Firmware - RGB Color Mixing Firmware for EZ-Color™

AN16035

Author: Ben Kropf

Associated Project: Yes
Associated Part Family: CY8CLED16, CY8CLED08, CY8CLED04
GET FREE SAMPLES HERE
Software Version: PSoC Designer™ 4.4
Associated Application Notes: AN14406, AN15733, AN33640
PSoC Application Notes Index

Application Note Abstract

The Cypress EZ-Color™ device is well suited for High Brightness LED (HBLED) color mixing applications. However, the firmware in these applications is quite complex. This application note details firmware that implements an RGB LED color mixing solution in the EZ-Color HBLED controller.

Introduction

Using EZ-Color™ to intelligently drive and control high powered LEDs leads to useful applications and designs. The firmware in these applications is extensive and complex. This application note describes the firmware in an application that features dynamically programmable color outputs, a high level of integration, and minimal use of system Flash. The firmware implementation of a RGB LED color mixing application is also described. A considerable amount of external hardware and thermal design must accompany the firmware. Cypress has produced other application notes that address both of these aspects of HBLED color mixing designs.

From a high level perspective, this firmware inputs values in CIE 1931 chromaticity coordinate form, and converts the coordinates into the appropriate dimming values for each of the three LED channels. A dimming value is the percentage of maximum luminous flux to which an LED must be dimmed. If an LED has its current quickly switched on and off in an intelligent fashion, the LED has its flux output controlled.

In other words, one chromaticity coordinate is input into the firmware. The firmware combines this coordinate with its preprogrammed knowledge of the LED characteristics in the system, along with LED junction temperature measurements. It then completes the necessary transfer function that correctly converts the chromaticity coordinate into a dimming value for each LED. This process enables their light outputs to mix together to create the color of the chromaticity coordinate input into the system.

Accompanying Hardware and Software

The firmware described in this application note is included with the CY3261A-RGB demonstration board kit available in the Cypress Online Store. This kit is specifically developed for the hardware circuitry that is exhibited on the board. The kit also includes a PC software application that controls the color output produced by the board.

High Level Firmware

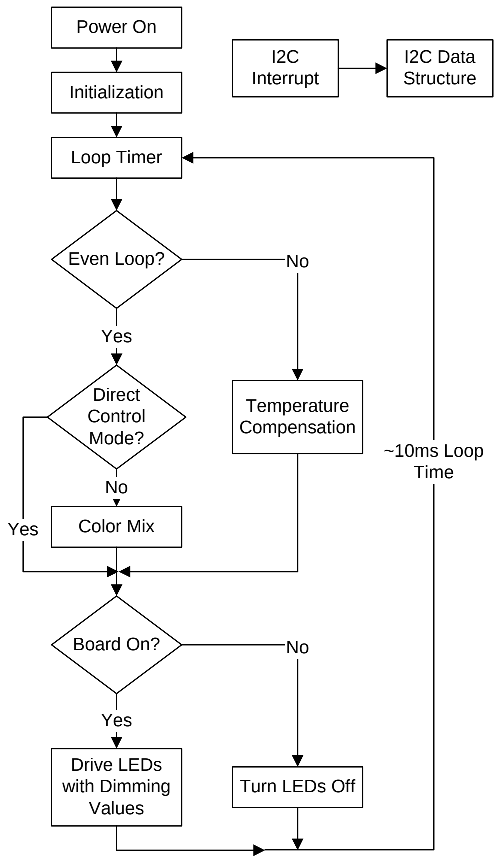
This firmware repeatedly runs through a loop that performs four tasks (refer to Figure 1 on page 2). The chart in this figure represents the procedural execution of the `main()` loop in the associated project. The sections of code shown in Figure 1 are apparent in `main()`.

The two most complex and important tasks are Color Mix and Temperature Compensation. These tasks also take more time to execute. Therefore, the firmware is set up to alternate the task executed during each loop. The other two primary tasks in the loop are much simpler. The Loop Timer task delays program execution so that the time taken to execute the entire main loop is approximately 10 ms. The Loop Timer task keeps the entire program synchronized and orderly. The Drive LEDs with Dimming Values task is responsible for updating the LED driver hardware with the correct values to dim the LEDs.

Although it is not a procedural task, this firmware acts as an endpoint on an I^2C™ network. The firmware receives commands to display mixed colors through this communication protocol. I^2C communication with the master device is interrupt driven. Therefore, the main program execution is not affected. Whenever the I^2C master sends data to this microcontroller, the data is placed in a data structure held in RAM. When the tasks in the main loop, such as Color Mix, are executed, they retrieve and use values from this data structure. This creates a system where the program in the main loop constantly accesses and uses values from RAM, which is updated periodically in the background by the I^2C network master.

The Initialization task is not part of the repetitive main loop. Instead, it occurs once after boot up. The Initialization task is responsible for applying all initialization settings that enable the rest of the firmware to function correctly.

Figure 1. High Level Flow Chart of Firmware Execution



Color Mixing Inputs

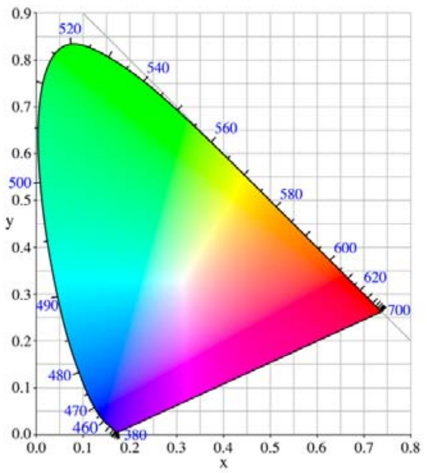
Before exploring the Color Mix or Temperature Compensation tasks, you must know more about the inputs and outputs of the system.

This firmware uses the CIE 1931 color space to input color requests to be serviced. Any particular color in the CIE 1931 color space is represented with three values, which form a vector (x, y, Y). The x and y values represent the color hue and saturation. This application note refers to the x and y values together as the (x, y) coordinate. Plotting the (x, y) coordinate on the chart in Figure 2 obtains a particular shade of color. The colored area represents all visible colors of light, and the white area represents colors that are not visible to the human eye. For example, an (x, y) coordinate of (0.7, 0.7) is not in the colored area and does not represent any visible color.

The third value of the (x, y, Y) vector specifies the luminous flux, in lumens (lm). While the (x, y) coordinate is dimensionless, the Y value can have units of lumens (lm), or it is expressed as a percentage to signify a relative flux. The Y value cannot be seen in the graph of Figure 2, but it is visualized as a vector normal to the page with a magnitude of Y at some (x, y) coordinate. This (x, y, Y) vector completely describes a light source by denoting its color and its total flux. The firmware must have inputs in (x, y, Y) vector form. Any set of one or more LEDs can have an (x, y, Y) vector that specifies its average color and total flux output at some rated current and junction temperature (for example, T_J = 25°C and IF = 350 mA).

This firmware receives color requests in the form of three values. In this particular implementation, the (x, y) coordinate takes the form of two 16-bit words, where a value of 10,000 would correspond to an x or y value of 1.0. The Y value is input as an unsigned byte that specifies the number of total lumens the mixed color must have. In *main.c* of the example project, take note of the `wCurrentX`, `wCurrentY`, and `bFlux` variables in the `sLED_Data` data structure. The I^2C master is free to update these three values at any time. The Color Mix task can then use the values to determine the correct dimming values for the three LEDs that create the required (x, y, Y) color. This process is shown in Figure 2.

Figure 2. CIE 1931 Color Space Graph



Color Mixing Outputs

The challenge in developing this system lies in controlling three LEDs (with fixed (x, y) coordinates and rated lumens). The LEDs must be dimmed correctly to specific intensities, so that their individual colors can mix to create a requested color.

In this application, the hardware technology modules that control the three LEDs are precise illumination signal modulators (PrISMs). These PrISM modules are implemented in PSoC Designer with SSDM (stochastic signal density modulation) User Modules. The theory behind these modules is that the greater the signal density they have, the greater the flux of the LED that is being driven with it, and vice versa. Therefore, there is a linear relationship between an LED's flux and the density of the signal that is driving it. Each LED has a maximum lumens output when it is driven with 100% signal density. The firmware must use this important maximum lumens output parameter. For example, an LED may be specified to have 44 lm of light output when driven with 350 mA of current. If the dimming signal density is 50%, expect 22 lm of light out of that LED. The following equation shows this relationship.

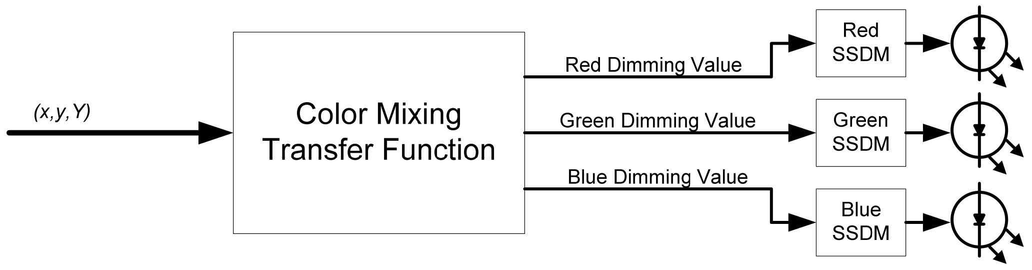
$$Y_{out} = Y_{\max} \times SignalDensity \qquad (1)$$

In Equation 1, Y_{max} is the lumens an LED would have if it had a constant rated current flowing through it. The *SignalDensity* term is called the "dimming value." It is a value from 0.0 to 1.0 (that is, a percentage), which is used to linearly dim an LED down from its maximum flux. The PrISM hardware available in EZ-Color has a selectable bit resolution. This application's PrISM hardware modules each have a resolution of 8 bits, and so they must have an 8-bit input value that determines what their signal density output is. Therefore, the previous equation is rewritten as seen in Equation 2.

$$Y_{out} = Y_{\max} \frac{n}{2^N - 1} \qquad (2)$$

In Equation 2, *N* is the resolution of the PrISM hardware, and *n* is an *N*-bit value. These equations only determine the lumen outputs of the LEDs; the respective *(x, y)* coordinates of each LED stays relatively fixed. Figure 3 shows the inputs and outputs of this firmware. Everything shown in this figure is internal to the EZ-Color except for the actual LEDs. The color mixer block is a mathematical function executed by the EZ-Color's CPU, and the PrISM modules are internal hardware in the EZ-Color device.

Figure 3. Firmware Inputs and Outputs

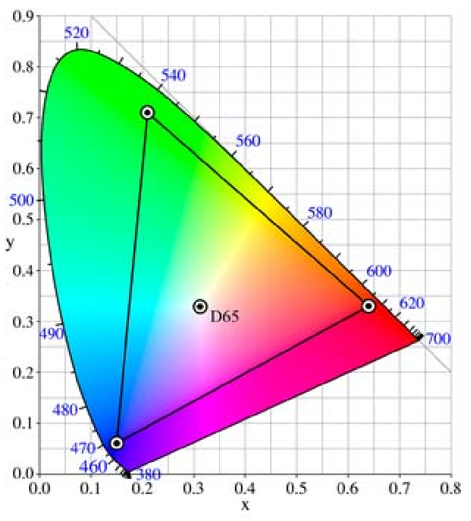


Dynamic Color Mixing

The final challenge for the firmware is converting the CIE 1931 color coordinates into three dimming values for the respective red, green, and blue LED-driving channels. There are several methods to carry out this process. This firmware is one approach to implement the conversion.

The three dimming values for each mixed color are not stored in a look up table in the Flash memory. This is referred to as the "static method." The process shown in Figure 3 is referred to as the "dynamic method." This method has several advantages that are not addressed by the static method.

Figure 4. Achievable Color Gamut Triangle



First, the dynamic method provides a dynamic range of color inputs. In the chart shown in Figure 4 on page 3, the triangle represents the color gamut that is an arbitrary set of red, green, and blue LEDs. The three points of the triangle are the *(x, y)* coordinates of each respective LED. The area inside the triangle is the gamut of achievable colors with this particular set of three LEDs. Any *(x, y)* coordinate within the triangle can be input into the system, providing a broad range and high resolution of unique colors that are produced with this system. If dimming values are pre-stored in the Flash memory, the system is limited to the colors that have been predefined in the Flash or EEPROM memory.

Second, the dynamic method enables the use of feedback. The characteristics of LEDs such as color coordinates, lumen output, and forward voltage change with the junction temperature and lifetime of the LEDs. Therefore, this method makes it possible to feed back junction temperature or lifetime measurements into the system. You could then dynamically adjust the base *(x, y)* coordinates and lumen ratings of the LEDs, compensating for environmental changes like ambient temperature. If the static method of pre-storing dimming values into the Flash memory is used, feedback of any sort generally becomes unfeasible.

Finally, a third advantage is that the system becomes simpler from a high level standpoint. The internal complexity of the dynamic method is greater than the static method, but being able to define desired mixed colors in CIE 1931 coordinates is simpler and more user-friendly than defining colors with dimming values.

This system implements a fully dynamic method of color mixing. All the input variables of the system are exposed to an I^2C master that can dynamically update any of the inputs. This causes the EZ-Color device’s firmware to respond accordingly.

Color Mixing

Figure 3 shows the inputs of the firmware and the translated outputs. The mathematical functions in this section describe how the three dimming values are obtained from one *(x, y, Y)* coordinate.

The first step is the creation of a matrix, as shown in Equation 3. The color subscript (for example, red) denotes the *x* or *y* value of the respective red, green, or blue LEDs in the system. The “mix” subscript denotes the *x* or *y* value of the input color coordinate request.

$$A = \begin{bmatrix} \frac{x_{red} - x_{mix}}{y_{mix}} & \frac{x_{green} - x_{mix}}{y_{mix}} & \frac{x_{blue} - x_{mix}}{y_{mix}} \\ \frac{y_{red} - y_{mix}}{y_{mix}} & \frac{y_{green} - y_{mix}}{y_{mix}} & \frac{y_{blue} - y_{mix}}{y_{mix}} \\ 1 & 1 & 1 \end{bmatrix} \qquad (3)$$

The first mathematical operation is taking an inverse of the previous matrix, as seen in Equation 4.

$$A' = A^{-1} \qquad (4)$$

After the matrix inversion, the LED dimming data is solved and is located in the matrix elements a'_{02}, a'_{12}, and a'_{22}. The next step is to factor in the total flux information of that color. This is done by a matrix multiplication as seen in Equation 5.

$$\begin{bmatrix} Y_{red} \\ Y_{green} \\ Y_{blue} \end{bmatrix} = A' * \begin{bmatrix} 0 \\ 0 \\ Y_{mix} \end{bmatrix} \qquad (5)$$

The value of Y_{mix} is the number of lumens that the total mixed light output must produce. The resultant *Y* values of the product are the lumen output of each respective LED that is necessary to create the requested color and flux. At this point all the math operations give rise to two benefits. If any of the final product’s *Y* values in Equation 5 are negative, it signifies that the requested color coordinate is invalid and the LEDs in the system cannot create that color. In other words, the requested color is outside of the gamut of the LEDs. The second item to check is if any of the product’s *Y* values are larger than the maximum lumen output of any of the three LEDs. If this is the case, then it means that the Y_{mix} input is too large, and the LEDs in the system cannot create that much total flux at the given *(x, y)* coordinate. The firmware checks to see if either of these conditions occurs. If the *(x, y)* coordinate is invalid, the firmware turns the LEDs off and reports an error. If the requested flux is too large, the firmware scales back the values so that they produce the maximum possible flux at the requested *(x, y)* coordinate.

$$DimValue_{red} = \frac{Y_{red}}{Y_{\max,red}} * \left(2^N - 1\right) \qquad (6)$$

Equation 6 expresses how a dimming value is produced from the Y_{red} value (the exact same equation would also apply to the other colors). $Y_{max,red}$ is the lumens that the red LED has if it is not dimmed at all, which is its maximum flux. *N* is the number of bits of resolution that the hardware dimmers have. In this system, *N* is equal to 8. After applying this equation to each color channel, each channel has a unique dimming value that is applied to the PrISM hardware LED dimmers.

The Color Mix task from Figure 1 on page 2 does the math described in this section, which produces the dimming values from the color vector input. Many parts of the math are optimized to execute faster and use less code space by not performing unnecessary operations. For instance, most of the operations of Equation 5 are unnecessary because they always multiply by zero. As a result, this part of the math is not executed in firmware.

Figure 5 on page 5 shows the block diagram of the Color Mix task. It shows how all the data variables stored in the RAM factor into the math equations of the task.

Temperature Compensation

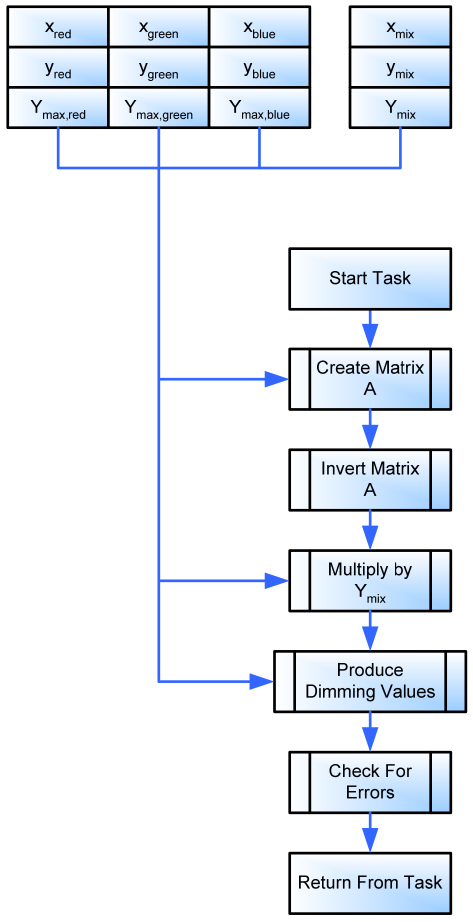
The forward voltage, color, and flux of an HBLED can significantly change as a function of its junction temperature (T_J). T_J is highly dependent on uncontrollable environmental factors, such as ambient temperature. For this reason, it is advantageous to monitor T_J so that variations in the LED characteristics are compensated for. The theory behind this process is made possible by the following equation:

$$T_J = T_B + \theta_{JB} * P_D \quad (7)$$

Equation 7 shows the relationship between the measured temperature of the circuit board (T_B) and the junction temperatures of the LEDs. θ_{JB} is the thermal resistance between the junction of an LED and the board. A good approximation of this value is obtained by using the thermal resistance value from an LED data sheet. P_D is the power being dissipated by the LED. This is shown in Equation 8.

$$P_D = V_F I_F (Dim\%) \quad (8)$$

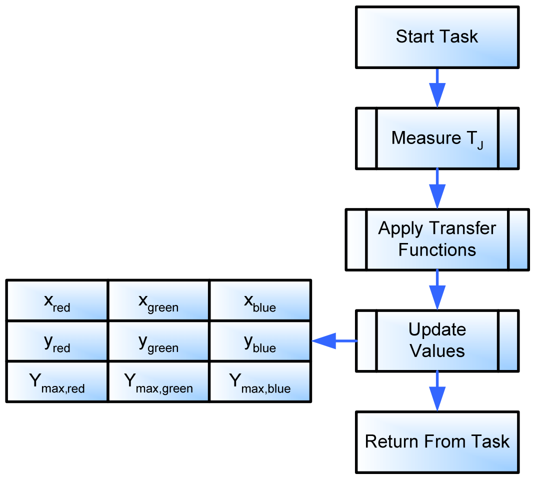
Figure 5. Color Mixing Process Flow Chart



The values for VF and IF are either measured directly with an analog-to-digital converter (ADC) on the EZ-Color, or static values are used because VF only changes slightly and IF is typically held constant by the LED driver. The products of each V_F and I_F value is then multiplied with the respective dimming value of each LED, this produces the average power dissipation of each LED. These values can then be used in Equation 7 to produce TJ for each LED. After this has been determined, new values for the maximum flux and *(x, y)* coordinate variables for each LED is updated.

Figure 6 details the flow of the Temperature Compensation task from Figure 1 on page 2. It shows how the task determines the T_J of each LED, applies transfer functions to determine how LED characteristics change, and updates the RAM variables to reflect these changes.

Figure 6. Temperature Compensation Task Flow Chart



Final Tasks

The final two primary tasks seen in Figure 1 on page 2 are the Loop Timer and the Drive LEDs with Dimming Values tasks. The Loop Timer task is a variable clock resource available to the EZ-Color (called VC3) and can generate interrupts with every rising edge. These interrupts decrement a variable, which creates a down counter in the firmware. The Loop Timer task delays until the down counter reaches zero, at which point it resets the counter to an appropriate value. This timing system is set up so that the main loop takes roughly 10 ms to execute. Any execution path through the main loop must not exceed the loop time of 10 ms.

The Drive LEDs with Dimming Values task is implemented by the PrISM hardware on the EZ-Color device. The PrISM channels have API functions that make it easy to apply the dimming values to produce correct signal densities. After the dimming values are determined by the rest of the firmware, it is easy to apply these values to the PrISM modules. The hardware then modulates its signal density output to control the LED current-driving circuitry to create the desired flux.

LED Bin Information

The challenge that HBLEDs present in a design is their variance in characteristics from part to part. Therefore, LED manufacturers have devised systems of codes that denote the characteristics of a given lot of LEDs. These codes are called “bin codes,” and typically there exists a code for light wavelength (color), forward voltage, and luminous flux. There is little standardization between manufacturers regarding bin code systems. As a result, each brand of LED may have its own unique system of bin codes.

This firmware was developed using Luxeon® K2 LEDs made by Lumileds™. It is necessary to create a file to store the bin information and make it available at compile time. In this project, this file is called *bin_tables.h*. It contains a large amount of preprocessor code that defines the correct LED characteristic values in the executable firmware, based upon what bin codes the LEDs have that are in the system. The different codes in the *bin_tables.h* file follow the bin code system that Lumileds LEDs use.

Code 1

```
#define       R_LUMEN_BIN   ('R')
#define       G_LUMEN_BIN   ('R')
#define       B_LUMEN_BIN   ('P')
```

Code 1 shows the location of the flux bin code in *bin_tables.h.* Note that in the code the red and green LEDs have their luminance bin codes set to ‘R’, and the blue LED has its bin code set to ‘P’. The ‘R’ and ‘P’ bin code values make sense in the context of the Lumileds Luxeon K2 documentation. After the bin codes in Code 1 are changed, the project is rebuilt for the changes to take effect. Code 2 shows preprocessor commands. When the bin code is set to ‘R’, the commands define the `R_MAX_LUMENS` string with the value 11712. The Lumileds flux bin ‘R’ means that the LED produces 45.75 lumens on average at a rated current of 350 mA. The value of 11712 is 45.75 multiplied by a scaling factor of 256 so that it is not a floating point number. After the definition is made, the value is used in the executable firmware code for calculations that require the maximum rated lumens of an LED. It must be noted that setting up the *bin_tables.h* file is not very difficult, because it is often just a transcription of data from a manufacturer’s documentation.

Code 2

```
#if(R_LUMEN_BIN == 'P')
       #define       R_MAX_LUMENS  ( 6925 )
#endif
#if(R_LUMEN_BIN == 'Q')
       #define       R_MAX_LUMENS  ( 9011 )
#endif
#if(R_LUMEN_BIN == 'R')
       #define       R_MAX_LUMENS  ( 11712 )
#endif
```

Conclusion

This application note provides an overview of the different firmware processes that take place in designing RGB LED color mixing using an EZ-Color device. To get a complete look at this system, refer to other associated application notes. To gain a better understanding of the code, the project associated with this application note is examined and thoroughly commented. To see this project in action with hardware, use the CY3261A-RGB kit.

About the Author

Name: Ben Kropf

Title: Applications Engineer

Background: Past studies include electrical engineering at Seattle Pacific University. Has been working on using PSoC and EZ-Color in many different applications. Ben enjoys the fact that mixed-signal arrays are used for such a wide variety of purposes.

Contact: btk@cypress.com
425.787.4867

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone: 408-943-2600
Fax: 408-943-4730
http://www.cypress.com/