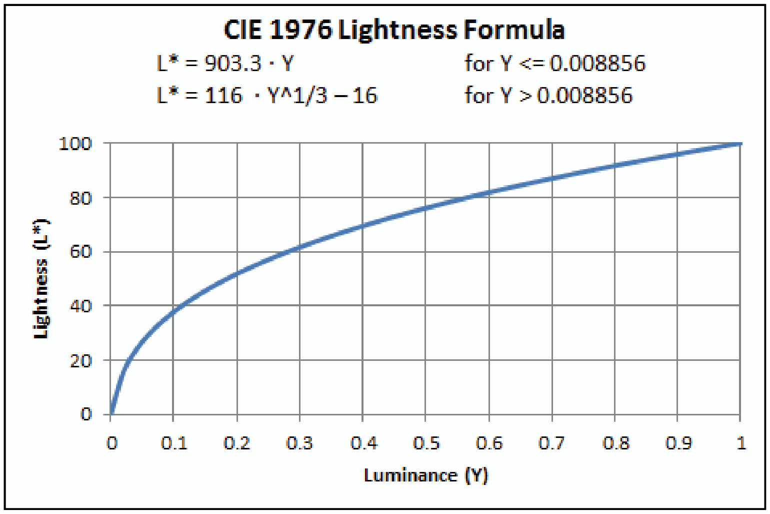
Controlling LED brightness using a PWM and a Compensation Table

Using PWM (pulse width modulation) to control brightness of an LED is relatively simply task for just about any microcontroller. However, in using this method you may have noticed that the brightness of the LED is not linear with respect to the pwm signal being used for dimming. In other words, using a pwm duty cycle of 50% does not dim the LED by 50%. What you typically see is that the LED brightness ramps quickly and as you reach a pwm duty cycle of 50% you have nearly reached the maximum brightness. This can be explained by the CIE 1976 lightness formula which shows that human perception of brightness is not linear.

The X-axis on the graph could essentially be replaced by "PWM Duty Cycle" and you can get an idea of what the problem is. So how do we fix this? The simplest way would be to determine the inverse equation of the this curve and run every incoming pwm value through this equation to determine a "compensated" value that is used for the actual pwm duty cycle.

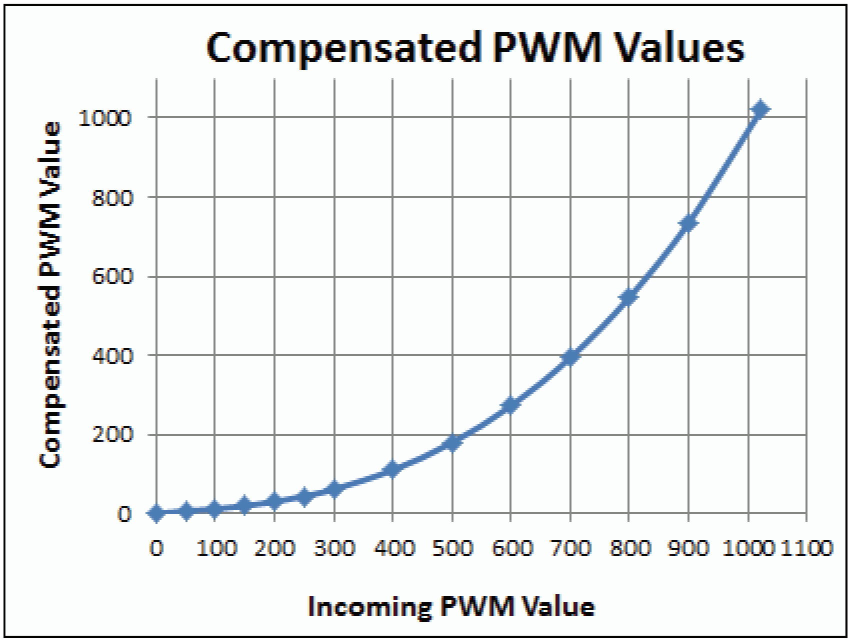


For a 10 bit PWM (1024 steps), the inverse equation would look like this:

Compensated PWM value = (((((PWM$_{IN}$ * 100)/1024) + 16)/116)^3) * 1024
where PWM$_{IN}$ is the uncompensated PWM input value in the range of 0 - 1023

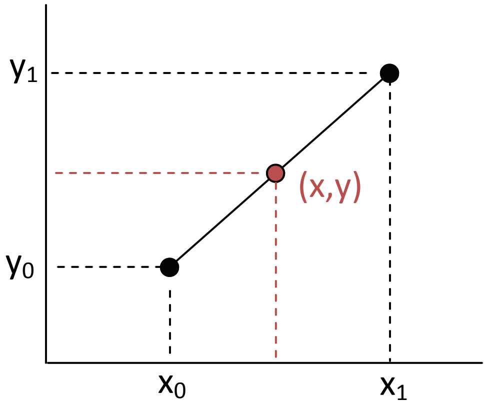
While certainly possible, it is not practical for some microcontrollers to calculate every point because of the memory requirements and calculation time required for complex math functions. However, there is another method that can be used that is relatively simple and greatly reduces the resources required by the MCU. This method is know as **linear interpolation** or linear approximation and uses a table of several points along the curve to approximate the value of any point on the curve. The number of points required in the table depends on how non-linear the function is that you are trying to approximate but for something like the brightness curve, this can easily be done with less than 20 points. A table created from the inverse equation above and the plot made from these values are shown below.

Array Element	Incoming PWM Value (x)	Compensated PWM Value (y)
0	0	0
1	50	6
2	100	11
3	150	19
4	200	29
5	250	43
6	300	61
7	400	110
8	500	179
9	600	272
10	700	394
11	800	547
12	900	736
13	1023	1021



Using the interpolation table

The plot below shows the premise for how the points in the table are used to find the approximated result (y) for a given input (x). In our application, the x value represents the incoming pwm value and the y value represents the compensated or corrected PWM value. The x and y values for x_0,y_0 and x_1,y_1 are known points from the table and using the equation below you can find the value of y for a given value x that falls between x_0 and x_1.



Equation for finding the value y given a value x that falls between x_0 and x_1.

$$y = y_0 + (y_1 - y_0)[(x - x_0)/(x_1 - x_0)]$$

Example

Using the table on the previous page, find the compensated PWM value for an input PWM value of 325.

Going to the table we see that our input PWM value of 325 falls between the values in the table of (300,61) and (400,110).

Therefore the values in our interpolation equation and result are as follows:

$x = 325$ // Incoming pwm value
$x_0 = 300$, $y_0 = 61$ // point in table “below” our incoming value
$x_1 = 400$, $y_1 = 110$ // point in table “above” our incoming value

Using the equation to the left, we solve for y:
y = 61 + (110 - 61)[(325 - 300)/(400 - 300)] = <u>**73.25**</u>

So for an input PWM value of 325, the compensated PWM value is 73.25.

Implementation for XC8 Compiler

Implementing the table requires that the values are stored in an array for easy access like this:

```
const unsigned int x_interp[14] = {0,50,100,150,200,250,300,400,500,600,700,800,900,1023};
const unsigned int y_interp[14] = {0,6,11,19,29,43,61,110,179,272,394,547,736,1021};
```

And we need another constant that indicates the maximum array index

```
const unsigned char max_array = 13; // array elements are 0 to 13
```

The qualifier `const` is not required but can be used as the values in the table are constant and the XC8 compiler will use program memory rather than RAM to store the values. This can be beneficial for situations where RAM space is at a premium.

The function on the next page shows how you can implement linear interpolation to calculate a corrected pwm value based on the table shown on the first page. This routine takes an input of an un-compensated pwm value and returns the compensated value. Upon entry into the routine, it will start at array element #1 in the table (because element 0 is zero) and increments through the x values in the table until it finds the first value that is larger than the incoming pwm value. That point in the table becomes x_1,y_1 and the previous point in the table becomes x_0,y_0 . The value for y (corrected pwm) is then calculated and returned by the function.

<u>NOTE:</u> the actual equation for calculating the corrected value can be problematic because of fractional values less than one and how the compiler deals with this for different data types. The equation in the code below was written in a format specifically for use with the XC8 compiler and unsigned int data types for all the variables. If you choose to change the data types you may need to change how the equation is written to get valid values.

Function for calculating corrected PWM value using linear interpolation table

Note: The declarations for the x and y interpolation arrays and max_array variables on the previous page need to be added as global variables prior to calling this function.

```
unsigned int LinearizedPWM(unsigned int PWMinput)

{
    unsigned int x,x0,x1,y0,y1;   //known points on curve (from table)
    unsigned int compensated_pwm; // calculated pwm to compensate for curve
    unsigned char i;
    unsigned char valueFound;

    // first deal with input pwm values of zero or less

    if (PWMinput <= 0)
        compensated_pwm = y_interp[0];  // grab first value in table and jump out
    else
    {
        // first find the x value (pwm input) point in the array that matches or is smaller
        // then the given input

        i=0;
        valueFound = 0;

        while (!valueFound)
        {
            if (x_interp[i+1] >= PWMinput ) // find first value in table equal or smaller than input pwm
            {
                valueFound = 1;
                if (x_interp[i+1] == PWMinput) // if they are equal then we already know the y value
                    compensated_pwm = y_interp[i+1];
                else // we have to calculate
                {
                    x = PWMinput;
                    x0 = x_interp[i];
                    y0 = y_interp[i];
                    x1 = x_interp[i+1];
                    y1 = y_interp[i+1];
                    compensated_pwm = y0 + ((y1-y0)*(x - x0))/(x1 - x0);
                }
            }
            else
            {
                if ((i+1)>= max_array)// then we use last values in the table

                {
                    compensated_pwm = y_interp[max_array];
                    valueFound = 1;
                }
                else
                    i++; // jump to next set of values in the array
            }

        }   // end of while loop

    } // end of if/else loop


    return (compensated_pwm);

}
```

Theoretical vs. reality and some minor modifications

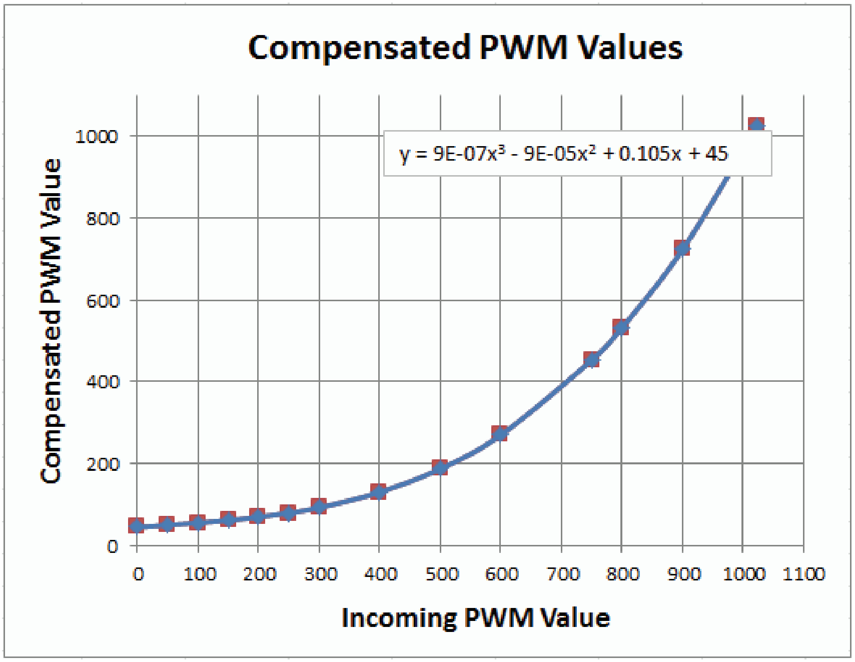
All the material presented prior to this section was based on attempting to compensate for the ideal CIE curve on the first page. As you may have guessed, PWM frequency, different LEDs, the driver circuit being used and other factors can affect the actual LED response to a given PWM signal. In our testing the ideal curve was an excellent place to start and we only ended up tweaking it slightly based on ramping the LED brightness up and down and visually observing the result. We did have to alter the minimum PWM value based on the fact that the LEDs don’t turn on at all until a certain duty cycle is reached. We found that the green LED turned on before the red or blue and required a duty cycle count of about 50/1023 to turn on. The LEDs we were using were Luxeon Rebel LEDs.

Our final formula for the compensation curve was this:

$y = 9\text{E-}07x^3 - 9\text{E-}05x^2 + 0.105x + 45$ where x is the uncompensated pwm value from 0-1023 and y is the compensated value. The graph of this formula and the table of points used in our final code example are shown below.

```
const unsigned int x_interp[14] = {0,50,100,150,200,250,300,400,500,600,750,800,900,1023};
const unsigned int y_interp[14] = {45,50,56,62,70,80,93,130,188,270,453,532,723,1022};
```

Array Element	Incoming PWM Value (x)	Compensated PWM Value (y)
0	0	45
1	50	50
2	100	56
3	150	62
4	200	70
5	250	80
6	300	93
7	400	130
8	500	188
9	600	270
10	750	453
11	800	532
12	900	723
13	1023	1022


Compensated PWM Values
y = 9E-07x³ - 9E-05x² + 0.105x + 45
Compensated PWM Value
Incoming PWM Value