Freescale Semiconductor
Application Note

Document Number: AN3815
Rev. 0, 04/2009

Implementing a Modular High Brightness RGB LED Network

by: Bruno Bastos, Humberto Carvalho, Alexandre Dias, Renato Frias
Freescale Design Center Brazil

Contents

1 Introduction

The popularity of using high-power LEDs for lighting applications is increasing constantly, as the cost of these devices becomes more attractive. However, to efficiently drive these LEDs a switched power supply is required.

This document describes a simple application in which a DSC (digital signal controller) network is used to drive a high-power RGB LED matrix. The MC56F8006 is the Freescale device selected for this application.

The high-speed serial peripheral featured on the DSC ensures fast communication between the network nodes. The color matching control algorithm is managed by the analog-to-digital converter, programmable gain amplifier, and PWM modules.

1.1 Objective

The document describes the implementation of a modular RGB LED matrix using the MC56F8006 digital signal controller. Two different boards are used on the

freescale semiconductor

application, and the hardware and software for both designs are discussed. To support the writing of this document some prototypes were built and tested — the last section of this document provides information about those tests. The software used on the tests is available for download along with this document.

2 System Overview

The system consists of a high power RGB LED network and a gateway board that feeds the LED network with commands. The LED RGB network is formed by a set of pixel boards, each with an MC56F8006 DSC and a high-power RGB LED.

The gateway board receives information from a computer or from a network and addresses each pixel board to update the color it is displaying. It can also receive feedback from the different pixel boards. Each pixel board has its own address. They receive information from the network master on the gateway board.

The connection between the gateway and computer could be implemented by USB or Ethernet and the communication on the LED network could use industrial or specific lighting protocols. On this application example USB is used on the former and RS-485 on the latter.

Figure 1 depicts the connection between the different system elements.

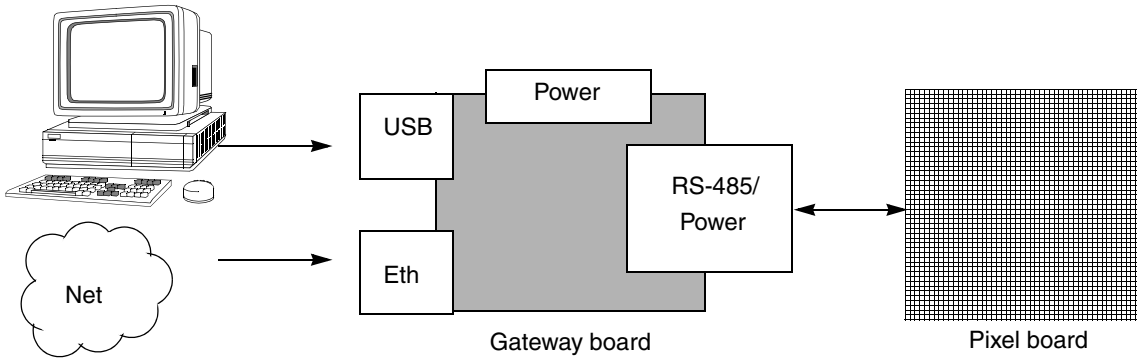


Figure 1. System Overview

The next sections will discuss in detail the pixel and gateway boards.

3 Pixel Board Overview

The main components on the pixel board are the high power RGB LED and the MC56F8006 DSC. The DSC drives the LED according to either the data received from the RS-485 network or preset values stored on DSC internal flash memory. The color mixing algorithm relies on the PWM and analog-to-digital modules to display the desired color. Figure 2 shows the pixel board main components.

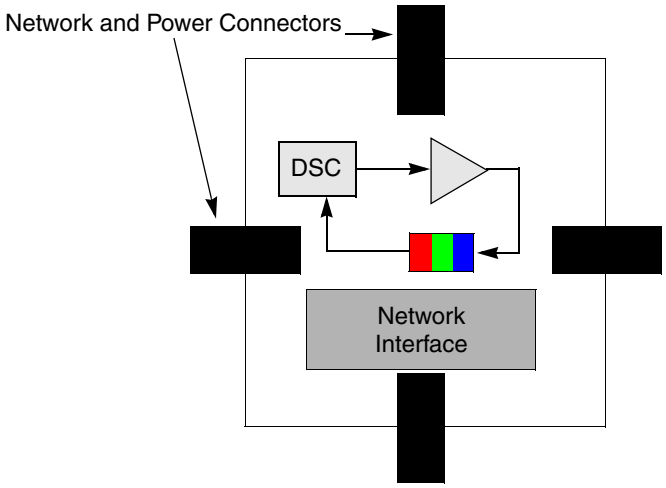


Figure 2. Pixel Board Components

3.1 Pixel Board Requirements

Some pixel board requirements are:

- Power distribution over the pixel network
- High-speed communication interface
- Efficient LED control by constant-current or switched power supply
- Color mixing algorithm aided by color sensor or current feedback

To achieve these requirements, the following features were implemented:

- Connectors were placed on the four sides of the board to distribute power and network signals among the pixel boards.
- An RS-485 transceiver was connected to the DSC's serial communication interface (SCI), which on the MC56F8006 can be set to up to 6 Mbps. RS-485 is a robust industrial standard and several network protocols can be implemented over it.
- A buck converter was implemented to drive the high-power LED. LEDs are current-driven devices, and a switched power supply is an efficient and simple way to drive them.
- The analog-to-digital module on the DSC was used to sense the current on the LED. A programmable gain amplifier module can also be used to amplify the input signal up to 32×, giving a better work range for the ADC. Another approach to solving the feedback issue would be the use of a color sensor, which would provide a more accurate color mixing algorithm. However, the drawbacks of this solution are the cost of the part and the more sophisticated mechanics that it would require to reflect the LED light towards the sensor.

3.2 Hardware Description

This section provides an example of the pixel board hardware implementation. For explanatory purposes, it can be broken down into four different functional parts:

- Controller

- LED driver circuitry
- Address switch
- Network interface and power supply connectors

The figures below show the schematics of these four different parts.

3.2.1 Main Controller

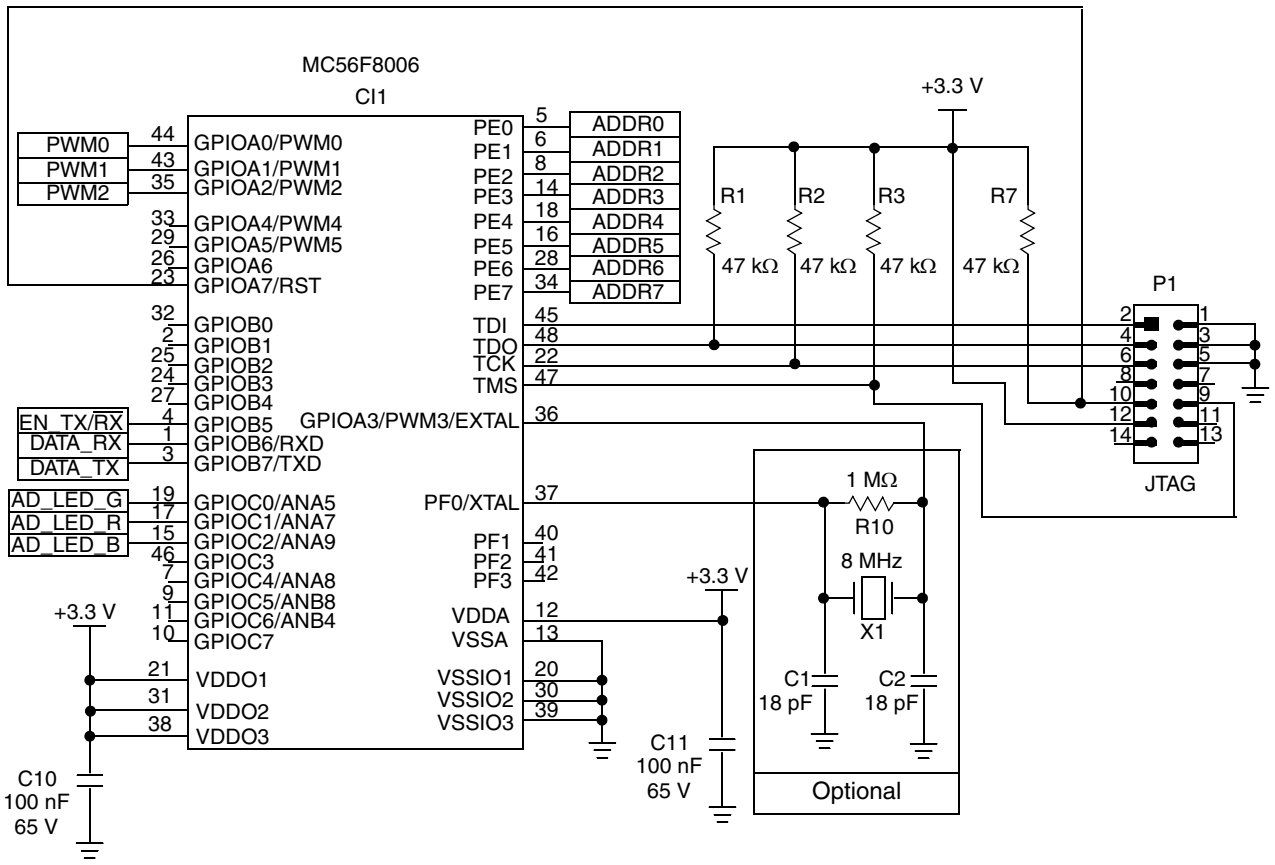


Figure 3. MC56F8006

Instead of the MC56F8006 a simpler 8-bit microcontroller (9S08-based) could be used on this application as the main controller. However, the choice of the MC56F8006 DSC ensures faster communication on the pixel matrix and a more accurate color mixing algorithm.

The MC56F8006 uses the 56800E core, based on a dual Harvard architecture. A rich peripheral set is integrated on the SoC allowing different applications to be implemented while reducing the overall component count on the system. Pins associated with these peripherals can be configured as general purpose input/output.

Two devices are part of the MC56F8006 family, the MC56F8006 and the MC56F8002. The difference between them is the flash memory size. Both devices are available in different packages and pin counts. A complete list of peripherals and features for these devices can be found in the MC56F8006 documentation downloadable from the Freescale website, www.freescale.com.

As can be seen in Figure 3, the MC56F8006 is powered by a 3.3 V supply. It has an 8 MHz internal relaxation oscillator, and can feed the PLL module to generate higher frequencies to be used by the SCI module. Therefore the use of an external clock reference is optional.

To generate the SCI clock, the internal oscillator works as an input to the PLL module that outputs a 192 MHz clock. It then is divided by two to achieve the 96 MHz high-speed peripheral clock. The baud rate generator on the SCI then uses this 96 MHz clock to communicate up to 6 Mbps.

A JTAG interface is also used on the DSC circuit. It is used to program the controller internal flash memory and to debug software applications. Four DSC pins are used on the JTAG interface: TDI, TDO, TMS, and TCK.

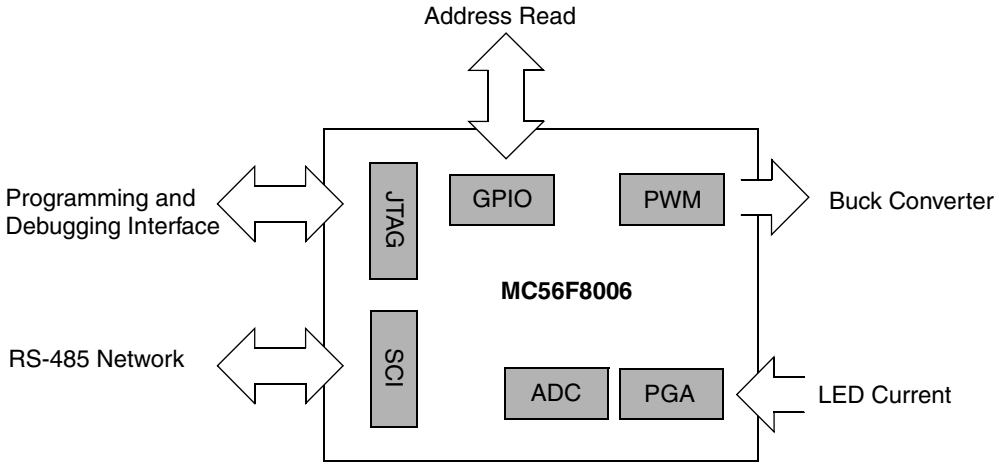


Figure 4. MC56F8006 Peripheral Modules

As described above, the MC56F8006 has several integrated peripherals. The ones used on the pixel board are:

- Three PWM channels connected to field-effect transistors to control the buck switch for the LED current supply
- Eight I/O's used for address and system setup configuration
- Three analog-to-digital inputs to read the current flowing on the LEDs
- SCI connected to a RS-485 transceiver, using three pins: Tx, Rx, and a GPIO for data enable.

Another feature that could be used is the programmable gain amplifier (PGA), which is intended to operate in concert with the on-chip analog-to-digital converter (ADC). By itself, the PGA has no useful function. When used to pre-process ADC inputs, it amplifies up to 32× and converts differential signals to a single-ended value, which is passed on to the ADC for conversion to digital format. The MC56F8006 has two PGAs with differential inputs — on this application three analog signals need to be read, for the red, green, and blue LEDs. Therefore the PGA will not be used, and the analog signal will input directly to the ADC.

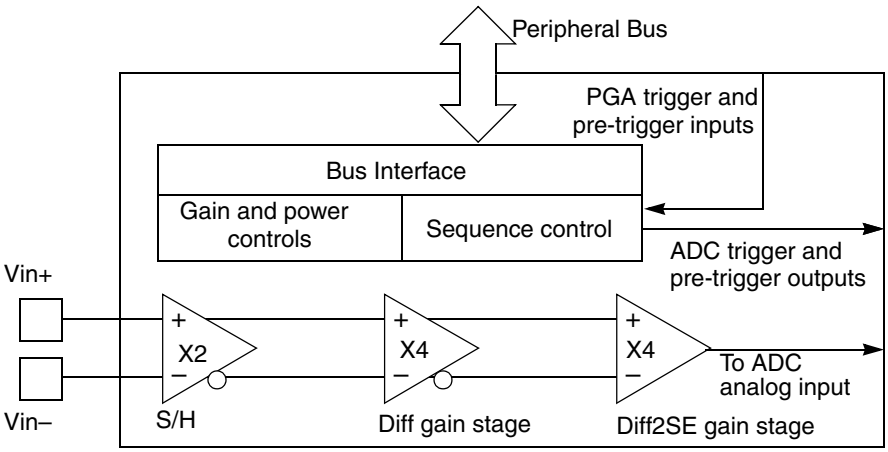


Figure 5. PGA Module

Table 1. Pins Used

Pin Group	Pins Used
JTAG	4
I/O for Address	8
PWM	3
SCI	3
ADC	3
Power	8
Xtal (Optional)	2
Total Used	31

3.2.2 LED Control

The buck circuit is used to drive the high brightness LED, as shown in Figure 6.

The operation of this topology has two distinct periods. The first one occurs when the series switch is on. The input voltage (5 V) is connected to the input of the inductor (L). The output of the inductor is the output voltage (V_{Out}), and the rectifier diode is reverse-biased. Because there is a constant voltage source connected across the inductor during this period, the inductor current begins to linearly ramp upwards.

During the on period, energy is stored within the core material in the form of magnetic flux. If the inductor is properly designed, there is sufficient stored energy to carry the requirements of the load during the off period.

The next period is the off period of the power switch. When the power switch turns off, the voltage across the inductor reverses its polarity and is clamped at one diode voltage drop below ground by the diode. The current now flows through the diode, maintaining the load current loop. This removes the stored energy from the inductor.

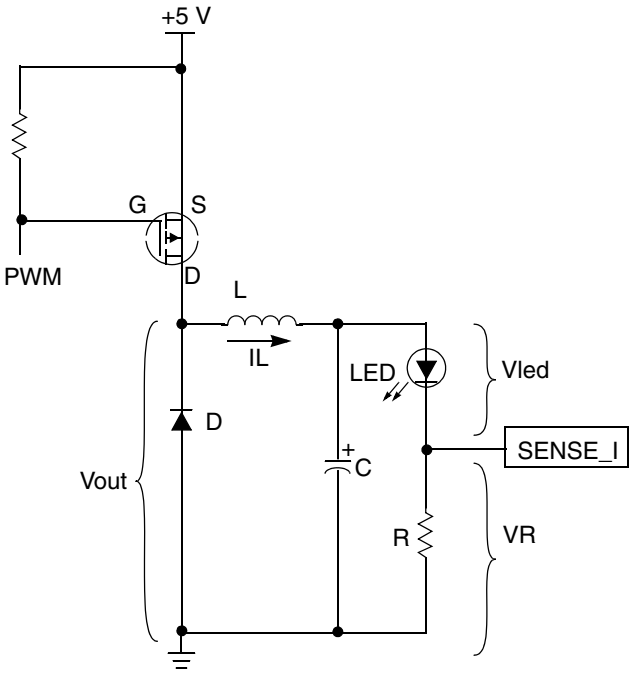


Figure 6. Buck Topology

Figure 7 shows the signals on the Buck circuit:

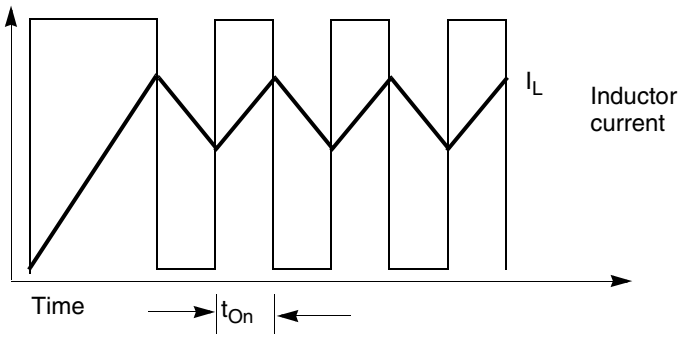


Figure 7. Inductor Current (IL)

This circuit is used to reduce the input voltage (5 V) to a lower level appropriate for each LED. It is not equal for all of them, as the brightness is not identical for the same current for different colors.

The color luminous intensity of LEDs that are of different colors is also not uniform.

An example of forward voltages versus brightness is given below:

Condition: IF (Forward Current) = 250 mA, 25° C

Red	7150	mcd	2.3 V
Green	11250	mcd	3.5 V
Blue	2850	mcd	3.5 V

These features require a color mixing algorithm that must be executed in the DSC. Its faster execution of instructions and peripheral set allow more accuracy on the color control functions than lower-end microcontrollers.

To provide a specific voltage and current to LEDs, it is first necessary to identity the values of components L, C, and R shown in Figure 6.

The value of the resistor must be low to avoid high voltage in the input AD channel. Therefore, the selected value is:

R = 5 ohms

The maximum current value is 250 mA for each LED.

Then, the VR will be:

$VR = R \times I_{LED}$

$VR = 5 \times 0.250$

VR = 1.25 V

Now it is possible to calculate the V_{Out} needed for each LED:

$$V_{Out\_Color} = V_{Out\_LED} + VR \qquad \textit{Eqn. 1}$$

Color	VF [V]	V_{Out_Color} [V]
Red	2.3	3.55
Green	3.5	4.75
Blue	3.5	4.75

Freescale application note AN3321, “High-Brightness LED Control Interface,” available on the Freescale website, provides useful information on high-brightness LED circuit design. Some equations from this application note were applied to calculate the inductor and capacitor values that were used on the pixel board.

3.2.2.1 Inductors

Voltage Across the Inductor

$$V_{L\text{-}on} = V_{In} - V_{Out} \qquad \textit{Eqn. 2}$$

(while the switch is on)

VL-on for LED Color	VL-on [V]
Red	1.45
Blue and Green	0.25

$$VL\text{-}off = V_{Out} \quad \textit{Eqn. 3}$$

(while the switch is on)

VL-off for LED Color	VL-on [V]
Red	3.55
Blue and Green	4.75

Duty Cycle Values

$$D = V_{Out}/V_{In} \quad \textit{Eqn. 4}$$

Duty Cycle for Color	%
Red	71%
Blue and Green	95%

Inductor Value

$$L = (VL\text{-}on \times D / F)/ I_{Ripple\_Max} \quad \textit{Eqn. 5}$$

With a switching frequency (F) of 300 kHz and a max ripple current of 20 mA

Inductor Value	L [uH]
Red	50
Blue and Green	12

3.2.2.2 Capacitor Value

$$C \geq IL / (8 \times FS \times V_{Out}) \quad \textit{Eqn. 6}$$

Capacitor	C [nF]
Red	10
Blue and Green	10

Figure 8 shows the circuit with the values calculated above. To make the pixel board design simpler, the power supply of the LEDs was implemented on the gateway board. This also allowed a smaller pixel board dimension. The power supply that feeds all the LEDs on the network is 5 V. The trade-off of this approach is that an N-channel FET is used as a pre-drive for the P-channel FET gate, as MC56F8006 I/Os are 3 V and this voltage level would not be enough to turn off the P-channel FET.

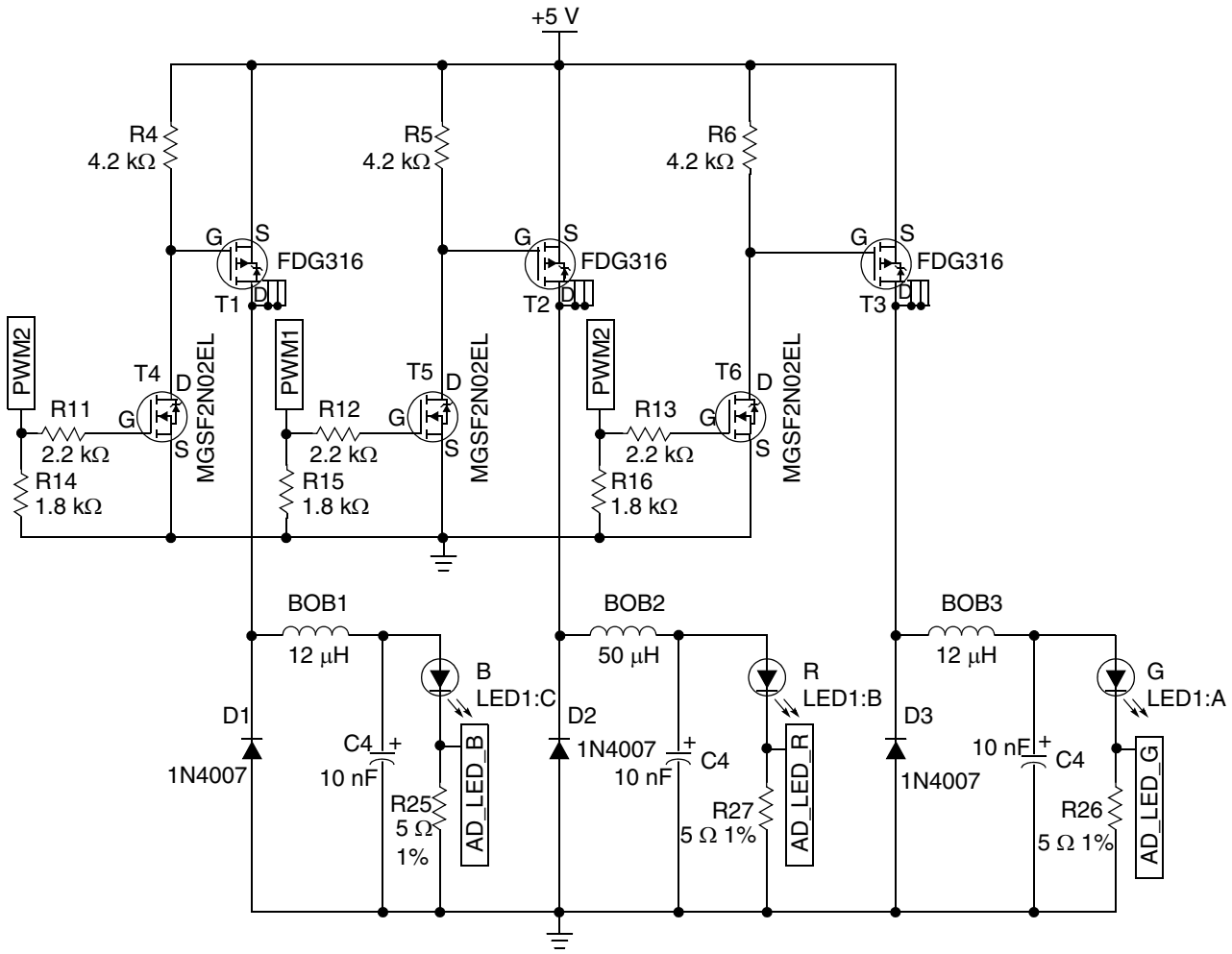


Figure 8. LED Drive

3.2.2.3 Current-Sense Resistor

The voltage V_R across the current-sense resistor R is directly proportional to the current through the LED.

This voltage is measured when it is sampled in the AD channel and converted in a level of current that corresponds to the color luminosity.

The system compares this information with the requested value of the luminosity and revises the PWM duty cycle. If the current value is lower, the PWM increases it until is the correct level, and vice-versa.

As an example, if the requested value of luminosity is proportional to 100 mA in the LED, the voltage in the resistor would be:

$$V_R = 5\ \Omega \times 0.1\ A = 0.5\ V \qquad \textit{Eqn. 7}$$

If the voltage measured by AD is 0.7 V, the PWM is decreased until the voltage in the resistor results in 0.5 V.

Figure 9 shows this situation:

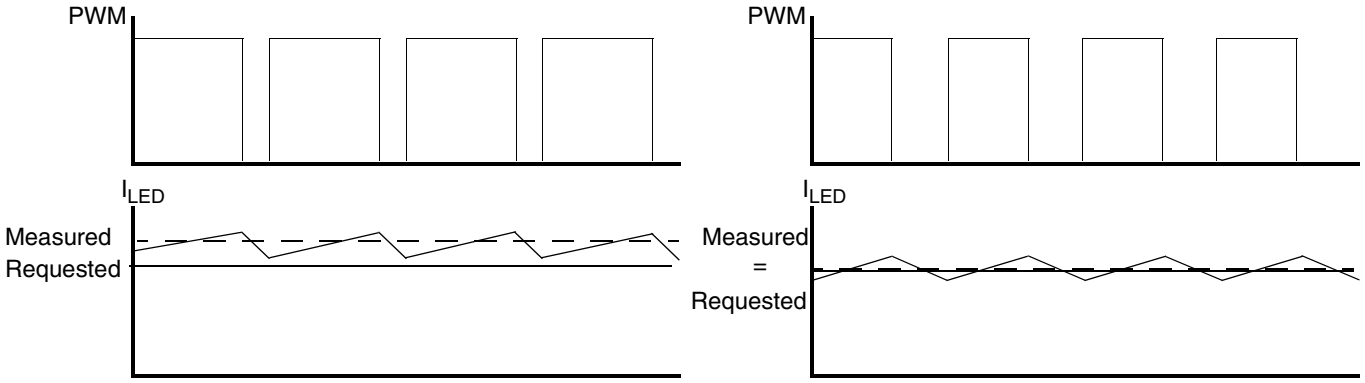


Figure 9. PWM Adjustment

3.2.3 Address Switch

The 8-bit switch shown in Figure 10 is used to set the pixel board address. Some bits could be used as configuration pins. The use of this switch is optional as there are different ways to set the pixel board address.

The address could be hard-coded on MC56F8006 flash, avoiding a hardware switch. However, each pixel board would have to be flashed with a different code, and this would provide less flexibility when assembling the pixel network organization.

Another approach would be to assign the address automatically based on the pixel board neighbor-to-neighbor connections. The drawback in this case is the software design effort required to implement it.

Besides the PCB space constraints, the hardware switch option was chosen due to its simplicity in implementation and its flexibility when configuring the pixel board network. Using the 8-bit switch 256 boards are allowed on the network. If the pixel network constitutes a matrix the software can also read the part switches representing lines and the rest columns. For example, bits 0 to 3 on the switch could represent the line where the board is, whereas bits 4 to 7 would represent the column. In this way a 16 × 16 matrix could be formed.

The address is read at the initialization of the code, before the main loop. It is easy to modify the way the address is assigned to the board by software.

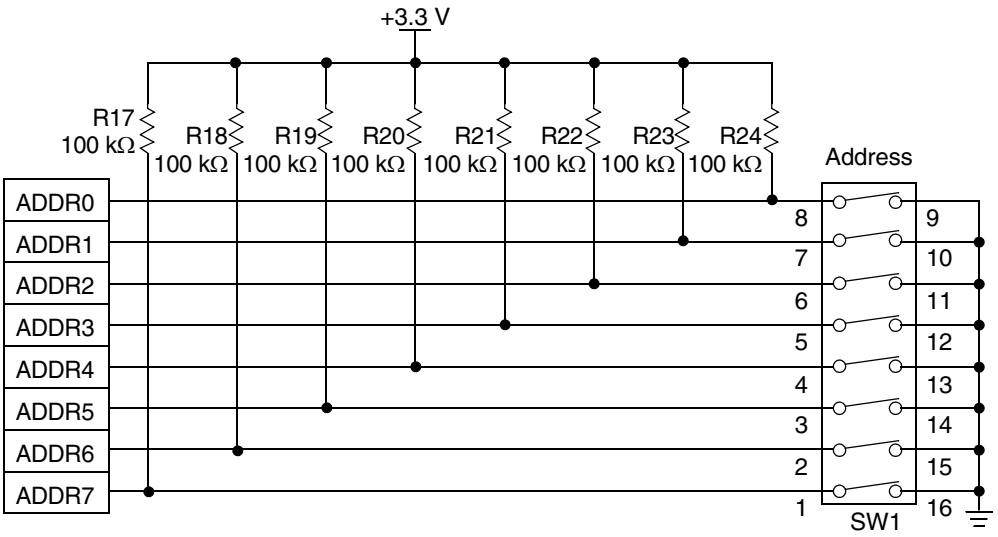


Figure 10. Address and Configuration Switch

3.2.4 Network and Power

RS-485 is an industrial protocol, and there are some lighting standards that communicate over it. RS-485 can be used in both simplex and half-duplex mode. The signal is transmitted using only two wires, named A and B. The signal is shown in Figure 11.

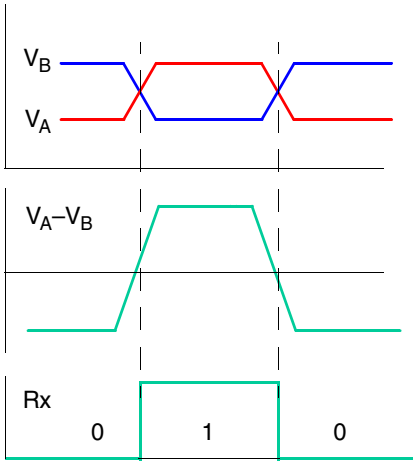


Figure 11. RS-485 Signals

The RS-485 specific IC interface works at up to 20 MHz. For this application, the speed used will be 6 MHz.

The main feature of RS-485 is the communication in differential mode that avoids interference from noises. If the A wire receives interference, the B wire will receive interference in the same way, but the voltage difference between the two wires will be the same and the transmitted signal will therefore not be damaged.

The board is powered externally through its external connectors CON1, CON5, CON2, and CON6, allowing connections with its neighbor pixel board.

Voltage supplied through this connector should be positive 5 V, and signals A and B from the RS-485 network should be pin 3 and pin 4 respectively. Table 2 and Figure 12 show the connections.

Table 2. External Connections

1	5 Volts
2	5 Volts
3	RS485 – A
4	RS485 – B
5	Ground
6	Ground

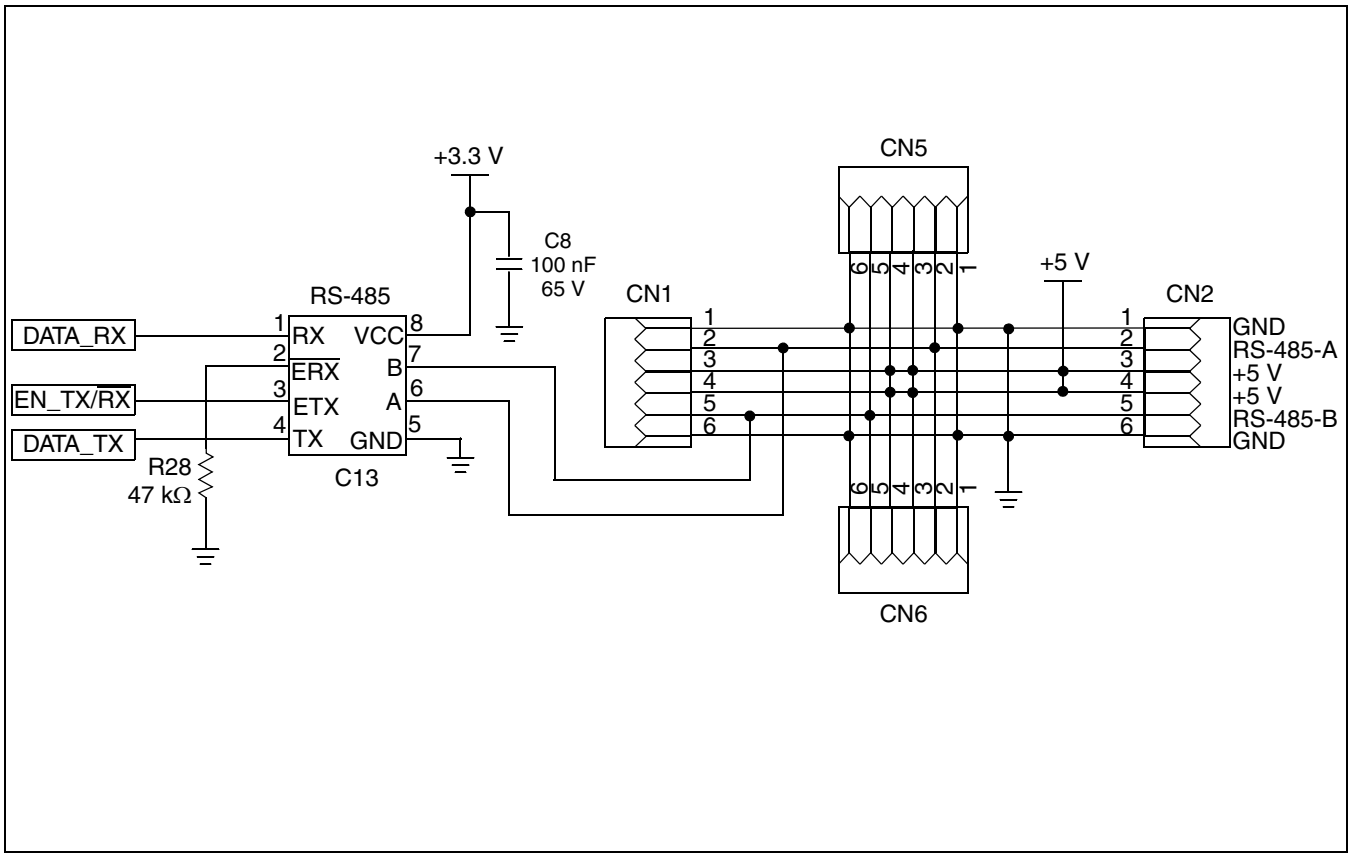


Figure 12. RS-485 Transceiver and Connectors

3.3 Software Description

In this section we will discuss how the pixel board software is structured. Block diagrams and description of state machines are provided to explain the code. First a flow chart of the software is presented to illustrate pixel board functionality, and then the different software modules are described.

3.3.1 Functional Description

Figure 13 depicts a flow chart of the software.

1. The MC56F8006 applies the initial settings to the system (ADC, PWM, GPIO, SCI and TIMER).
2. It blinks the LED once to indicate it has started, and then waits for a message from the network master.
3. If it receives a message, the DSC checks whether it is valid, and then stores or updates the data of the color of the LED according to the command received.
4. If the timer overflows, the DSC checks whether there was any change in the color of LED in comparison to the standard value previously received from the master. If there was variation, the DSC makes the adjustments needed to adjust. If there was no variation, the microcontroller returns to waiting for a message from the master.

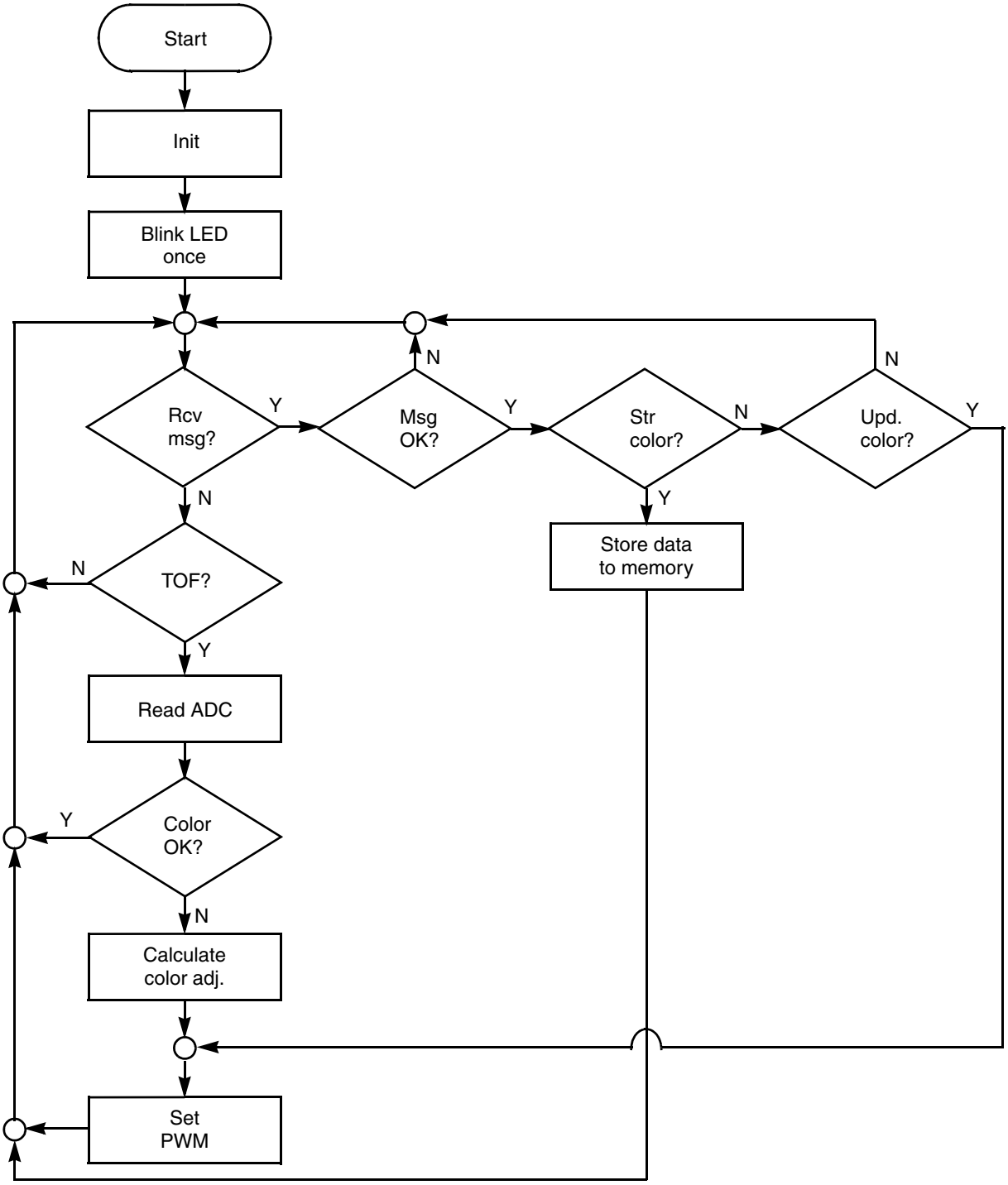


Figure 13. Software Flow Chart

To implement the flow chart, the software is broken down into six different modules, as shown in Figure 14. Each different module represents a C language source file and its header file. Low level routines access MC56F8006 peripherals such as SCI, analog-to-digital converter, and pulse width modulation,

represented by RS-485, ADC, and PWM on the diagram. Control functions are implemented on the communication protocol and color manager files. Finally, the application manager calls the different control functions.

The IDE used to develop DSC software was CodeWarrior 8.2 for 56800/E Digital Signal Controllers.

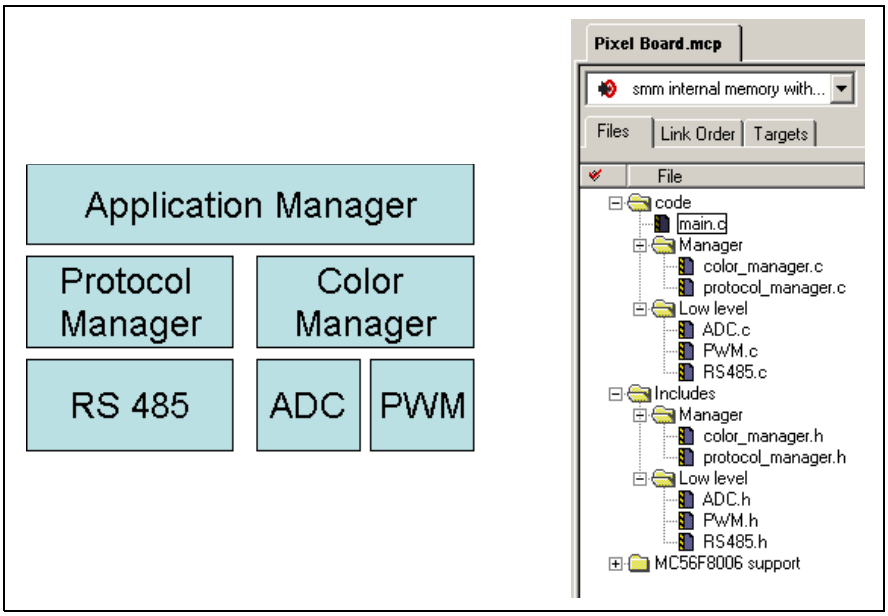


Figure 14. Software Architecture and Software Files on CodeWarrior

Next, each of these modules is explained in more detail.

3.3.2 Application Manager (main.c)

Application manager is implemented on the “main.c” file. Its main role is to initialize the system and to call the different control functions. Pseudo-code for the application manager is shown below:

```
//init DSC and peripherals
        init_functions()

 for(;;){
      watch_dog()
      color_manager()
      protocol_manager()
}
```

3.3.3 Protocol Manager (protocol_manager.c)

The Protocol Manager state machine is illustrated in Figure 15.

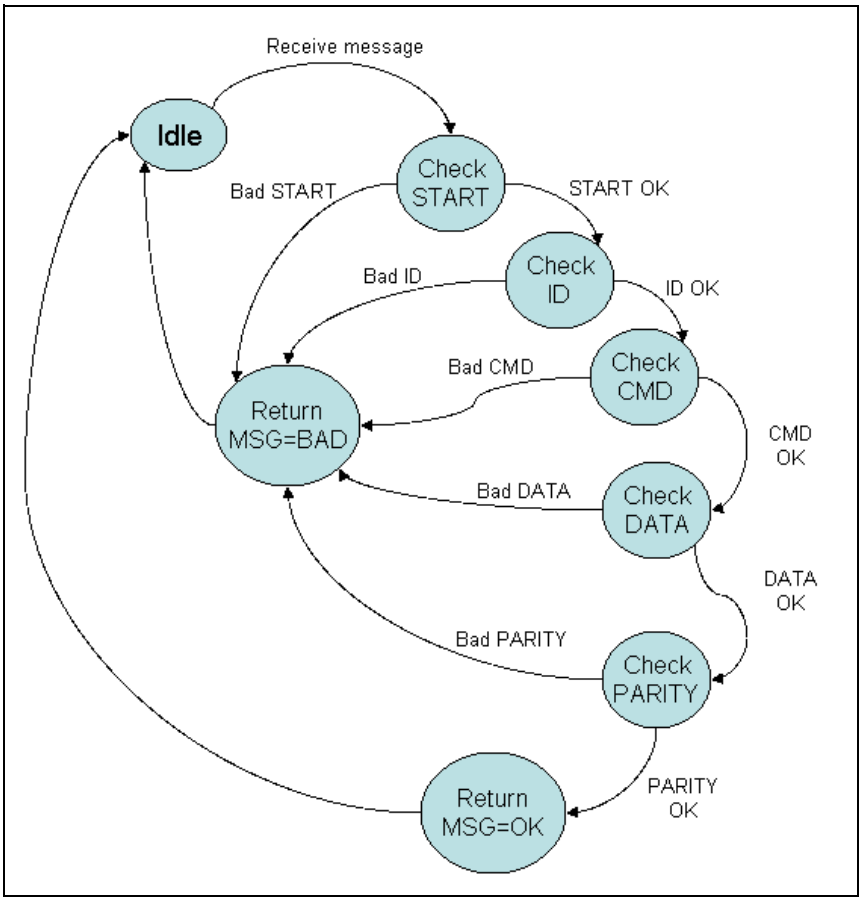


Figure 15. Protocol Manager State Machine

The protocol manager state machine works in this way:

The microcontroller is waiting for a message.

If it receives a message, the microcontroller checks whether the characters of START, ID, CMD, DATA, and PARITY are valid, and then returns this check information.

The message frame is structured like this:

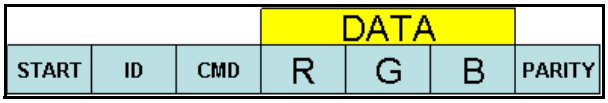


Figure 16. Message Frame

- START — Initial frame character, represented by @ (0x40). This block has a size of one byte.
- ID — ID or slave address. This block has a size of one byte, and can address up to 256 slaves.
- CMD — Command to be executed: store or update color. This block has a size of one byte.
- DATA — Contains details of the colors of the LED (RGB). This block has a size of three bytes.
- PARITY — Checks whether the message was corrupted during transmission. This block has a size of one byte.

The message frame has a total size of seven bytes.

The code snippet below shows the pseudo-code to implement the first two states of the machine. For the complete implementation, refer to the function “protocol_task” on “protocol_manager.c”.

```
//Protocol State Machine
Switch (state)
     Case Idle:
         If SCI_buffer_flag != Empty
             State = start_byte
     Break
     Case start_byte:
         If SCI_buffer[data] == '@'
             State = check_address
         Else
             State = idle
     break
// refer to the protocol_task on the code for the
//complete implementation
```

3.3.4 Color Manager (color_manager.c)

The Color Manager state machine is illustrated in Figure 17.

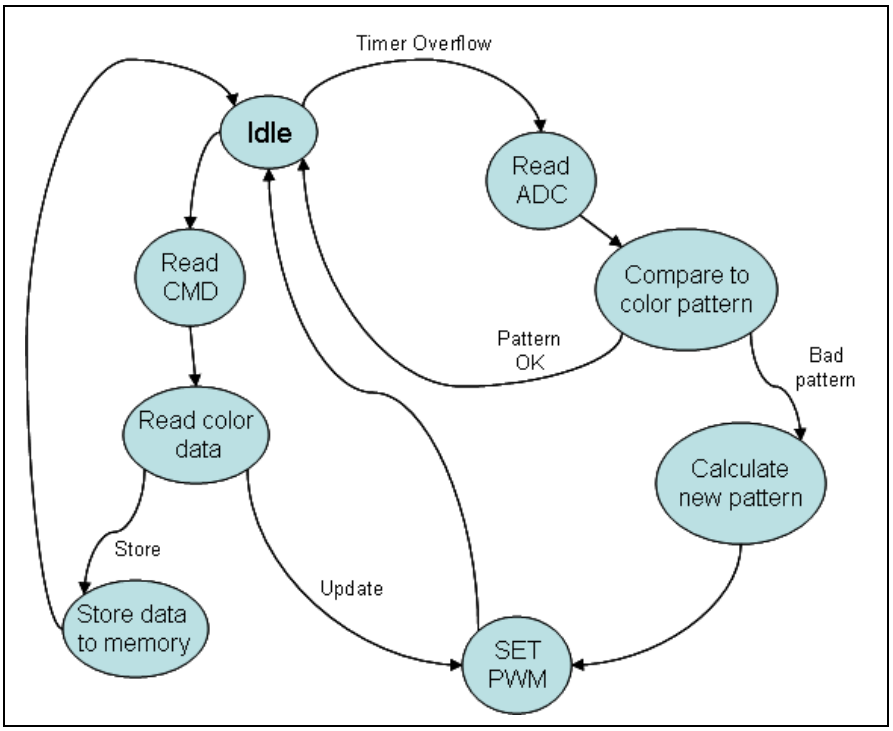


Figure 17. Color Manager State Machine

3.3.5 Color Manager State Machine

1. The DSC waits for an event to occur.

2. If the timer overflows, the DSC reads the ADC and compares the reading with the default color received previously from the master. If there is any difference, the MC56F8006 adjusts the color. Otherwise the microcontroller returns to waiting for an event to occur.
3. If it receives a message, the DSC can store the data of the color in memory, or update the color of the LED. These actions depend on the command received in the message.

For more information refer to function “color_task()” in color_manager.c file.

3.3.6 Low-Level Functions (RS-485, ADC, PWM)

The three low-level functions interface with I/O pins and peripherals. They communicate with control algorithms by global variables and with callback functions.

4 Gateway Board Overview

The gateway boards provide a way for the end-user to program the modular pixel network, by allowing a connection between the LED boards to and from a PC.

4.1 Gateway Board Requirements

Gateway board main requirements are:

- Supplying power to the pixel board network
- Hosting the master of the high-speed communication pixel board network
- Enabling communication with a PC or external network

An example of the Gateway board is the MC56F8006 evaluation board, MC56F8006DEMO (for expanded info, please visit the Freescale website at www.freescale.com). Two controllers are used on this board: 9S08JM (an 8-bit microcontroller) and the MC56F8006. The 9S08JM has an integrated USB controller that can be used to interface with a PC, whereas the MC56F8006, with its fast serial communication peripherals, can work as the master of the pixel board network. The missing parts are the power supply and the RS-485 transceiver; the former is not in the scope of this document, and an external supply was used on the proof of concept described in “Section 6, “Testing.” The RS-485 transceiver can be implemented using a simple ASIC.

The master of the RS-485 network on the gateway board is the MC56F8006. This processor is responsible for generating the master protocol, which then commands each slave address on the RS-485 network to change its color according to the programmed sequence informed by the 9S08JM and consequently by the computer.

Regarding the PC connection, there is the possibility of using Ethernet communication instead of USB, by using another microcontroller which contains Ethernet and SPI modules integrated. There are several instances of such a device in the ColdFire family which could be used instead of the 9S08JM.

4.2 Hardware Description

As described above, the MC56F8006 evaluation board will be used as the gateway board. The main components of the Gateway module are a microcontroller, the 9S08JM, the MC56F8006, and the RS-485 transceiver, which was added externally. These components are responsible for establishing communications between the computer and the pixel board network. The 9S08JM contains an internal USB module which performs the USB communication with the external computer. Both the 9S08JM and the MC56F8006 contain IIC, SPI, and SCI modules, which can used as a bridge between the USB and RS-485.

Hardware details can be found in the MC56F8006DEMO evaluation board documentation on the Freescale website.

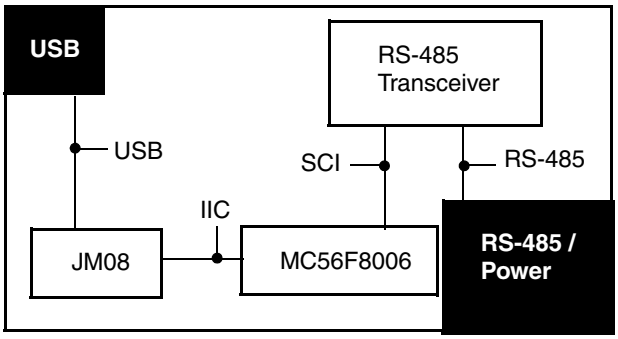


Figure 18. Gateway Board Functional Diagram

Ideally, SPI should be used due to its faster communication rates over IIC. However, to reuse MC56F8006DEMO hardware as much as possible, IIC was used on this proof of concept. As can be observed in Figure 19, MC56F8006 ports PB4 and PA6 were connected to PTC1 and PTC0 respectively on 9S08JM.

The MC9S08JM60 has two serial peripheral interfaces, SPI modules, both of which are used on the MC56F8006DEMO. One is used to program the MC56F8006 emulating a JTAG; the second is used to control the regulator that supplies power to the DSC. Thus the inter-integrated circuit module, IIC, is used to connect MC9S08JM60 and MC56F8006.

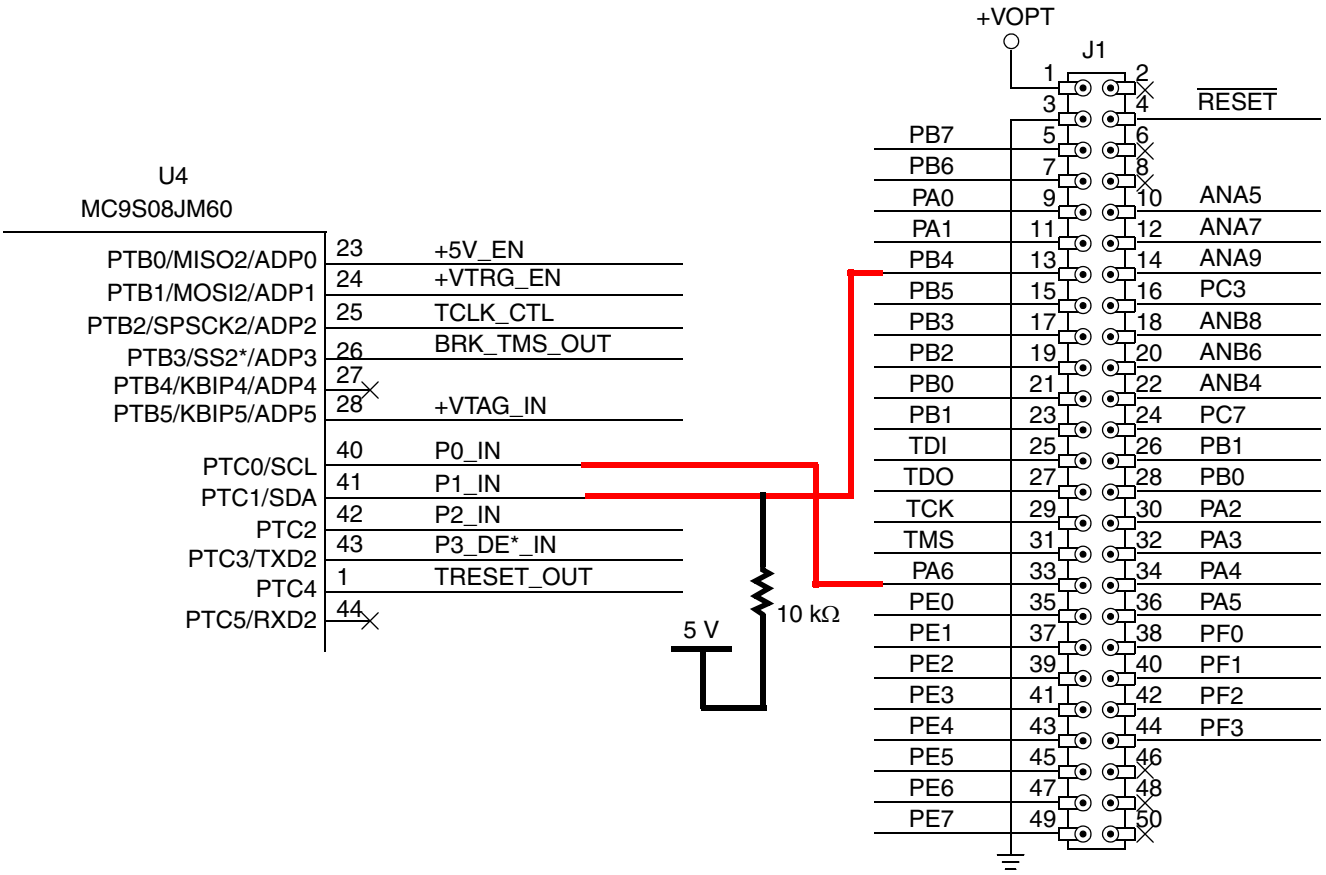


Figure 19. MC56F8006DEMO IIC Connection

4.3 Software Description

This section is split in two parts: the first explains the software running on 9S08JM and the second describes MC56F8006 software.

4.3.1 Gateway MCU

The software of the gateway MCU can be seen as a communication bridge that takes inputs from a computer and sends them to the master device of the RS-485 network. The universal serial bus (USB) interface is used to connect the computer to the MC9S08JM60 MCU, and the inter-integrated circuit module (IIC) is used to connect the MC9S08JM60 to the MC56F8006.

Although the embedded USB 2.0 module of the MC9S08JM60 incorporates several parts of the USB protocol such as physical signaling and data filters, it is still necessary that software take control of these layers. This is called a USB stack. It takes care of transactions like USB module configuration, USB enumeration, transaction type, handling endpoints, and sending/receiving data. The stack allows a simpler operation after configuration has been done, providing faster application development. The gateway MCU software was developed based on a CDC terminal example, provided on USB-Lite by CMS stack for MC9S08JM60 devices. More details about USB stack usage can be found in Freescale application note

AN3565, “USB and Using the CMX USB Stack with 9S08JM Devices,” available on the Freescale website.

The gateway MCU software is a command-line interpreter for lines typed on a computer terminal. The software used on the computer can be any serial terminal utility such as HyperTerminal. To make control of the pixel boards simpler for the end user, a simple serial application in C# was developed. It is able to control a matrix of up to four pixel boards.

For more information about using the serial application, please refer to Section 6, “Testing.”

4.3.2 MC56F8006 Network Master Software

Figure 20 is a flowchart of the software that will be running on the MC56F8006 gateway board.

When the gateway is turned on, it will first initialize the MC56F8006 peripherals (timers, GPIO’s, IIC and SCI modules) to work accordingly. Then it waits for a command from the PC. If there is no command from the PC, it will command the pixel boards connected on the network to blink according to a default initialization pattern. When a command is received from the PC, it will first interpret this command and depending on the command it will update the pixel boards on RS-485 network. Different commands can be implemented for the gateway device, but there is a possibility of implementing several functionalities for this device. To test the application, a single command was implemented to light an LED as a certain color.

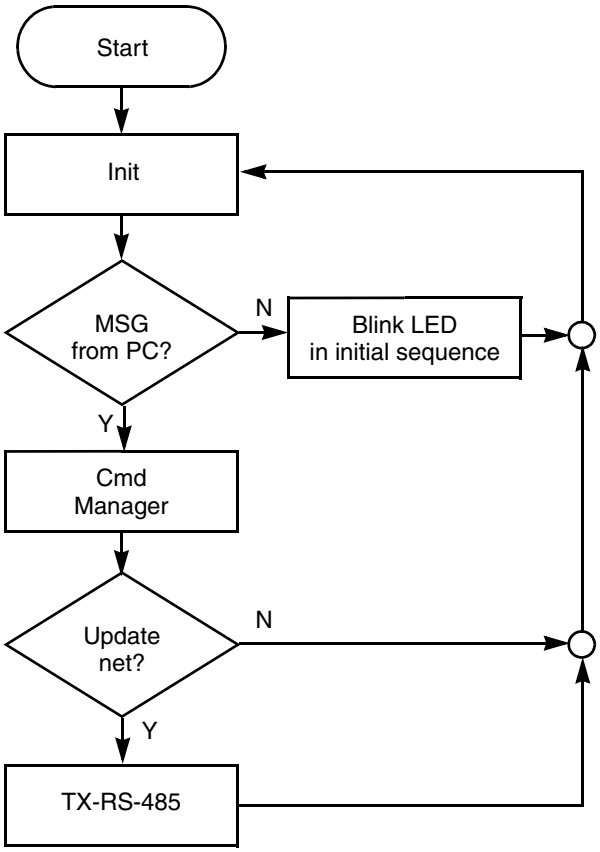


Figure 20. Gateway Board Network Master Flow Chart

As with the pixel board software, the gateway board MC56F8006 can also be implemented on different modules as shown in Figure 21. As can be seen, five different software blocks are used. IIC and RS-485 implement low-level communication with the MC56F8006 peripherals to send and receive data over IIC and SCI respectively. PC manager interprets data coming from the MC9S08JM60, while the network manager sends and receives messages to and from the pixel boards on the network, according to a known protocol. Finally, the application manager manages the execution of the software by calling the different tasks.

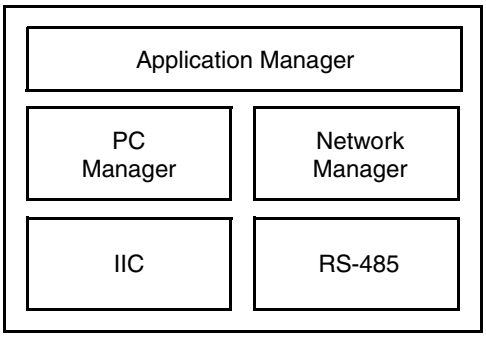


Figure 21. Gateway Board Software Architecture

5 Conclusion

In this document we have presented an implementation example of a modular high-brightness LED matrix and its network master. This serves as a reference for the use of different peripherals on the MC56F8006, such as the PWMs, SCI on high speed data rates, and ADC. It also serves as a starting point for some high-brightness LED applications with the MC56F8006 as the main controller.

6 Testing

To test and validate the concept described in this document, some pixel board prototypes were built. To test the application a small matrix of four pixel boards was connected to a gateway board plugged into a PC. Software running on the PC sends information to control the LEDs on the matrix. The aim of this section is to describe the tests performed, providing a real application example.

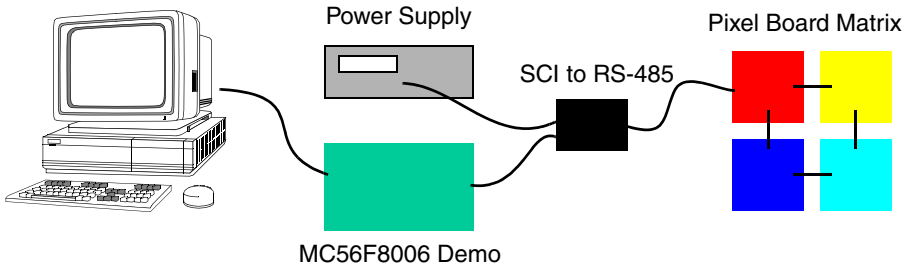


Figure 22. System Setup

6.1 PC Software

A simple C# program was created to send commands over the USB to the MC56F8006 board. Using this program the user can send messages to the four pixel boards independently, lighting them up with different colors.

These steps show how to communicate with the gateway board and make it activate the LED on the pixel board.

1. Connect the gateway board to the computer using a USB cable. Your computer will recognize it as a CDC device and will begin installation. Some interaction is needed to provide the USB driver. When prompted by the computer, specify a location and provide the “HCS08JMxx.inf” driver located on the software counterpart of this application note. When hardware installation is completed, the gateway board will be recognized as a CDC device with an assigned virtual COM port.
2. Verify the assigned virtual COM port for your board in the device manager under “Ports (COM & LPT)”.
3. Open the test program.
4. Set the port assigned to your board.
5. Set baud rate to 9600.
6. Open COM port.
7. Select the desired color by entering numbers from 0 to 255 on the red, green, and blue text boxes.
8. Use the “Press to Change Color” button to verify the selected color.
9. Press the “Send to Pixel [xx]” button to activate the desired pixel board LED.
10. Repeat the steps 7 through 9 for the other pixel boards.

By clicking the send to pixel button, a message will be sent via USB to the MC9S08JM60.

The frame sent will have this format:

- Frame Init: “start” (5 bytes)
- Address: The address of the LED to be lit (1 byte)
- Command: “L” (1 byte)
- RGB: red, green, and blue values (3 bytes)
- Frame End: “#” (1 byte)

Below is the code that runs when the “Send to Pixel” button click event is detected:

```
private void send(byte add, byte red, byte green, byte blue)
        {
            if (serialPort1.IsOpen)
            {
                serialPort1.Write("start "); // Frame init
              serialPort1.Write(new byte[] { add, Convert.ToByte('L'), red, green, blue,
Convert.ToByte('#')}, 0, 6); // Address, “L”, R, G, B and ‘#’
                serialPort1.Write("\n\r");
            }
            else
            {
```

```
                MessageBox.Show("Please Open the Serial Port");
            }
        }
```

Figure 23 shows the user interface of the test application that runs on the PC.

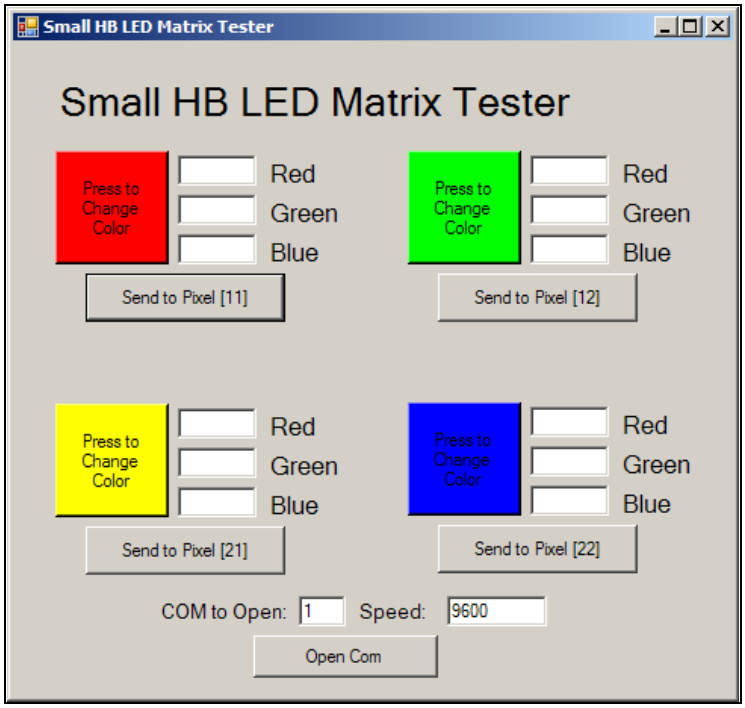


Figure 23. PC Test Application Screen

6.2 Gateway Board

As mentioned in Section 4, "Gateway Board Overview," the gateway board used to test the system is an MC56F8006DEMO that is connected to hardware that performs SCI to RS-485 translation.

The gateway board receives information from the PC on the USB controller of the MC9S08JM60. This MCU forwards the information though IIC to the master of the RS-485 pixel network, the MC56F8006. The DSC processes the information and sends data over the SCI to the pixel network. Before reaching the matrix the transmission signal is converted to RS-485 levels.

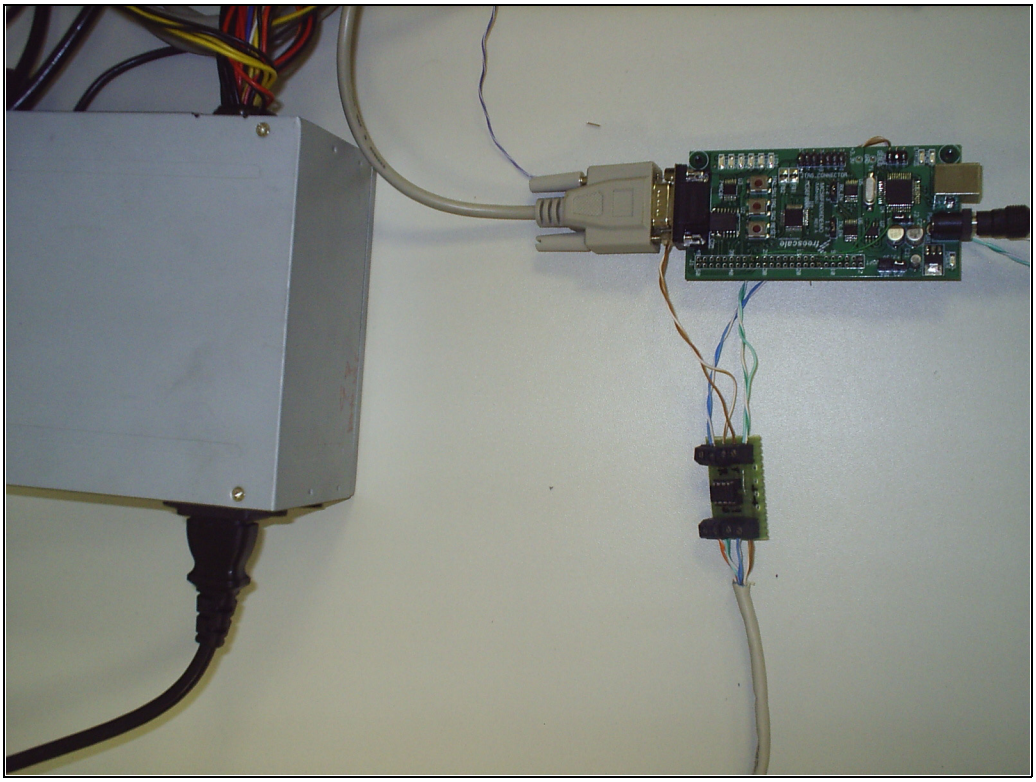


Figure 24. Gateway Board

6.3 LED Matrix

Figure 25 shows the set of pixel boards built to test the system. The matrix receives commands from the network master over the RS-485. The signals are converted and reach the DSC SCI port. Each pixel is addressed individually to update the color it is displaying.

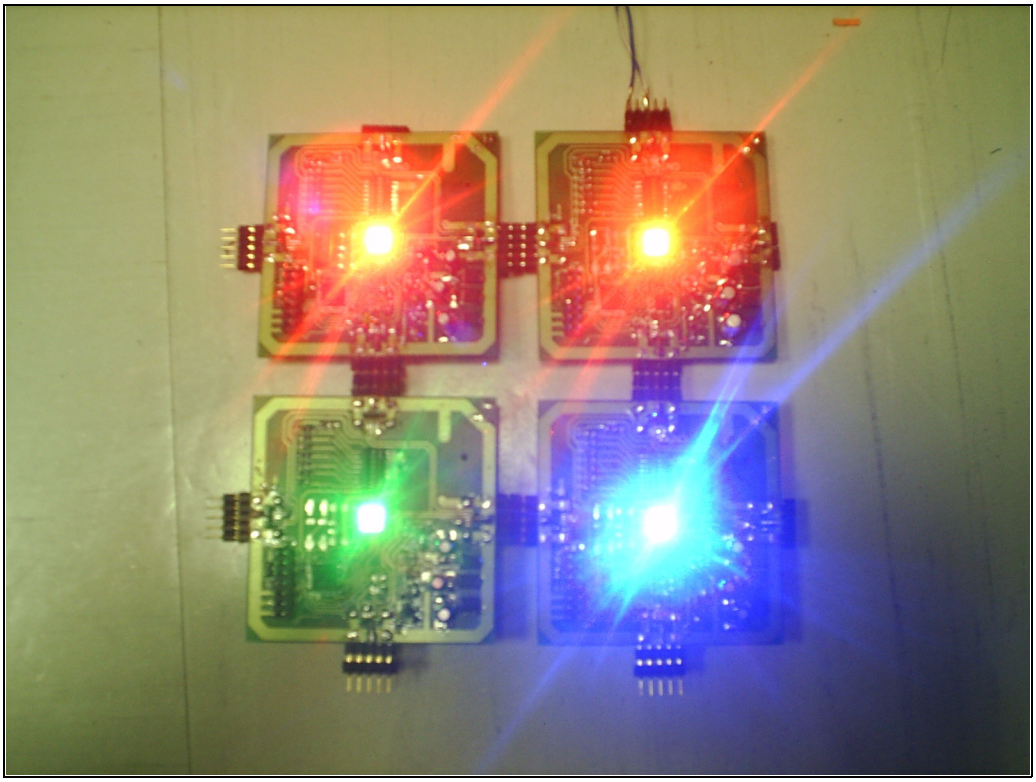


Figure 25. Pixel Matrix

As explained in Section 3, “Pixel Board Overview,” the MC56F8006 uses the PWM to drive a buck converter circuit that supplies current to the high-brightness LED. To control the color displayed, a shunt resistor is used to feedback the LED current into the DSC’s ADC. Figure 26 shows both the PWM output on the MC56F8006 pin and the signal after it has passed through the buck converter and is fed to the ADC pin.

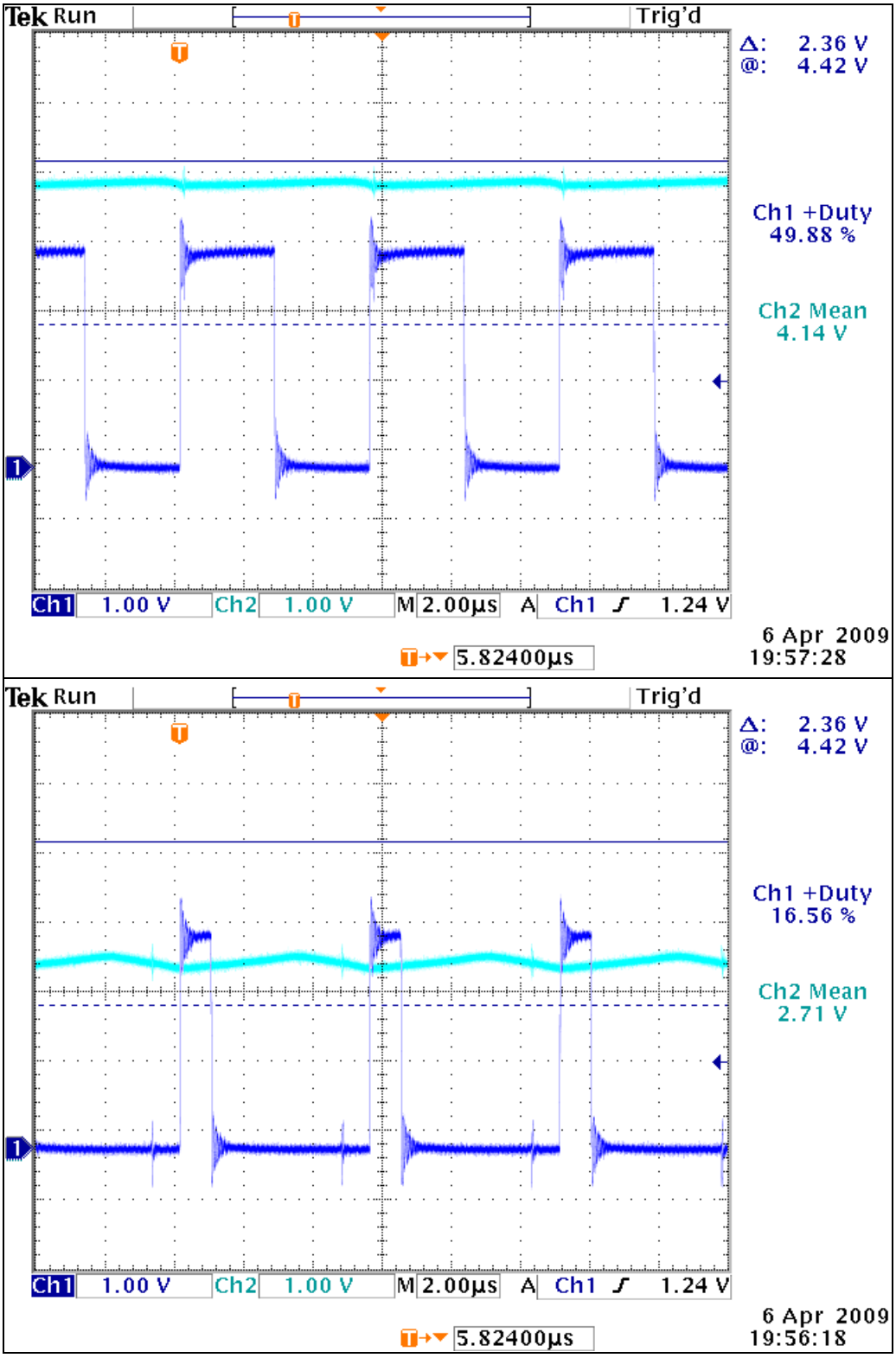


Figure 26. PWM and LED Current Signals

6.4 Making it Work

For illustrative purposes, Figure 27 shows the prototypes working on the test bench.

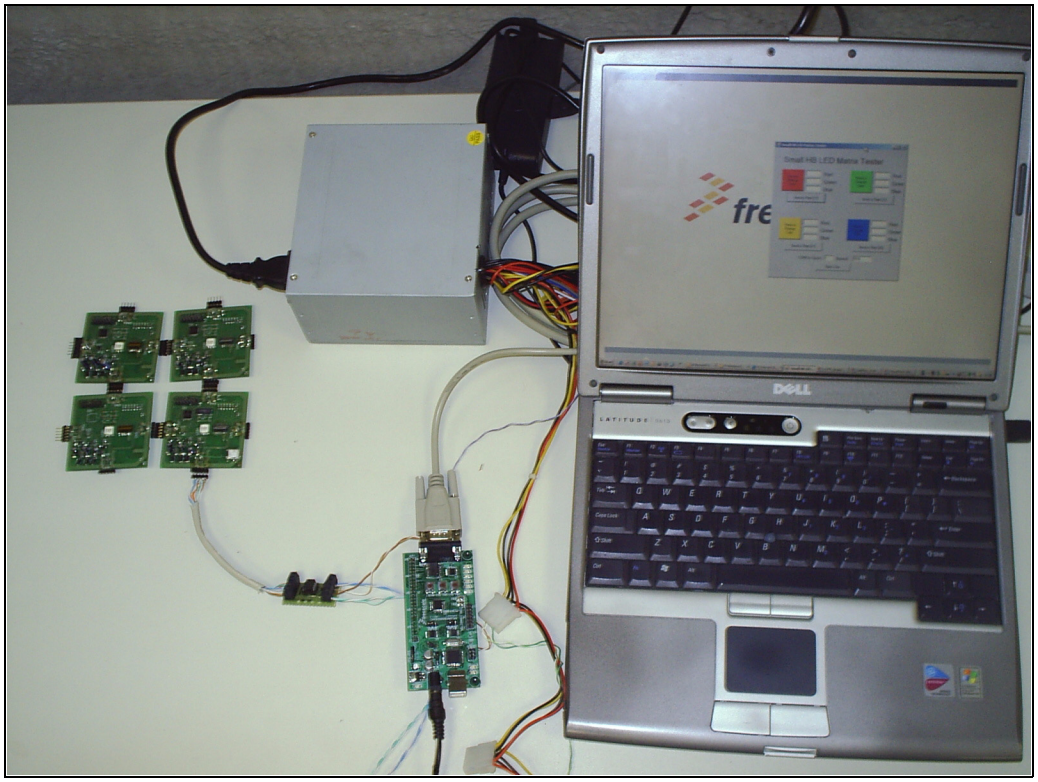
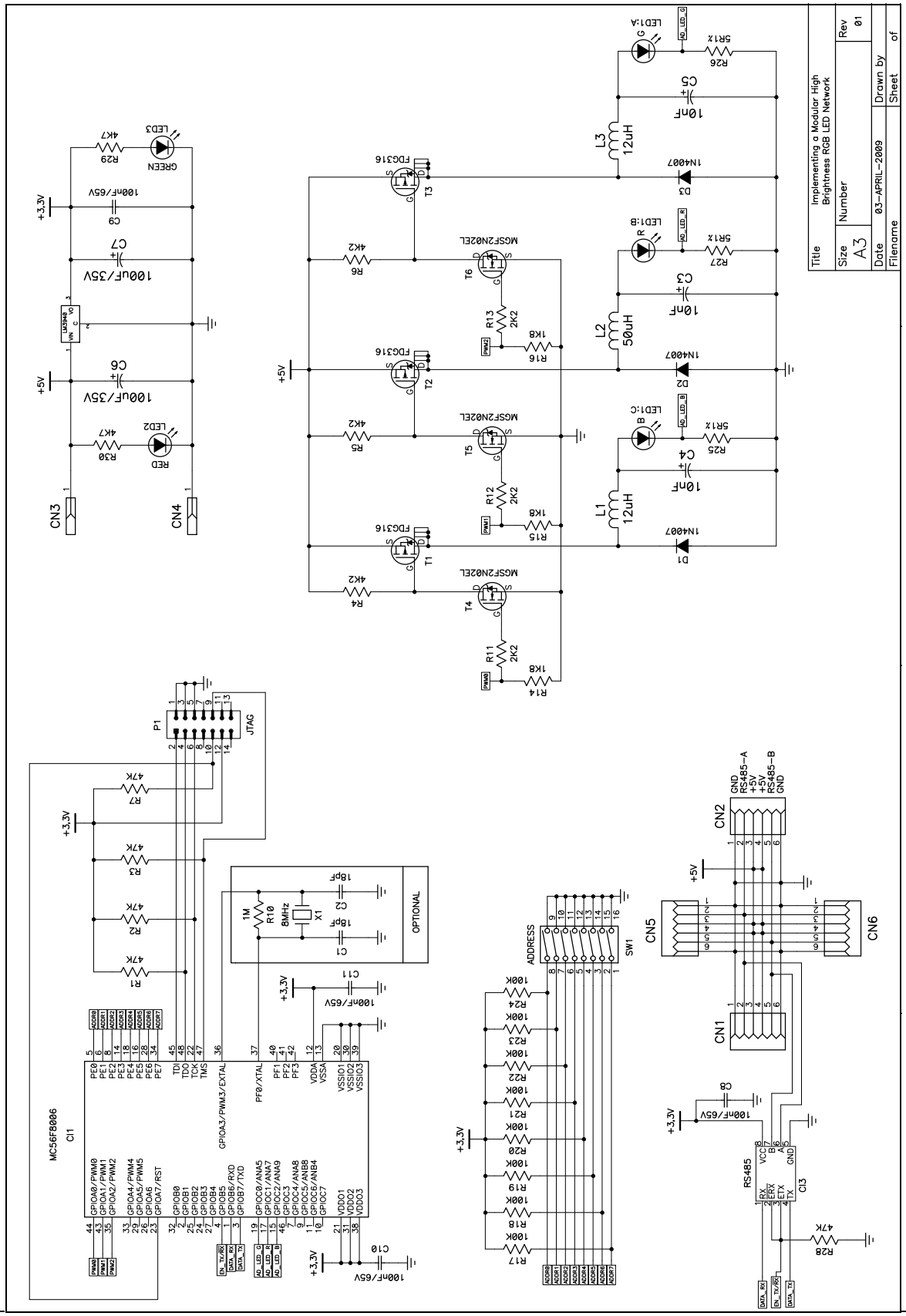


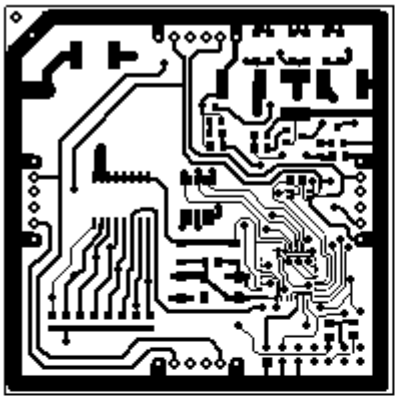
Figure 27. Test Bench

Appendix A Pixel Board Schematics

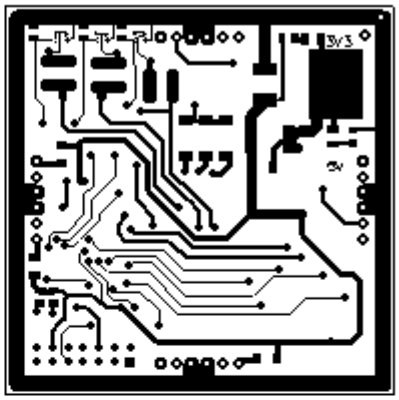


Appendix B Pixel Board Layout

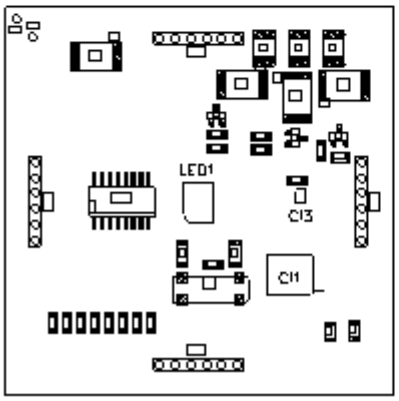
Top Layer



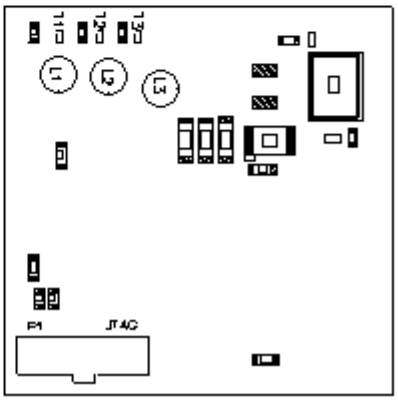
Bottom Layer



Top Placement



Bottom Placement



Appendix C Pixel Board Software Reference

This appendix lists all functions and variables on pixel board firmware. This documentation was generated using “Doxygen.”

C.1 adc.c—File Reference

```
#include "util.h"
#include "adc.h"
#include "pwm.h"
```

C.1.1 Functions

- void adc_init ()
  Initializes ADC.
  Set all ADC related functionality.
- uint16_t adc_get_color_value (adc_colors color)
  Returns an ADC reads average for a given color.
  — Parameters:
  color: the color for which the average should be calculated.
  — Returns
  color average that depends on buffer size.
  Sums all ADC samples taken and returns their average.
- void adc_isr (void)
  ADC convertion complete interrupt.
  Stores ADC samples and color buffers.

C.1.2 Variables

- uint16_t adc_green_buffer [ADC_BUFFER_SIZE]
  ADC Data buffer for the green color.
- uint16_t adc_red_buffer [ADC_BUFFER_SIZE]
  ADC Data buffer for the red color.
- uint16_t adc_blue_buffer [ADC_BUFFER_SIZE]
  Data buffer for the blue color.
- adc_state_machine state_machine
  ADC task state machine controller.
- uint8_t adc_count
  Pointer for ADC data buffers.

C.2 adc.h—File Reference

```
#include "mc56f8006.h"
#include "cpu.h"
```

C.2.1 Defines

- #define ADC_GREEN_CHANNEL ADC0_ADCSC1A_ADCH_2 | ADC0_ADCSC1A_ADCH_0
- #define ADC_RED_CHANNEL ADC0_ADCSC1A_ADCH_2|ADC0_ADCSC1A_ADCH_1 | ADC0_ADCSC1A_ADCH_0
- #define ADC_BLUE_CHANNEL ADC0_ADCSC1A_ADCH_3|ADC0_ADCSC1A_ADCH_0
- #define ADC_BUFFER_SIZE 16

C.2.2 Enumerations

- enum adc_state_machine { READ_GREEN, READ_RED, READ_BLUE }
  Enumerator:
  — READ_GREEN
  — READ_RED
  — READ_BLUE
- enum **adc_colors** { ADC_GREEN, ADC_RED, ADC_BLUE }
  Enumerator:
  — ADC_GREEN
  — ADC_RED
  — ADC_BLUE

C.2.3 Functions

- void adc_init (void)
  Initializes ADC.
  Set all ADC related functionality.
- void adc_isr (void)
  ADC convertion complete interrupt.
  Stores ADC samples and color buffers.
- uint16_t adc_get_color_value (adc_colors)
  Returns an ADC reads average for a given color.
  — Parameters
  color: the color for which the average should be calculated.
  — Returns
  color average that depends on buffer size.

Sums all ADC samples taken and returns their average.

C.3 color_manager.c—File Reference

```
#include "util.h"
#include "adc.h"
#include "color_manager.h"
#include "pwm.h"
```

C.3.1 Functions

- void colorm_init (void)
  Initializes color manager local variables.
  This function should be called before "colorm_task()".
- void colorm_task (void)
  Color Manager state machine implementation.
  This task controls the color intensity that should be lit by the High Power-LED. It also receives new color requests by functions that call "colorm_rcv_value".
- void colorm_rcv_value (uint8_t r, uint8_t g, uint8_t b, CM_STATUS cmd)
  Receives commands and color values.
  — Parameters:
  r the red component to be set or stored.
  g the green component to be set or stored.
  b the blue component to be set or stored.
  cmd the command to be executed, set or store.
  Receives desired color values and commands from other functions.

C.3.2 Variables

- int16_t cm_color_red
  Control variable to set the LED color.
- int16_t cm_color_green
  Control variable to set the LED color.
- int16_t cm_color_blue
  Control variable to set the LED color.
- uint16_t cm_desired_red
  Store color value received from protocol manager (SCI).
- uint16_t cm_desired_green
  Store color value received from protocol manager (SCI).
- uint16_t cm_desired_blue
  Store color value received from protocol manager (SCI).

- uint16_t cm_count
  Timeout decremented each time IDLE state runs.
- CM_STATUS cm_status
  Store the command to be performed by color manager task.
- CM_STATE_MACHINE colorm_state_machine
  Color manager state machine variable.

C.4 color_manager.h—File Reference

C.4.1 Defines

- #define CM_INIT_RED 100
  Initial Red Led Intensity
- #define CM_INIT_GREEN 100
  Initial Green Led Intensity
- #define CM_INIT_BLUE 100
  Initial Blue Led Intensity
- #define CM_LED_MIN 100
  Minimum LED acceptable value to avoid blinking on low intensity values
- #define CM_TIMEOUT 100
  Number of interactions on IDLE state before controlling LED intensity
- #define CM_ADC_SCALE 5
  CM_ADC_SCALE is calculated based on ADC reading 1 V (1240) divided by maximum command (255). 1 V ADC read means 200 mA on the LED, the maximum allowed current.

C.4.2 Enumerations

- enum CM_STATE_MACHINE { CM_IDLE, CM_ADJ_PWM, CM_READ_ADC, CM_READ_CMD }
  - Enumerator:
    CM_IDLE
    CM_ADJ_PWM
    CM_READ_ADC
    CM_READ_CMD
- enum CM_STATUS { CM_CMD_NONE, CM_CMD_LIT, CM_CMD_STORE }
  - Enumerator:
    CM_CMD_NONE
    CM_CMD_LIT
    CM_CMD_STORE

C.4.3 Functions

- void colorm_init (void)
- void colorm_task (void)
- void colorm_rcv_value (uint8_t, uint8_t, uint8_t, CM_STATUS)
  Receives commands and color values.
  — Parameters:
    r the red component to be set or stored.
    g the green component to be set or stored.
    b the blue component to be set or stored.
    cmd the command to be executed, set or store.
  Receives desired color values and commands from other functions.

C.5 gpio.c—File Reference

```
#include "cpu.h"
#include "util.h"
#include "gpio.h"
```

C.5.1 Functions

- void gpio_init (void)
  Initializes GPIOs.
  GPIO PORT E state. Reads and returns PORT E pin states.
- uint8_t gpio_get_address (void)
  Reads GPIOs.
  Set pin muxing and data direction.

C.6 gpio.h—File Reference

C.6.1 Functions

- uint8_t gpio_get_address (void)
  Reads GPIOs.
  GPIO PORT E state. Reads and returns PORT E pin states.
- void gpio_init (void)
  Initializes GPIOs.
  Set pin muxing and data direction.

C.7 protocol_manager.c—File Reference

```
#include "util.h"
#include "protocol_manager.h"
```

```
#include "sci.h"
#include "gpio.h"
#include "pwm.h"
#include "color_manager.h"
```

C.7.1 Functions

- void protocolm_init (void)

  Initializes protocol manager local variables.

  This function should be called before "protocolm_task()".
- void protocolm_task (void)

  Protocol Manager state machine implementation.

  This task receives data from SCI using sci_read_byte(), if a valid frame is received color manager is updated. Five States are implemented, new commands can be added on "PM_READ_CMD" state.

C.7.2 Variables

- uint8_t red_value

  Desired red color value received from network master on SCI.
- uint8_t green_value

  Desired green color value received from network master on SCI.
- uint8_t blue_value

  Desired blue color value received from network master on SCI.
- PM_STATE_MACHINE protocolm_state_machine

  State machine variable.
- uint8_t flag_color

  Auxiliary variable to store color data from SCI.

C.8 protocol_manager.h—File Reference

C.8.1 Enumerations

- enum PM_STATE_MACHINE { PM_IDLE, PM_READ_START, PM_READ_ADDRESS, PM_READ_CMD, PM_READ_COLORS, PM_READ_END }
  - Enumerator:

  PM_IDLE

  PM_READ_START

  PM_READ_ADDRESS

  PM_READ_CMD

  PM_READ_COLORS

PM_READ_END

C.8.2 Functions

- void protocolm_init (void)
- void protocolm_task (void)

C.9 pwm.c—File Reference

```
#include "cpu.h"
#include "util.h"
#include "pwm.h"
```

C.9.1 Functions

- void pwm_init (void)
  Initialize PWMs.
  Set all PWM related functionality.
- void pwm_set_values (int16_t red, int16_t green, int16_t blue)
  Set PWM values.
  — Parameters:
  red—red color pwm value
  green—green color pwm value
  blue—blue color pwm value
  Receive color values from other functions to set PWM duty cycle output.

C.10 pwm.h—File Reference

C.10.1 Defines

- #define PWM_RED_PRESCALER 15
- #define PWM_GREEN_PRESCALER 10
- #define PWM_BLUE_PRESCALER 10

C.10.2 Functions

- void pwm_init (void)
  Initializes PWMs.
  Sets all PWM related functionality.
- void pwm_set_values (int16_t red, int16_t green, int16_t blue)
  Set PWM values.
  — Parameters:

red—red color pwm value

green—green color pwm value

blue—blue color pwm value

Receive color values from other functions to set PWM duty cycle output.

C.11 sci.c—File Reference

```
#include "util.h"
#include "sci.h"
#include "MC56f8006.h"
```

C.11.1 Typedefs

- typedef unsigned char byte

C.11.2 Functions

- void sci_init (uint32_t baudrate)

  Initializes Sci.

  — Parameters:

  baudrate value for SCI.

  Set all SCI related functionality.
- uint8_t sci_byte_received (void)

  Returns if a byte was received by the SCI.

  — Returns:

  1 for byte received, 0 otherwise.

  Tests buffer out and in pointers to check if data was received.
- int8_t sci_read_byte (void)

  Returns a byte that was received from the SCI.

  — Returns:

  the next byte on the rx buffer.

  Gets data from the RX Buffer and updates output pointer. Returns "-1" if no data was received. (A better error treatment should be implemented.)
- void sci_isr (void)

  SCI RX Interrupt, it puts rx data on the buffer.

  Stores Rx data on the buffer and updates pointers.

C.12 sci.h—File Reference

```
#include "mc56f8006.h"
```

C.12.1 Defines

- #define BSP_OSCILLATOR_FREQ 8000000L
- #define PLL_MUL 1L
- #define PLL_POSTSCALER (1<<0)
- #define SCI_CLOCK_HZ (BSP_OSCILLATOR_FREQ * PLL_MUL / PLL_POSTSCALER / 2)
- #define SCI_START '@'
  SCI symbol for communication protocol start frame.
- #define SCI_END '#'
  SCI symbol for communication protocol end frame.
- #define SCI_CMD_LIT 'L'
  SCI symbol for communication protocol Lit command.
- #define SCI_RX_BUFFER_SIZE 16
  SCI Rx buffer size.
- #define SCI_BAUDRATE_115k2 4400l
  Used to set SCI baud rate to 115,200 bps.
- #define SCI_BAUDRATE_6M 200000l
  Used to set SCI baud rate to 6 Mbps.

C.12.2 Functions

- void sci_init (uint32_t)
  Initializes Sci.
  — Parameters:
  baudrate value for SCI.
  Set all SCI related functionality.
- uint8_t sci_byte_received (void)
  Returns if a byte was received by the SCI.
  — Returns:
  1 for byte received, 0 otherwise.
  Tests buffer out and in pointers to check if data was received.
- int8_t sci_read_byte (void)
  Returns a byte that was received from the SCI.
  — Returns:
  the next byte on the rx buffer.
  Gets data from the RX Buffer and updates output pointer. Returns "-1" if no data was received. (A better error treatment should be implemented.)
- void sci_isr (void)
  SCI RX Interrupt, it puts rx data on the buffer.

Stores rx data on the buffer and updates pointers.

C.13 sys.c—File Reference

```
#include "util.h"
#include "sys.h"
```

C.13.1 Defines

- #define DisableWatchdog()

```
Value:SIM_PCE |= SIM_PCE_COP; \
   COP_CTRL &= ~COP_CTRL_CEN; \
   SIM_PCE &= ~SIM_PCE_COP
```

C.13.2 Functions

- void sys_enable_pwm_clk (void)
  Enable PWM clock gate.
- void sys_enable_pwm_sci_3x (void)
  Enable high speed clock to PWM and SCI.
- void sys_init (void)
  Enable PLL and other system init.

C.14 sys.h—File Reference

C.14.1 Functions

- void sys_enable_pwm_clk (void)
  Enable PWM clock gate.
- void sys_enable_pwm_sci_3x (void)
  Enable high speed clock to PWM and SCI.
- void sys_init (void)
  Enable PLL and other system init.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3815
Rev. 0
04/2009

freescale™
semiconductor