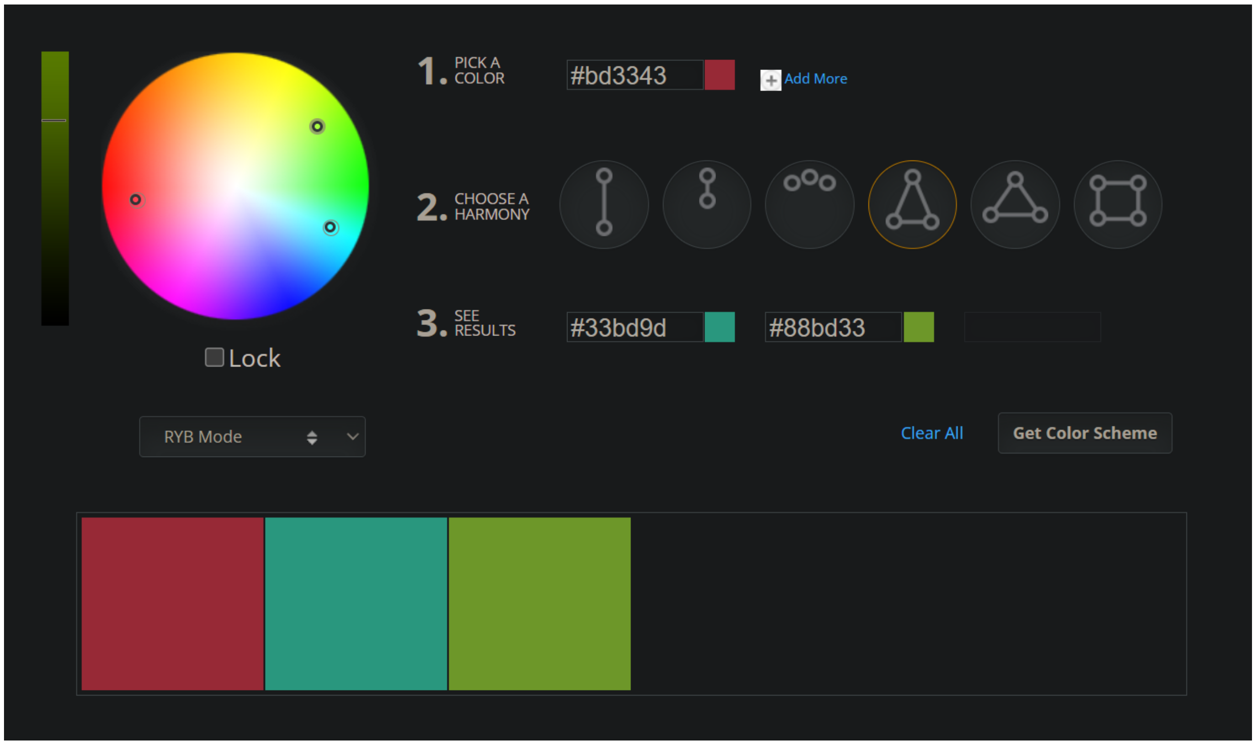
☑ Lock

RYB Mode

Clear All

Get Color Scheme

Split Complementary:



1. PICK A COLOR #bd3343 Add More

2. CHOOSE A HARMONY

3. SEE RESULTS #33bd9d #88bd33

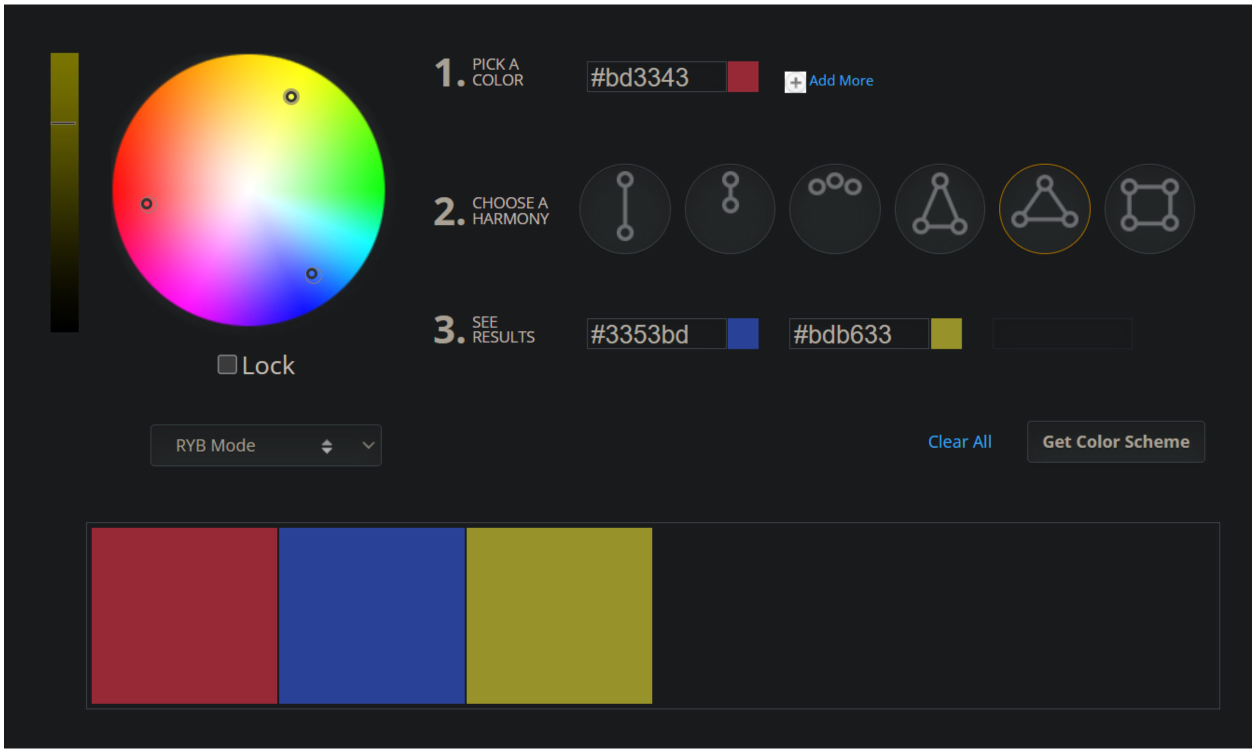
☐ Lock

RYB Mode

Clear All

Get Color Scheme

Triad:



#bd3343

#3353bd

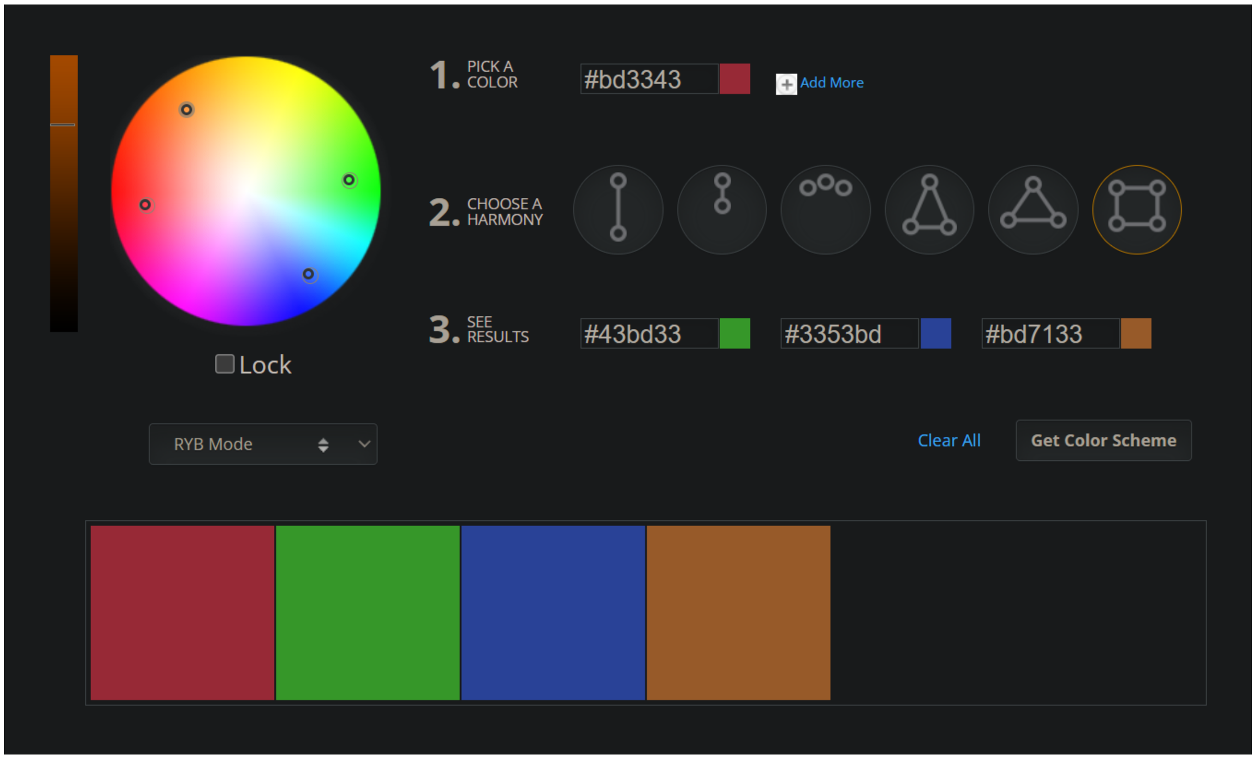
#bdb633

The main color and two colors at even angles to the right and left of the complementary color on the wheel are displayed. All 3 have their monochrome tints displayed below them. The angle of the accent colors can be changed, which moves them from split complements (under 60 degrees from the complement) to a triad (exactly 120 degrees apart, even spacing of all colors around the wheel) to adjacent (under 60 degrees from the main color).

Split Complementary with Complement/Tetrad/Adjacent with Complement (SPCOMPCOMP)

Same as the previous color scheme, but with the addition of the complementary color, seen in complementary above. All 4 have their monochrome tints displayed below them. The angle of the split complements can be changed, which moves them from split complements (under 60 degrees from the complement) to a terad (exactly 90 degrees apart, even spacing of all colors around the wheel) to adjacent with complement (under 60 degrees from the main color).

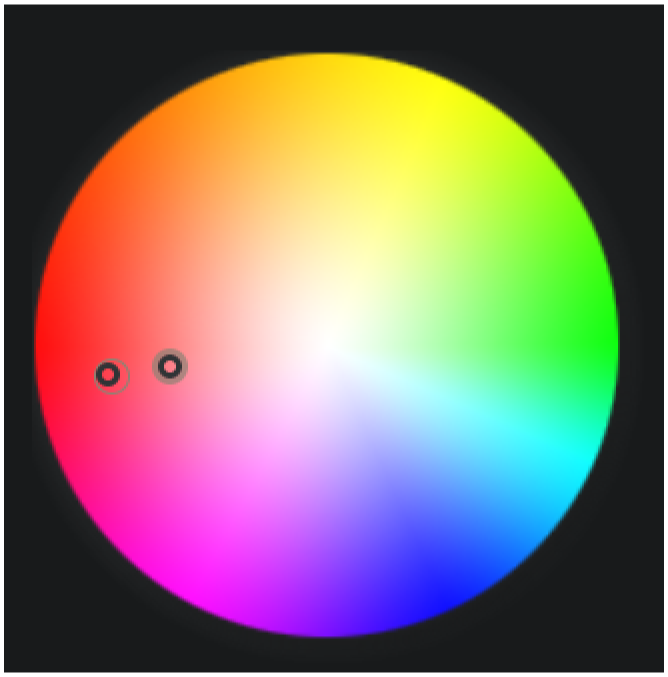
Dual Complementary/Tetrad (DUALCOMP)



Two pairs of complementary colors. All 4 have their monochrome tints displayed below them. Changing the main color moves all colors around the color wheel. The angle between the pairs can also be changed, moving the second pair of complements around the color wheel, closer to or farther away from the first pair. When that angle is 90 degrees, a tetrad is again formed.

Gradient (GRAD)

The colors fade into each other in the color wheel below in gradients:



This is not really a color theory scheme, but it’s useful for LED color scheme generation. A gradient has two different colors as endpoints, and gradually fades one end to the other, through a third intermediate color (usually the shortest distance between the two colors on the color wheel). For example, a yellow to purple gradient goes through an orange/red. A green to blue gradient goes through a teal. Changing the main color moves both endpoints through the color wheel. Changing the angle moves the endpoint of the second color. A middle color is automatically chosen, but the balance between the two colors can be adjusted, which skews the gradient toward one end or the other. This is displayed on the matrix at a 45 degree angle.