THIS SPEC IS OBSOLETE

Spec No: 001-51188

Spec Title: MULTI CHANNEL COLOR MIXING USING HB LEDS - AN51188

Sunset Owner: Sree Harsha Angara [sreh]

Replaced by: NONE

Multi Channel Color Mixing Using HB LEDs

AN51188

Author: Anshul Gulati

Associated Project: Yes
Associated Part Family: CY8CLED04, CY8CLED08, CY8CLED16
Software Version: PSoC Designer™ 5.0
Associated Application Notes: AN16035

Application Note Abstract

High Brightness (HB) LEDs are one of the emerging solutions for lighting. Cypress EZ-Color^{TM} solution combines powerful color mixing functionalities with LED modulating capabilities in one device. This application note discusses the basics of four-channel color mixing for LED dimming and implementation using EZ-Color devices.

Introduction

Cypress's EZ-Color devices are used to intelligently control HB LEDs, resulting in useful color mixing and tunable white light applications. This application note explains the mathematical operations involved in multi channel color mixing algorithms and their implementation using EZ-Color. The firmware implementation of four-channel color mixing is explained in detail.

The firmware inputs values in CIE 1931 chromaticity coordinate form. These values are converted into appropriate dimming values for all four LED channels resulting in the desired color.

EZ-Color HB LED controllers are not limited to RGB or RGBA systems. Many combinations of LED colors can be mixed intelligently. For instance, two white LEDs and an amber LED can form a WWA system that creates different shades of warm white color. Similarly, one red, one blue, and two green LEDs can form an RGGB system; this combination can be used to cover the entire black body locus. This enables the system to produce tunable white light varying from warm white to cool white.

This application note gives an insight into complete lighting system design with Cypress EZ-Color.

Assumptions

This document assumes that the reader is familiar with the three-channel color mixing algorithm. Refer to application note 'AN16035, *Firmware – RGB Color Mixing Firmware for EZ-Color^{TM}*'. Prior knowledge and understanding of EZ–Color device architecture, C programming language, and switching regulator design is also useful.

EZ-Color: Overview

Cypress' EZ-Color family of devices offers an ideal control solution for HB LED applications that require intelligent dimming control. EZ-Color devices combine the power and flexibility of PSoC® (Programmable System-on-Chip™) to provide lighting designers a fully customizable and integrated lighting solution platform.

EZ-Color's virtually limitless analog and digital customization allows simple integration of features in addition to intelligent lighting such as battery charging, image stabilization, and motor control.

Cypress offers various kits to understand the basic functionality of EZ-Color and create your own application. The CY3261-RGB is a HB LED color mix board; it is useful in understanding the three-channel color mix algorithm, temperature compensation, and so on. Another Cypress kit, CY3263-ColorLock, demonstrates the ability of EZ-Color device to take real time optical feedback and control the three HB LEDs.

For more information on EZ-Color solutions and associated technologies, visit http://www.cypress.com/ez-color/

Example Projects

The firmware described in this application note has been used in the development of example projects on the following kits.

- CY3261A-RGB EZ-Color Evaluation Kit
- CY3263-ColorLock Evaluation Kit
- CY3265N-RGB Evaluation Kit
- CY3265O-RGB Evaluation Kit
- CY3267 PowerPSoC Lighting Evaluation Kit
- CY3268 PowerPSoC Lighting Starter Kit
- CY3269N Lighting Starter Demonstration Kit

In the PowerPSoC kits, the three channel color mixing algorithm has been extended to four channel color mixing algorithm for use with the PowerPSoC devices. The same firmware as described in the application note has been extended for use with DMX enabled devices. An example reference design is,

Four-Channel PowerPSoC in Small Form Factor Designs

The firmware used in the above kits can be downloaded from the respective web pages or obtained by purchasing the kits.

Accompanying Software

The firmware described in this application note is developed using PSoC Designer™ 5.0. The latest version of this software tool is available for free download at http://www.cypress.com/psocdesigner/

Color Science and Multi Stimulus Space

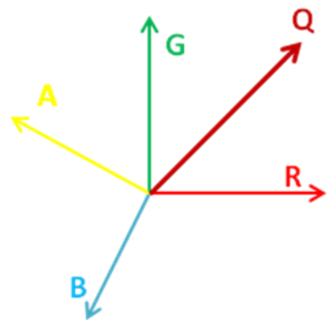
Similar to any other mathematical representations of physical phenomenon, color models can be expressed in different ways. Each has its advantages and drawbacks. The goal of modeling is to minimize formulation complexity and the number of variables while maximizing “substance” and breadth of coverage.

Historically, whatever the meaning assigned to the variables, three of them were enough to describe all colors: RGB, Hue-Saturation-Brightness (HSB), and other HS based models, such as L*a*b and xyY. One common feature was the number of variables or dimensions.

In multi-stimulus space, color stimuli are denoted by letters, such as Q, R, G, B, and A. Q represents an arbitrary color stimulus and the letters R, G, B, and A are reserved for fixed primary stimuli chosen for color matching experiments. The primary stimuli are Red, Green, Blue, and Amber. A color matching between a given stimulus Q and the additive mixture in suitable amounts of the fixed various primary stimuli R, G, B, and A can be expressed by the following vector equation:

$$Q = R_Q R + G_Q G + B_Q B + ... + A_Q A \qquad \text{Equation 1}$$

Figure 1. Multi Dimensional Color Space



In multi dimensional space, a color stimulus Q is represented by the multi-stimulus vector Q where the scalar multipliers R_Q, G_Q, B_Q, A_Q measured in terms of the assigned respective units of given primary stimuli R, G, B, and A respectively are called multi-stimulus values of Q.

The geometric representation of Equation 1 in linear multi-dimensional space is shown in Figure 1. The unit vectors R, G, B, and A represent the primary stimuli, defining the space. They have a common origin and point in four different directions. The vector Q has the same origin as R, G, B, and A. Its four components are located along the axes defined by R, G, B, and A, and have lengths respectively equal to R_Q, G_Q, B_Q, and A_Q, the multi-stimulus values of Q. The direction and length is obtained by simple vector equation defined by Equation 1. The space defined by R, G, B, and A is called multi-stimulus space. In this space, a color stimulus Q appears as a multi-stimulus vector (R_Q, G_Q, B_Q, and A_Q).

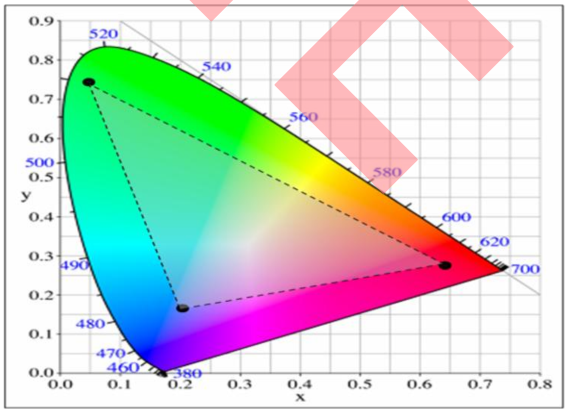
In color mixing algorithm, the firmware calculates what these values should be to derive the color stimulus Q. This is explained in the next section.

Color Mixing

Color mixing is essentially an experiment where secondary color is generated by adding the appropriate proportion of base primary colors. Primary colors are not a fundamental property of light but are often related to the psychophysical response of the eye to light. It is conceived that primary colors are completely independent from each other and sets of colors that can be combined to generate a useful range (gamut) of colors.

Figure 2 shows the CIE 1932 color chromaticity diagram. There are three LEDs, red, green, and blue plotted in the diagram. By mixing appropriate proportion of two primary colors such as red and green, all colors along the line which joins red and green can be generated. Color mixing these three LEDs can generate any color that lies within this triangle. This area is called the color gamut. However, in the CIE 1931 standard the color distribution is not homogeneous and contains discontinuities. Therefore, linear transformation cannot be applied to decide the proportion of primary color required to generate the desired secondary color.

Figure 2. CIE Chromaticity Diagram



Four-Channel Color Mixing

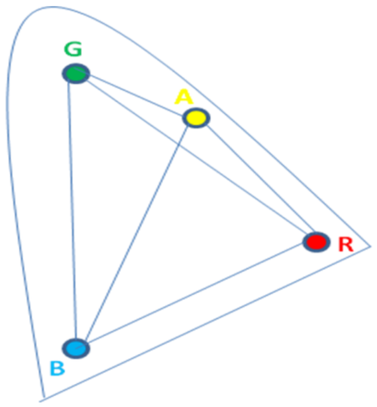
In color mixing applications, the firmware inputs values in CIE chromaticity coordinate form. It converts the coordinates into appropriate dimming values for each LED channel. A dimming value is simply the percentage of maximum luminous flux to which an LED must be dimmed. If an LED has its current quickly switched on and off in an intelligent fashion, the LED has its flux output controlled.

The firmware combines this coordinate with its preprogrammed knowledge of the characteristics of the LEDs in the system. 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.

Four-channel color mixing solution is based on the principle of superposition. It uses three-channel color mixing algorithm as its base. In a three-channel color mix, if the color points of three LEDs are mapped onto the CIE 1931 chart, it forms a triangle. If the three LEDs are red, green, and blue then the triangle formed is called the color gamut (see Figure 2). 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 is input into the system. This provides a broad range and high resolution of unique colors that is produced with this system. For more details of three-channel color mixing and associated mathematics refer AN16035.

For four-channel color mixing, if the color points of four LEDs are mapped onto a color space chart, it becomes apparent that there are exactly four triangles formed by the lines drawn between the four LED color points. See Figure 3.

Figure 3. Superposition on Four-Channel Color Mixing

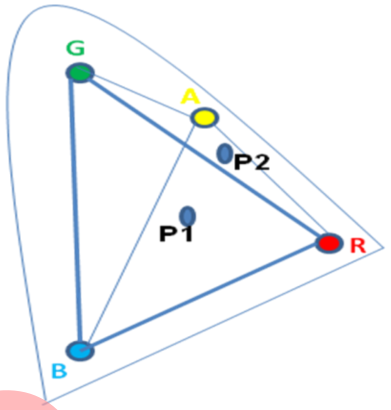


The method described here is easily expandable to more than four LED colors. In Figure 3, the four triangles are made up of the following LED triplets: TR1(R,G,B), TR2(R,A,B), TR3(R,G,A), and TR4(G,A,B).

Each triangle is solved for dimming values using three-channel color mixing functions. Out of these four triangles two give all non negative dimming values and two have one or all dimming values negative. Triangles with any or all negatives values are not valid and are discarded. Dimming arrays with all positive values are accumulated. The interpretation of negative dimming values is that the desired point lies outside the triangle formed by three basic colors. For example, in Figure 4, RGB triangle returns all non negative values for P1; for P2, at least one dimming value is negative.

Two positive dimming values for each desired color are added and scaled appropriately. A negative dimming value implies that the desired color is not inside the gamut so that cannot be generated using the particular base colors.

Figure 4. Positive and Negative Dimming Values

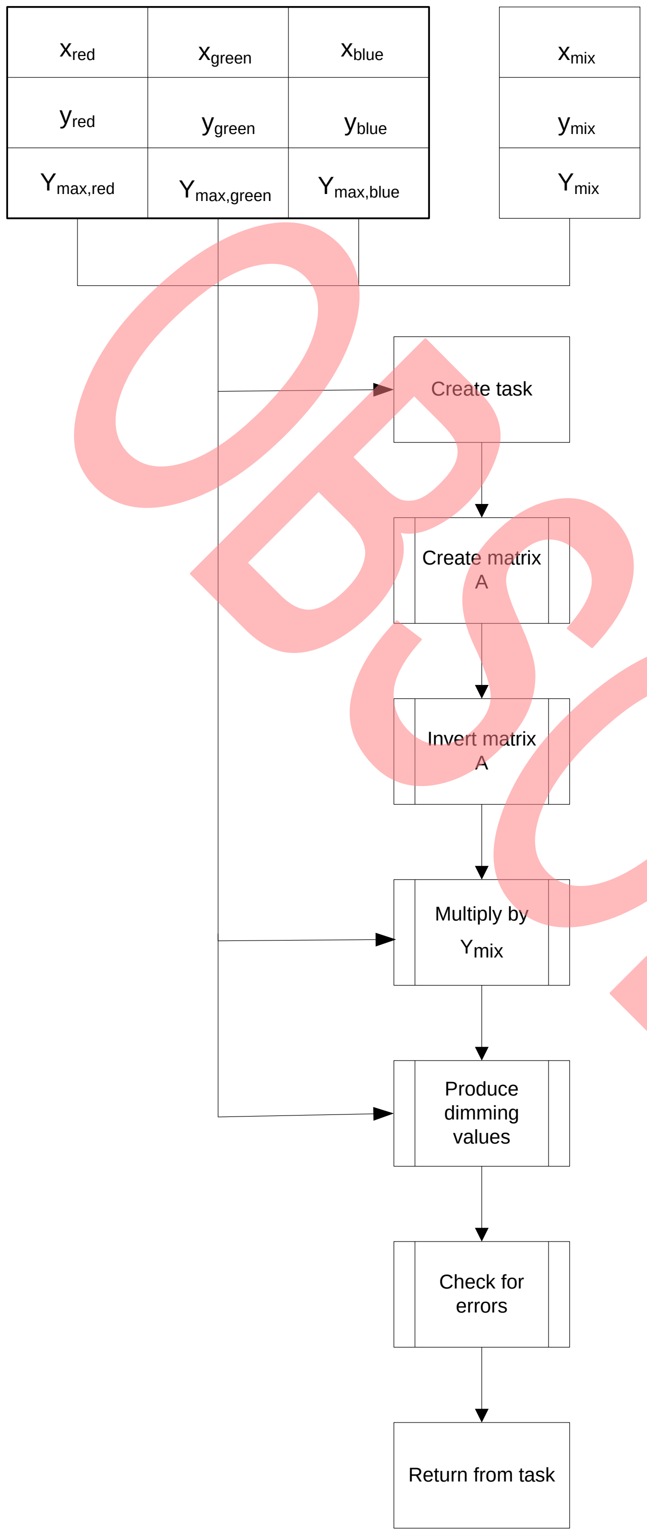


Color Mixing using EZ-Color

The firmware uses CIE 1931 color space to input color requests. A particular point in the CIE 1931 color space is represented with three values (x, y, Y). The point is defined by (x, y) where x and y value represent the color hue and saturation. Hue is one of the dimensions of CIE 1931 color space. Saturation is the second dimension of this color space. The third value of (x, y, Y) vector specifies the luminous flux, in lumens (lm). The firmware must have inputs in the (x, y, Y) vector that specifies its color and flux output at some rated current and junction temperature.

The implementation of four-channel color mix is based on three-channel color mix. The first step in the algorithm is the creation of a matrix. Then, find the inverse of the matrix and multiply it with Y_{mix}. Y_{mix} is the number of lumens that the total mixed light output must produce. These steps are shown in Figure 5.

Figure 5. Flowchart for Three-Channel Color Mixing

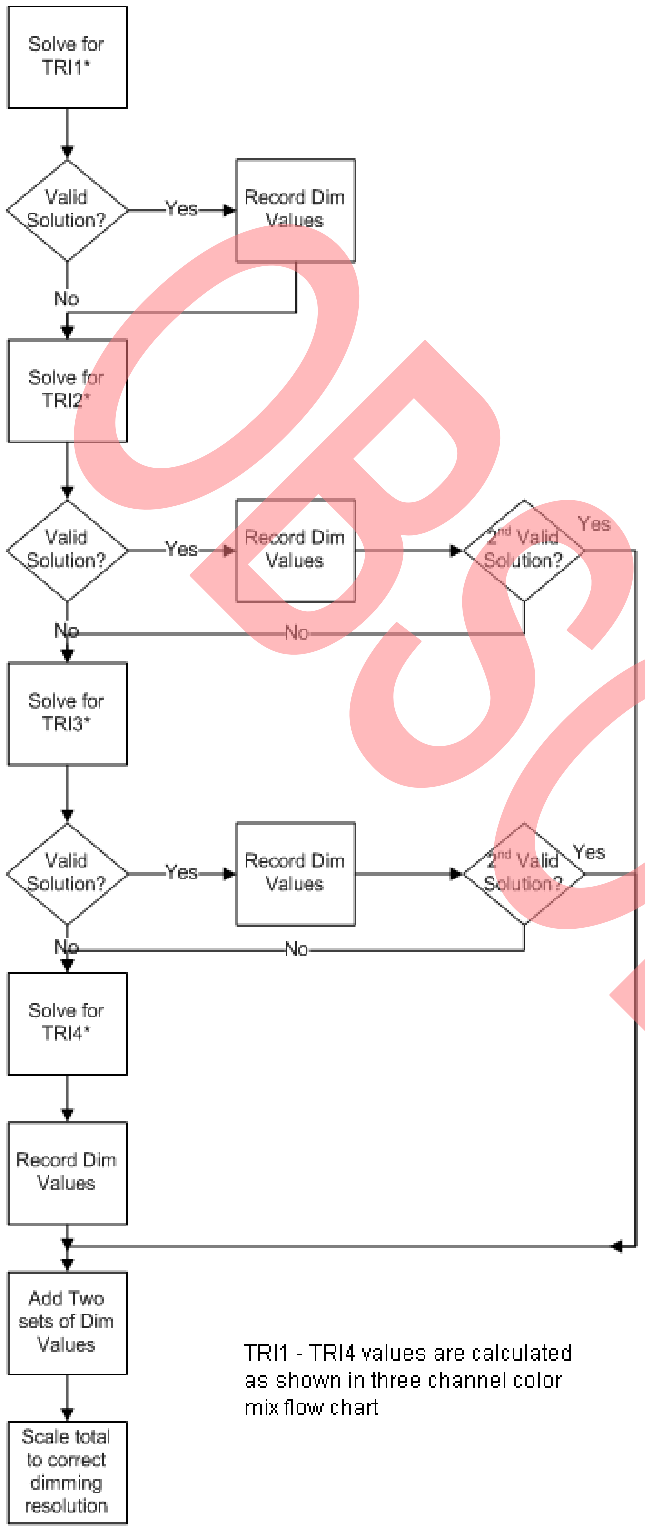


The resultant Y values of the product are the lumen output of each respective LED that is necessary to create the requested color and flux. At this point all math operations give rise to two benefits of doing the math this way. If any of the final product's *Y* values are negative, it signifies that the color coordinate that was requested is invalid. In other words, the requested color was outside the color gamut. Also check if any of the product's *Y* values are larger than the maximum lumen output of any of the three LEDs. This means that the Y_{mix} input was too large. In this case, the firmware scales back the values so that they produce the maximum possible flux at the requested *(x, y)* coordinate. For more details of the three-channel color mix algorithm and implementation, refer AN16035.

The flowchart in Figure 6 describes the steps required for four-channel color mixing algorithm. If the color points of four LEDs are mapped onto the chart, it forms four triangles. These triangles are made up of the following LED triplets: (R,G,B), (R,A,B), (R,G,A), and (G,A,B). These triangles are referred to as TRI1, TRI2, TRI3, and TRI4 in the flowchart. The three-channel algorithm is implemented to solve dimming values for each of these triangles. Each triangle is solved to calculate TRx. If any of the three dimming values obtained from this process are negative, then the solution is invalid. If the solution is valid, the three dimming values are saved. When two sets of three valid dimming values are obtained, there is no need to proceed with the other triangles. The operation flow skips down to the "Add Two Sets of Dim Values" process as shown in Figure 6. The six saved dimming values are added together so that there are four values: one for each of the four LEDs in the system. These four values are scaled to the appropriate dimming resolution and the dimming value solution is complete.

Lastly, these four dimming values are given as inputs to the external drivers which control the brightness of LEDs by modulating the current flowing through each channel. If any three of four solutions are invalid that means the desired color is not present in the color gamut. The user can implement this error condition. It may be done by continuing to retain the old color, turning off the LED, and so on.

Figure 6. Flowchart for Four-Channel Color Mixing



Advantage of Four-Channel Algorithm

The four-channel algorithm has the following advantages:

- It is deterministic and always takes the same approximate amount of time.
- It can be written using fixed point math or floating point math. In this application note, the fixed point math is used because it requires less RAM, code size, and CPU load when compared to floating-point math.
- It is not an iterative algorithm.
- The code size for the solution grows very little with each added LED.

Summary

This application note provides an overview of the four-channel color mix algorithm. This can be easily extended to more LEDs. The solution presented in this document can be extended for several lighting applications. The source code with some minor modifications can also be used as a framework to develop various applications.

Appendix

This firmware combines the following functions:

- I^{2}C slave to take the (x, y, Y) inputs.
- Four-channel color mixing to calculate the dimming values.
- LED driving using SSDM modules to send the dimming signal to external driver, which control the LEDs.

A top level code snippet is illustrated. This snippet shows the main.c and also the function “`LED4_fColorMix`” to calculate the dimming values for a triangle formed by three LEDs. The entire firmware project is attached with this application note. The project includes the various functions to implement the calculation of the dimming values.

```
//Include files
//Structure LED_Char_Block defines things like maximum lumens for the LED, base x,y
coordinate (Y, x, y),
// These values are provided by LED Manufacturer.
#pragma abs_address 0x7FC0
const struct EEPROM_Block
{
    LED Channels[4];
    BYTE Pad[40];

} LED_Char_Block = {

    91, 6951, 3033,         //Red
    110, 1585, 6765,        //Green
    23, 1469, 366,          //Blue
    63, 5945, 4037,         //Amber
};
// These values we get from LED Manufacturer. (x, y, Y)
#pragma end_abs_address

struct StateMachine                   //This structure defines the registers that will be shared
{                                     // with the EncoreIII on the I2C bus.
    // Read/Write fields.

    // LED on/off state,           bit 0:  1 -- ON, 0 -- OFF
    // Direct LED control on/off, bit 1:  1 -- ON, 0 -- OFF
    unsigned char bfStatus;

    // Current  Requested CIE vector.
    unsigned int wCurrentX;
    unsigned int wCurrentY;

    unsigned char bFlux; // 0 to 255 Lumens

    // Direct control LED Dimming Values, 0 to 255
    unsigned char abDimmingValues[NUM_LEDS];

    // Read only fields.

    // LED coordinates (read only).
    unsigned int wRedLEDX;
    unsigned int wRedLEDY;

    unsigned int wGreenLEDX;
    unsigned int wGreenLEDY;

    unsigned int wBlueLEDX;
    unsigned int wBlueLEDY;

} sLED_Data;
```

```
// abDimmingValues[4] Stores the dimming values that have to be loaded in the SSDM Signal
Density register.
unsigned char abDimmingValues[4];
// sMixedColor stores the coordinated of the new color to be set.
COORD sMixedColor;
unsigned int i;


void main()
{

    // Initialize the asLEDArray[4] with the max lumens, x and y coordinates of each LED
    LED asLEDArray[4];

    /* I2C interface initialization */
    // Set up the I2C registers
    EzI2Cs_1_SetRamBuffer(VISIBLE_DATA, WRITABLE_DATA, (BYTE *)&sLED_Data);
    EzI2Cs_1_Start();
       //Start I2C slave

    LED4_Initialize(asLEDArray, LED_Char_Block.Channels);

    /*Set dimmers with signal densities of 1 (off) */
    SSDM_RED_WriteSignalDensity(0x01);
    SSDM_GREEN_WriteSignalDensity(0x01);
    SSDM_BLUE_WriteSignalDensity(0x01);
    SSDM_AMBER_WriteSignalDensity(0x01);

    /* Start the SSDM User Modules */
    SSDM_RED_Start();
    SSDM_GREEN_Start();
    SSDM_BLUE_Start();
    SSDM_AMBER_Start();

    /* Start clocking PWM */
    VC4_Start();

    /* Enable the Interrupts */
    M8C_EnableGInt;                             //Enable interrupts globally

    while(1)
    {
        //Only if the board in not under direct control of the GUI...
        if((sLED_Data.bfStatus & DIR_CTRL_MASK) == 0)
        {
//---------------------------------------------------------------------------------------------
/* This do-while loop below is to ensure that all three values that get passed to the
ColorMix function are stable. They will continually be copied into the 'mirror' variables
until they all equal the original variables. This is to ensure that the I2C master does not
overwrite one or two of the variables as they are getting passed to the ColorMix function.
Now, the values are indirectly passed into the ColorMix function through the mirror
variables, and the do-while loop ensures they are all part of the same set of values.

Note that the mirror variables could possibly be eliminated from this firmware, and other
already-existing global variables could possibly be used to save RAM space. However, the new
mirror variables are used for ease of reading this code. */
//---------------------------------------------------------------------------------------------

            do {
                sMixedColor.wMixed_x = sLED_Data.wCurrentX;
                sMixedColor.wMixed_y = sLED_Data.wCurrentY;
                sMixedColor.bOnPercent = sLED_Data.bFlux;
            } while((sMixedColor.wMixed_x != sLED_Data.wCurrentX) || (sMixedColor.wMixed_y !=
sLED_Data.wCurrentY) || (sMixedColor.bOnPercent != sLED_Data.bFlux));
```

```
            // Calculate the Dimming values
            LED4_fGetDimValues(sMixedColor, asLEDArray, abDimmingValues);

        }

        if(sLED_Data.bfStatus & ONOFF_MASK)     //If LEDs ON flag is high...
        {

            PRT1DR &= ~0x08;                    // Turn SHDN FET off (meaning LEDs are ON).
            SSDM_RED_WriteSignalDensity(abDimmingValues[LED1]);
            SSDM_GREEN_WriteSignalDensity(abDimmingValues[LED2]);
            SSDM_BLUE_WriteSignalDensity(abDimmingValues[LED3]);
            SSDM_AMBER_WriteSignalDensity(abDimmingValues[LED4]);

        }
        else
        {
            PRT1DR |= 0x08;                   // Turn SHDN FET on (meaning LEDs are OFF).
        }
    }
}
```

The function to calculate the dimming value for each triangle follows. This is the three-channel color mix algorithm.

```
unsigned char LED4_fColorMix(COORD sTarget, LED* pLED, unsigned char* pDimArray)
{
    unsigned char i;                //Variable used for indexing
    signed long lScratch1;          //Variable used to hold many temporary values
    signed int iScratch2;           //Variable used to hold many temporary values
    unsigned long dScaleFactor;     //Variable used to hold a scaling factor
    signed int aiOpMatrix[3][3];
    signed long alOpVector[3];

    //Index through this for loop once for each LED color.
    for(i=0; i<NUM_LEDS; i++)
    {
        //Do math operation: x(i) - xMix
        lScratch1 = (signed long)((*(pLED + i)).wLED_x - sTarget.wMixed_x);
        lScratch1 *= 1000;            //Do math operation: (x(i) - xMix) * 1000

        //Do math operation: y(i) / 2, this will be for rounding accuracy
        iScratch2 = (signed int)(*(pLED + i)).wLED_y >> 1;
        if(lScratch1 < 0){
            //Do math operation: ((x(i) - xMix) * 1000) +- (y(i) / 2)
            lScratch1 -= (signed long)iScratch2;}
         //Will subract if (x(i) - xMix) * 1000 is negative, otherwise it will add
         else{
             //This is for rounding correctly
            lScratch1 += (signed long)iScratch2;}

        //Do math operation: (((x(i) - xMix) * 1000) +- (y(i) / 2)) / y(i)
        lScratch1 /= (signed long)(*(pLED + i)).wLED_y;
        //Put that value into the ith column of the 1st row of the operation matrix
        aiOpMatrix[0][i] = (signed int)lScratch1;
    }
    //Index through this for loop once for each LED color.
    for(i=0; i<NUM_LEDS; i++)
    {
        //Do math operation: y(i) - yMix
        lScratch1 = (signed int)(*(pLED + i)).wLED_y - sTarget.wMixed_y;
        //Do math operation: (y(i) - yMix) * 1000
        lScratch1 *= 1000;

        //Do math operation: y(i) / 2, this will be for rounding accuracy
        iScratch2 = (signed int)(*(pLED + i)).wLED_y >> 1;

        if(lScratch1 < 0){
             //Do math operation: ((y(i) - yMix) * 1000) +- (y(i) / 2)
            lScratch1 -= (signed long)iScratch2;}

         //Will subract if (y(i) - yMix) * 1000 is negative, otherwise it will add
         else{
             //This is for rounding correctly
            lScratch1 += (signed long)iScratch2;}

        //Do math operation: (((y(i) - yMix) * 1000) +- (y(i) / 2)) / y(i)
        lScratch1 /= (signed int)(*(pLED + i)).wLED_y;

         //Put that value into the ith column of the second row of the operation matrix
         aiOpMatrix[1][i] = (signed short)lScratch1;
    }

    //Index through this for loop once for each LED color.
    for(i=0; i<NUM_LEDS; i++)
    {
        //Put value of 1 into ith column of the third row of the operation matrix
        aiOpMatrix[2][i] = 1;
    }


    i = LED4_fMatrixInverse_3x3(aiOpMatrix, alOpVector);
    //Do an Inversion operation on the matrix
    //This matrix inversion will return the values in the 3rd column of the matrix
    //as a vector of values into the alOpVector arrray. It returns only these three
    //values because they are the only ones of importance as far as calculating
    //dimming values is concerned.
    if(i == INVALID_COLOR)
    {
        return INVALID_COLOR;
    }

    //Index through this for loop once for each LED color.
    for(i=0; i<NUM_LEDS; i++)
    {
         //Multiply each element of a constant of 100 to scale things correctly
         alOpVector[i] *= 100;

         //Do math operation: alOpVector(i) / 256, this is to scale the values
         alOpVector[i] <<= 8;

         //Do math operation: Lumens(i) / 2, this will be to round correctly
         iScratch2 = (signed int)(*(pLED + i)).wRatedFlux >> 1;

         //Do math operation: (alOpVector(i) +- (Lumens(i) / 2)
         if(alOpVector[i] < 0){
             //Will subract if alOpVector(i) is negative, otherwise it will add
            alOpVector[i] -= (signed long)iScratch2;}
        else{ //This is for rounding correctly
            alOpVector[i] += (signed long)iScratch2;}

        //Do math operation: (alOpVector(i) +- (Lumens(i) / 2)) / Lumens(i)
        alOpVector[i] /= (signed long)(*(pLED + i)).wRatedFlux;
    }

    //The following if chain simply puts the largest of the three dimming
    if(alOpVector[LED1] > alOpVector[LED2]){

        //ratios in the lScratch1 variable.
        lScratch1 = alOpVector[LED1];
    }else{
        lScratch1 = alOpVector[LED2];
    }
    if(lScratch1 < alOpVector[LED3]){
        lScratch1 = alOpVector[LED3];
    }


    for(i=0; i<NUM_LEDS; i++)
    {
        alOpVector[i] *= 255;
        alOpVector[i] += (lScratch1 >> 1);
        alOpVector[i] /= lScratch1;

        //Set each dimming value in the sLED_Data structure
        *(pDimArray + i) = (unsigned char)alOpVector[i];
        //If any of them happen to be zero, set them to 1 instead
        if(*(pDimArray + i) == 0)
        {   //The dimming value 1 is the same thing as OFF
            *(pDimArray + i) = 1;
        }
    }

    //This point is reached if there were no previous color mixing errors.
    //In other words, the requested color is valid, and too many lumens were not requested.
    return NO_ERROR;        //Return the code that signifies there were no errors.
}
```

About the Author

Name:	Anshul Gulati
Title:	Applications Engineer Sr
Background:	Anshul Gulati holds a bachelors degree in Electrical and Electronics from BITS, Pilani, India and is currently working on Cypress' lighting solutions.
Contact:	gula@cypress.com

Document History

Document Title: Multi Channel Color Mixing Using HB LEDs

Document Number: 001-51188

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2646753	GULA	01/26/09	New application note.
*A	3113854	SNVN	12/17/2010	Update application note with location of example projects
*B	3148689	QUS	01/20/2011	Updated associated example project.
*C	4309120	SREH	03/14/2014	Obsolete document because the EZ-Color parts are obsolete

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