PANDORA PRODUCTS

Light Controller
WS2811 Lights

Jim Schimpf

Document Number: PAN-201502007
Revision Number: 0.3
30 March 2014

Pandora Products.
215 Uschak Road
Derry, PA 15627

Pandora Products.

215 Uschak Road

Derry, PA 15627

Phone: 724-539.1276

Email: jim.schimpf@gmail.com

Document Revision History

Version	Author	Description	Date
0.1	js	Initial Version	7-Feb-2011
0.2	js	Update the hardware design	29-Mar-2015
0.3	js	Correct typos	30-Mar-2015

Contents

List of Figures

1 Introduction

The WS2811 are networked LED's containing RED/GREEN/BLUE lamps that can have 8 bit brightness value. Any number of these lamps can be connected with power,ground and data wires in series. The details of the signaling and waveforms are given in [1]. The rest of this document describes a LPC1114 system used to run these lights.

1.1 WS2811 Lights

The WS2811 are a three LED (Red/Green/Blue) system with an integral processor. The lights are daisy-chained together with a +5 volt line, ground and a data line. Serial data is pumped into the data successively programs each lamp in the chain with 8 bits of data for the brightness of each of the three lamps in a module.

1.1.1 Data

The data is encoded as a form of pulse width encoding. Each is encoded as a high for a certain time, then a low on the data line for another fixed time. The patterns are shown below, the 0 bit has a short high followed by a longer low, the 1 bit has a longer high, followed by shorter low.

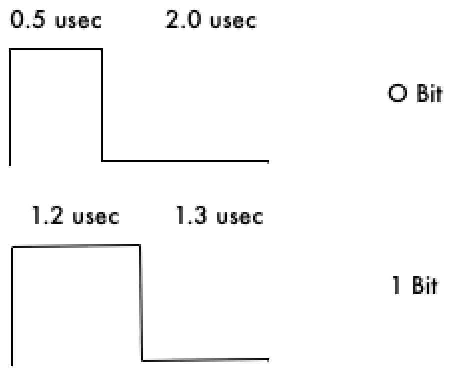

Figure 1: Bit pattern

This is the slower time, you can also send the same pattern but with all the times divided by 2. A single lamp takes 24 of these bits to program all 8 bits of the Red/Green and Blue lamps sent as most significant bit first RED-GREEN-BLUE. All these bits must be contigious. A gap of greater than 50 usec resets the whole string and you start over programming from the beginning. Also these times must be maintained within +/- 150 ns.

1.1.2 Generation

These times are barely obtainable with the LPC1114 at a 48 MHz clock. I had hoped to generate them via built in 32 or 16 bit timers in the chip but they do not have the resolution. To generate these I use code and have interrupts off and run the processor in bit banging mode.

```
//-------------------------------------------------------------
/*
    int send_word(unsigned int value)   - Clock out word

    INPUT:  value   24 bit RGB value

    OUTOUT  NONE
            Clock bits out MSB first

            Enter with line OUT_0 & leave with OUT_0
            Level shifter will invert before feeding
            lights.
*/
//-------------------------------------------------------------
static void send_word( unsigned int value )
{
    int mask = MASK;
```

```
        volatile int original = Chip_GPIO_GetPortValue(LPC_GPIO ,0);
        unsigned int bit =  0x1 << LAMP_BIT;
        unsigned int saved = 0;
        // Run till mask exhausted
        while( mask != 0 )
        {
            // Set bit OUT_1 at start
            original ^= bit;
            Chip_GPIO_SetPortValue(LPC_GPIO, 0, original);
            // Send 1 or 0 bit
            if( (value & mask) == 0  )
            {
            // O output (OUT_1 then OUT_0)
             ft(DELAY_T0H,saved);                      <-- Inline ASM delay
                original ^= bit; // Flip bit low
                Chip_GPIO_SetPortValue(LPC_GPIO, 0, original);
                ft(DELAY_T0L,saved);                      <-- Inline ASM delay
            }
            else
            {
            // 1 output (OUT_1 then OUT_0)
             ft(DELAY_T1H,saved);                    <-- Inline ASM delay
                original ^= bit; // Flip bit low
                Chip_GPIO_SetPortValue(LPC_GPIO, 0, original);
                ft(DELAY_T1L,saved);                    <-- Inline ASM delay
            }

            // Move mask to get next bit
            mask = mask >> 1;
        }
    }
```

The code is rather convential but the delay which is a spin loop is asm code so I have have the resolution needed. The spin code is:

```
    #define ft(delay,saved) __asm(\
    " mov %1,r0\n"\       <-- Save r0
    " mov r0,%0\n"\       <-- Use input count
    "$L_%=:  sub r0,#1\n"\ <-- Decrement r0
    " bne $L_%=\n"\       <-- Exit when 0
    "  mov r0,%1\n"\      <-- Restore r0
    ::"r" (delay),"r" (saved));
```

It is just code that enters saves r0 then uses the passed in value of r0 and decrements it to 0 and exits. Then you must use an oscilliscope to calibrate the waveforms to find the timing. The timing values that work are:

```
// 400 Khz timer values (for ft)
#define DELAY_T0H       2
#define DELAY_T0L       17
#define DELAY_T1H       9
#define DELAY_T1L       11
```

Originally this was written as an assembly code subroutine but it was found the subroutine overhead was too much so it was re-written as an inline assembly macro. This increases the code size a bit but the overhead of entering and exiting the routine is minimal.

1.1.3 Results

This shows the output of the above code generating a 0 and a 1 waveform for all bits.

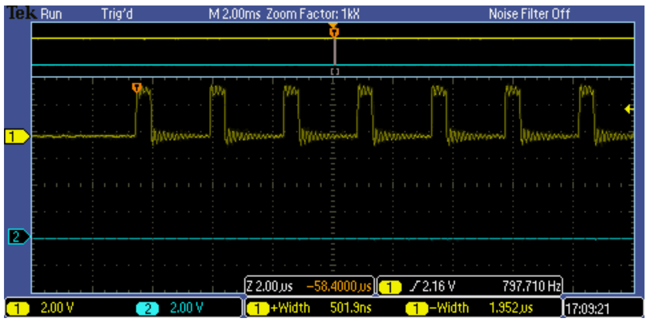

Figure 2: Zero Bit Waveform

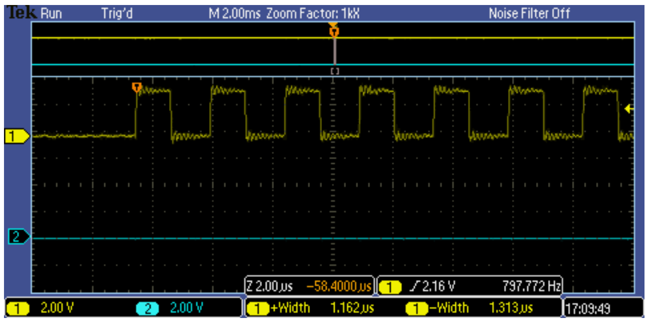

Figure 3: One Bit Waveform

Bit	HIGH	HIGH Measure	LOW	LOW Measure
0	0.5 us	0.502 us	1.2 us	1.952 us
1	1.2 us	1.162 us	1.3 us	1.313 us

Table 1: Waveform Measurements

As you can see from the + and - Width measurements are within the +/- 150 ns criteria

1.2 Light Driving Routine

1.2.1 Introduction

The routines in the bit banging routines (1.1 on page 1) do part of the job, output 24 bits for one light. Above that there have to be routines to output a whole string of lights and also update the pattern as necessary. The bit banging routine depends on running with interrupts disabled in total control of the machine.

1.2.2 Timer Setup

To run the lights the program sets up a 30 Hz timer interrupt and during that ISR we call the send_word() for each of the lights in the string. The timer is set up in the MyBoard_Init() routine as the program is started.

```
//------------------------------------------------------------------
    //  30 Hz timer
    /* Enable timer 1 clock */
    Chip_TIMER_Init(LPC_TIMER32_0);
    /* Timer rate is system clock rate */
    timerFreq = Chip_Clock_GetSystemClockRate();
    /* Timer setup for match and interrupt at 30 Hz rate */
    Chip_TIMER_Reset(LPC_TIMER32_0);
    Chip_TIMER_PrescaleSet(LPC_TIMER32_0, 2);
    Chip_TIMER_MatchEnableInt(LPC_TIMER32_0, 1);
    val = (timerFreq / 90); // 30 Hz tick (why 90 ???)
    Chip_TIMER_SetMatch(LPC_TIMER32_0, 1, val);
    Chip_TIMER_ResetOnMatchEnable(LPC_TIMER32_0, 1);
    Chip_TIMER_Enable(LPC_TIMER32_0);
```

Then before FreeRTOS is started we start the timer.

```
    // Start timer
Chip_TIMER_Reset(LPC_TIMER32_0);
NVIC_EnableIRQ(TIMER_32_0_IRQn);
// Start tasks running
vTaskStartScheduler();
```

1.2.3 Timer ISR

Once started the timer will call TIMER32_0_IRQHandler(). We have created a semaphore lampRun that the interrupt routine grabs each time it runs. Then any non-interrupt routine can try to grab this semaphore and then modify the light parameters. If a non-interrupt routine is modifying the parameters when the interrupt runs as you can see with the line:

```
    if( xSemaphoreTakeFromISR(lampRun,&higherTask ) == pdPASS )
```

The interrupt will check if this is the case and skip the update to the lamps on this cycle. The check is non-blocking so the interrupt will immediately return. If not blocked by the semaphore, it will run the lamplighter routine. The lamplighter() routine will read each of the lamp words and send them via send_word() to the lamps. Since this is run at interrupt level nothing will interfere with the timing on the bits and the C code in lamplighter() is fast enough to send each word with less than the 50 usec gap that would reset the string. Thus with the RTOS system we can have seamless user control of the lights while they are running.

```
    // TIMER interrupt routine
    void TIMER32_0_IRQHandler(void)
    {
        int higherTask;
        // Mark entrance & clear status
        Chip_GPIO_SetPinState(LPC_GPIO, 0, 7, 1);
        Chip_TIMER_ClearMatch(LPC_TIMER32_0, 1);

        // Run the lights if enabled
        if( xSemaphoreTakeFromISR(lampRun,&higherTask ) == pdPASS )
        {
            // Run the lamps
            lamplighter(LAMPS,lamps);

            // Release the semaphore

            xSemaphoreGiveFromISR(lampRun,&higherTask);
        }

        Chip_GPIO_SetPinState(LPC_GPIO, 0, 7, 0);
    }
```

2 Hardware

The board will be based on the LPC1114 28 pin DIP package. In addition it needs a level changing chip to boost the 3.3 V outputs of the ARM to 5 volts to run the lights. For this a 74HC14 hex inverter is used. All 6 parts of this chip are used because the WS2811 output is inverted twice so it has the correct polarity. In addition two other lines are brought out, these are SPI output SCLK and MOSI. This is becasue there is another light strip design that uses SPI control instead of the timing based waveforms of the WD2811. The system can also be used to control them.

The design assumes an external 5V supply that has enough capacity to supply the current for the lights and an on-board 3.3 V supply for the processor.

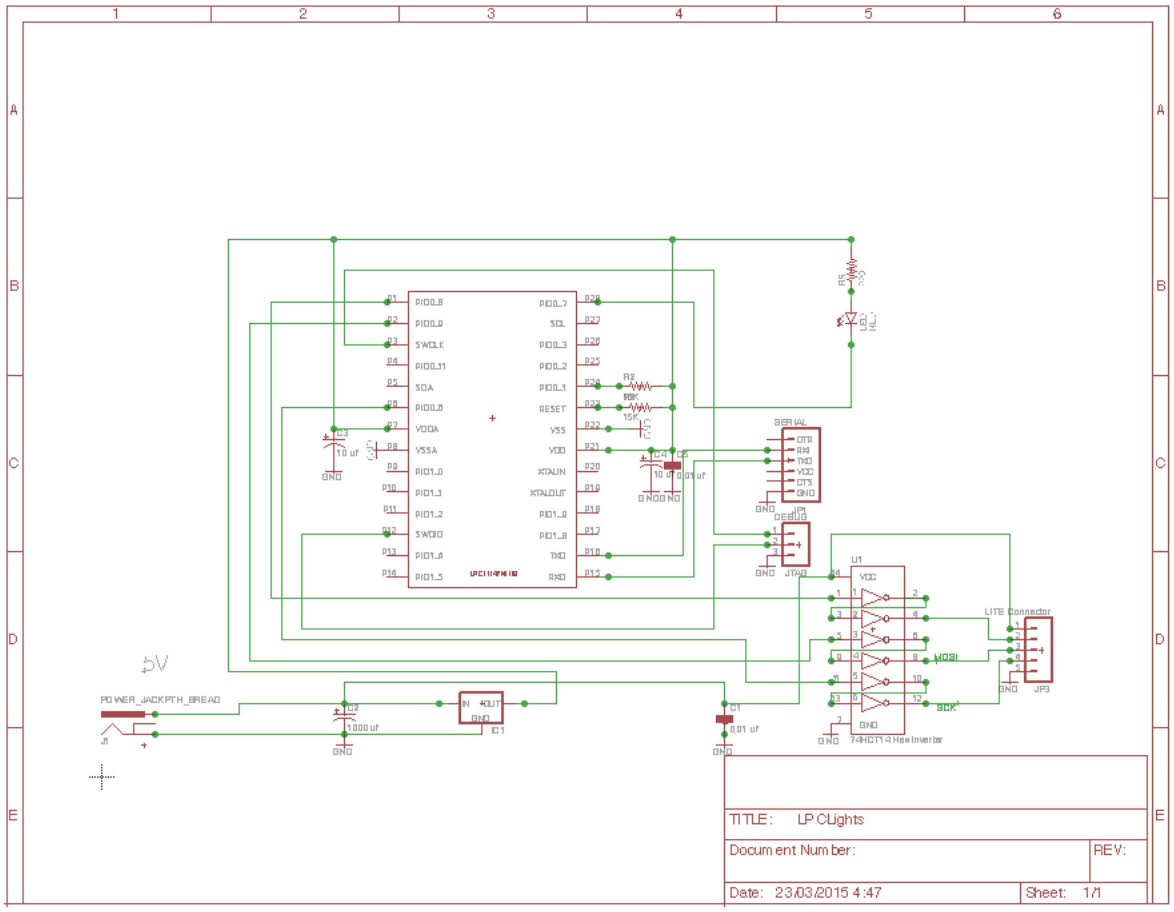

Figure 4: Schematic

This builds the following 1.95 “ X 1.95” board.

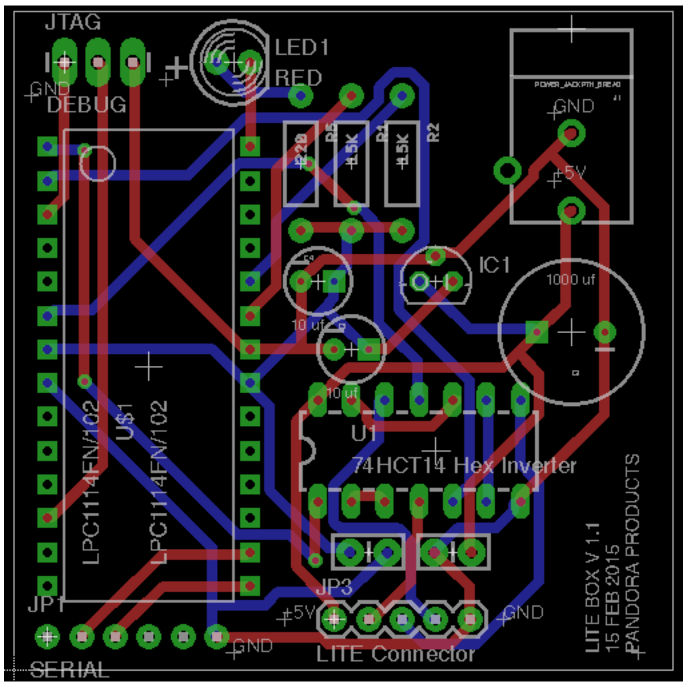

Figure 5: Board Design

Constructed board:

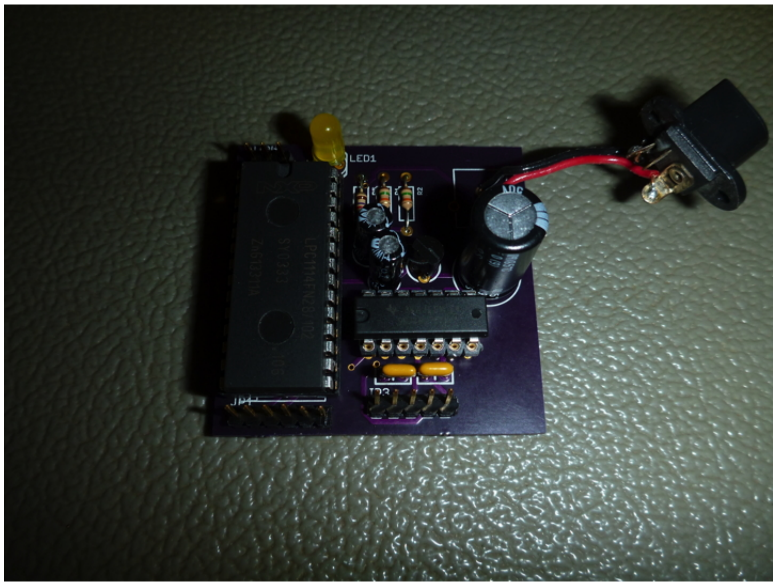

Figure 6: Initial Board Design

3 Interface

The interface is in two parts, there is a user interface and a computer interface. The user interface is a menu driven system allowing you to change parameters, stop and start the lights. The computer system give the same control but allows a computer to do this via checksummed messages.

3.1 User interface

The computer interface is a menu driven interface that looks like this:

```
LPC LIGHTS Program VER:[Feb  7 2015 11:09:20]
Lites Pgm
1 - Computer Control
      -- SETUP --
2 - Set Count
3 - Set Pattern
4 - Set Delay
5 - Set Shift
6 - Show Params
      --  SHOW --
7 - Start Show
8 - Stop Show
Choice:
```

3.2 Computer Interface

The computer interface uses checked/ACKed messages to allow a computer to have the same control.

3.2.1 Message Format

The message format is all printable ASCII characters and values are represented by HEX strings.

The items in the packet are:

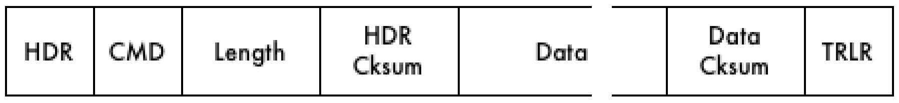

Figure 7: Control Packet

- HDR The character ':'
- CMD - Single character command specifier (see below)
- Length Hex representation of the packet, packet sizes are limited to < 60 characters.
- HDR Cksum MOD 256 sum of the header,CMD and length characters.
- Data Hex representation of the binary data (see below)
- Data Cksum MODE 256 sum of all the data
- TRLR The character ';'

3.2.2 Message Commands

The message commands are as follows:

3.2.2.1 ACK CMD = A This message has no data and is sent in response to any message correctly handled.

3.2.2.2 NAK CMD = N This message has no data and is sent in response to any message that either fails cksums or cannot be handled.

3.2.2.3 EXIT CMD = X This message has no data, when sent the program goes back to the user menu.

3.2.2.4 BEGIN CMD = B This message has no data and when sent starts the light show.

3.2.2.5 END CMD = E This message has no data and when sent stops the light show.

3.2.2.6 COUNT CMD = C The data in this message is the count of the number of lights in the string being run.

3.2.2.7 SHIFT TIME CMD = D The data in this message is the number of milliseconds between the shift of the pattern in the lights.

3.2.2.8 SHIFT CMD = S The data in this message is the number of lamps the pattern is shifted each time period. This can be 0.

3.2.2.9 LIGHTS CMD = L The data here allows changes to the pattern used to control the lights. The 4 bytes of binary data are:

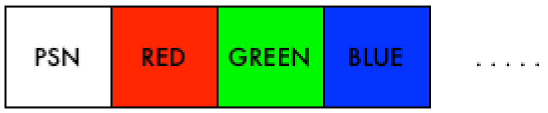

Figure 8: Light Word

This allows you to specify which lamp, and the color mix. You can have up to 10 of the settings in a single packet.

References

[1] Worldsemi. *WS2811*. Worldsemi, http://www.world-semi.com.