Music Controlled LED Disk
Team Lit Technologies

Written By:	Hunter Boock & Matthew Clemons
Class:	EE 403W Section 5
Published:	May 5, 2017

Executive Summary

Our final design project is a system that consists of a disk with 256 LEDs that displays different patterns based on frequency and amplitude characteristics of the inputted audio signal. The project has three different displays and the user is able to switch between these displays by clicking the desired image on the LCD. The board that is used for this project is the NXP OM13092 and the micro controller we were using is the LPCXpresso54608. The audio comes from a user device and then runs into the onboard audio input jack. The data is sampled and then a fast Fourier transform (FFT) is taken to get the frequency and amplitude characteristics of the audio signal. From there we are able to look through the frequency domain data to find the highest peak in desired frequency ranges. We split the data into three frequency ranges, low (0-200 Hz), mid (200-1500 Hz) and high (1500-5000 Hz). The peaks found in these three ranges were then either compared to the previous peak value or compared to the average power in that range to determine if certain LEDs were to be turned on or off. The method that was used depends on the display that is being played. Snippings of code will be shown through this document but the full code will be able to be seen on gitHub. You can get there by clicking the link attached at the end of this document

Background and Purpose

Lights that sync with music are used in many real world applications to capture the audience’s interest. Concerts, christmas lights, firework shows and parties are just a few examples of when audio controlled lighting can yield spectacular results. Our project was to design an audio controlled LED disk that displayed patterns and designs in response to an audio input. We wished to provide the user with multiple different design options as well as allow them to choose between them via the touchscreen LCD without interrupting their listening and viewing experience. The driving force behind this project is simply to provide enjoyment and entertainment for the user.

Implementation

Compiler and Design Tools

For this project we were writing C code in Keil uVision (microVision). The software development kit (SDK) and setup instructions can be found in the link posted at the end of the document.

Power

The LED disk (APA102C) requires a 5V 10A power supply. The VDD and ground wires from the disk are soldered to a power jack on a perfboard. Additionally, a 1000 µF bypass capacitor was included to minimize any noise. The 50 watt power supply was recommended because for each LED on the disk, it requires three leds (red, green and blue) to be able to create a large number of colors. Each one of these three internal LEDs uses up about 20mW of power so from that you get 60mW per LED when all colors are maxed out. With 256 LEDs the power is up around 10+ Watts when all of them are turned on and bright white.

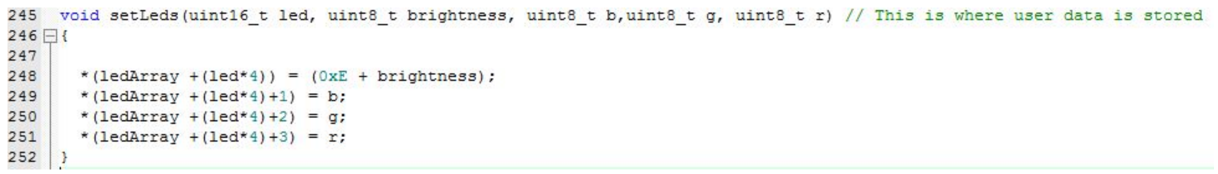
Software Architecture

Initializing Peripherals and Pins for SPI

The first part of our code consists of setup and initialization. The Flexcomm Interface is enabled and a 4 MHz clock source is selected. Specifically, Flexcomm 9 is chosen because it supports the SPI pins we use. As mentioned, SPI is chosen and set as the Flexcomm Interface function (Note: SPI 9 must also be chosen in correspondence to Flexcomm 9 Interface). SPI is configured to operate in master mode so that the clock and data are outputs because the primary goal is to simply write data to the disk. The FIFO is then configured for operation by enabling transfer data. Receive data for the FIFO is also ignored later with each write command to the disk. The Flexcomm Interface pin functions are configured through IOCON. The two pins that are configured are SCK and MOSI. SCK is set to J9 pin 9 (port 3_20) and MOSI is set to J9 pin 13 (port 3_21).

Creating Functions to Store and Transmit LED Characteristic Data

Code had to be made so that there is an easy way to change and update the brightness and color of a specific LED or range of LEDs. Our group accomplished this by creating a function called *setLeds.* The function allows the user to call it and then enter the LED number they want to write to, set the overall brightness and set the amount of the RGB values. At the beginning of the code an array, named *Array* was initialized with a length of 1100. The 1100 was more than we needed considering that we had 256 LEDs and each LED had 4 specific characteristics to be stored (brightness, blue, green, red). Therefor you would be fine with initializing this array to a length of 1024.Then a pointer was created to point to the data in the array, and called *ledArray.* Since the number of LEDs goes up to 256 we used a 16 bit integer to have some room for overflow and the color data for RGB were each 8 bits so we used an unsigned 8 bit integer for blue, green and red. Brightness is only a 5 bit word but still set as an unsigned 8 bit integer. The data sheet says that the order of the data words go blue,green,red and therefore have to be transmitted in that order. By storing them is this order you can use a simple for loop to transmit in the correct order. As you can see in figure 1 values are written to the array using pointer notation. By setting ledArray = Array in the main it makes *(ledArray) equal to Array[0], *(ledArray + 1) equal to Array[1] and so on. In the *setLeds* function we need to multiply the LED number by 4 and increase by one for each of the characteristics to store the data correctly. As you can see if you went to set LED 2 to have a full brightness and be all blue you would type “setLeds(2,31,255,0,0)”. The data in the parenthesis is passed to the function and then stored in slots [8,9,10,11] of Array[ ] in this example. So now Array[8] = 31, Array[9] = 255, Array[10] = 0, Array[11] = 0. All the other slots stay without a value until they are called and set. Notice that since there has to be a “111” at the beginning of the first word we add 0xE0 to the 5 bit brightness data since 0xE0 in hex equals 1110 in binary.

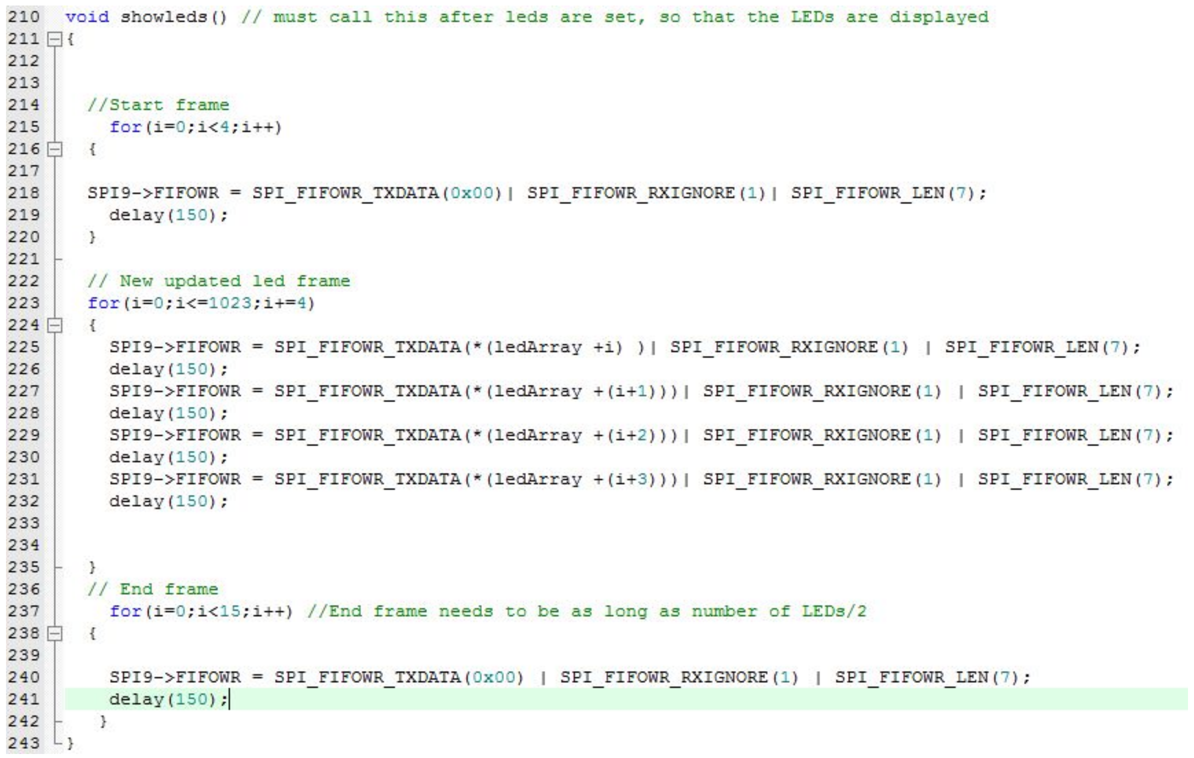


```
void setLeds(uint16_t led, uint8_t brightness, uint8_t b,uint8_t g, uint8_t r) // This is where user data is stored
{

  *(ledArray +(led*4)) = (0xE + brightness);
  *(ledArray +(led*4)+1) = b;
  *(ledArray +(led*4)+2) = g;
  *(ledArray +(led*4)+3) = r;
}
```

Figure 1 : Function that stores LED data

Once we had a function that allowed the user to set certain LEDs and store those settings, we needed a function that actually sent all the data out that the user set. The function we created is called *showleds*. The showleds function takes the data that is stored and iterates through all 1024 values of Array[ ]. Additional lines of code are added to make sure that there is a start and end frame in the transmitted data. The data sheet for the APA102 LED disk calls for 32 ‘0’s to be sent as the start frame and 32 ‘1’s to be sent in the end frame. Notice in line 218 of figure 2 0x00 is sent 4 times to make for a total of 32 0’s. The FIFO buffer was set to have a transmission length of 8 bits, that’s the reason that we send 8 0’s at a time. Then each of the array values are written to the buffer. This for loop uses i += 4 from 0 to 1023 and then the array value is just shown from *(ledArray+ (i+0))...*(ledArray+ (i+3)). This is one way of doing it, we could’ve kept it similar to the setLeds function and just multiplied by 4 and use i++ with a range of 0 to 255. In the SPI9 FIFO write lines of code we also set the RXIGNORE bit and make the FIFO buffer send words that are 8 bits in length. The RXIGNORE is set to tell the buffer that it doesn't need to look from data coming from the slave machine since we only need to send data to the LED disk. Setting this bit in the FIFO eliminates receive delays in the buffer. The last part of this function is the end frame. Even though the data sheet calls for 32 ‘1’s to be sent the correct end frame requires at least N/2 ‘1’s or ‘0’s (N = number of LEDs). A major problem occurred when we ran this code. It took our group a long time to figure out that we just needed a really small delay between the transmissions for the data to be sent and received correctly, that is the reason for the *delay(150)* call after each 8 bit transmission. The delay does not serve as an exact measurement of time, we just pass an integer into *delay()* which sets the number of times an empty for loop should execute. This wastes a very very small amount of time.



```
void showleds() // must call this after leds are set, so that the LEDs are displayed
{


  //Start frame
    for(i=0;i<4;i++)
  {

  SPI9->FIFOWR = SPI_FIFOWR_TXDATA(0x00)| SPI_FIFOWR_RXIGNORE(1)| SPI_FIFOWR_LEN(7);
    delay(150);
  }

  // New updated led frame
  for(i=0;i<=1023;i+=4)
  {
    SPI9->FIFOWR = SPI_FIFOWR_TXDATA(*(ledArray +i) )| SPI_FIFOWR_RXIGNORE(1) | SPI_FIFOWR_LEN(7);
    delay(150);
    SPI9->FIFOWR = SPI_FIFOWR_TXDATA(*(ledArray +(i+1)))| SPI_FIFOWR_RXIGNORE(1) | SPI_FIFOWR_LEN(7);
    delay(150);
    SPI9->FIFOWR = SPI_FIFOWR_TXDATA(*(ledArray +(i+2)))| SPI_FIFOWR_RXIGNORE(1) | SPI_FIFOWR_LEN(7);
    delay(150);
    SPI9->FIFOWR = SPI_FIFOWR_TXDATA(*(ledArray +(i+3)))| SPI_FIFOWR_RXIGNORE(1) | SPI_FIFOWR_LEN(7);
    delay(150);


  }
  // End frame
    for(i=0;i<15;i++) //End frame needs to be as long as number of LEDs/2
  {

    SPI9->FIFOWR = SPI_FIFOWR_TXDATA(0x00) | SPI_FIFOWR_RXIGNORE(1) | SPI_FIFOWR_LEN(7);
    delay(150);
   }
}
```

Figure 2 : Function that transmits LED data to disk

Fast Fourier Transform (FFT) Data Handling

To be able to look at the audio signals frequency and amplitude characteristics we needed to take an FFT of the signal. This is first done by initializing a buffer for the number of data points you plan on collecting from the transform. With this project we didn’t need to be exact or close to exact on different frequency ranges. Since this wasn’t important to the design we went with a 256 point FFT so it would execute faster than a higher point FTT and that’s why in line 89 we set the buffer size to 256 and in line 1135 we use 256 inside the FFT function. In the audio.c file you can see that we our sampling the data at 32kHz. With that sampling rate each data bin of our FFT only has a frequency resolution of 32000/256/2 = 62.5Hz. For most audio processing this would probably be too low of a resolution but since we define our own ranges of what low, mid, and high frequencies are, it doesn’t really matter.

Setting Thresholds by Sorting Through FFT Data

All three of the LED disk patterns have certain amplitude thresholds that the signal has to go over for a specific number of LEDs to light up. The first thing we did was separate the audio data into three distinct frequency ranges. The low frequency or bass we considered to be from about 0-200Hz. With our frequency resolution this meant we needed to look at bins 0-2 in the FFT data. We chose the middle frequencies to be from 200-1500Hz (bins 3-25) and the high range to be 1500-5000Hz (bins 26-79). We first tested only setting thresholds to fixed values. We pulled the highest amplitude/power peak out of each of the three frequency ranges by doing a sort on the data. We created a few lines of code (lines 1263, 1266-1270) that compared the past ‘maxAmpL’ value with the new FFT value and then ‘maxAmpL’ was replaced if the value was larger. This same method was used for the other two frequency ranges. Our group noticed that even the songs in the same genre had very different amplitude characteristics. Some songs might have way larger bass hits while others have very soft bass hits, etc. We knew that we were going to have to find a way to have thresholds that change as the music changes. We created two different ways to deal with this problem. The first method was to store the amplitude/power of each of the separated ranges and then make a calculation of the average amplitude every 50 iterations of some counter value (‘sumIt’). The newest, largest peak value (maxAmp_) is then compared to the ‘powerAvg_’ variable in each frequency range to see how much smaller or larger the present value is too the last calculated power average. The code for this method can be seen in lines 1305-1315 in the main.c file. If the present peak exceeds the power average by a certain amount it tells specific LEDs to light up. The other method was to just compare the present peak value with the peak value right before it, if the value was greater than the last value by a fixed amount it would light up certain lights. For example, if the first low frequency peak (maxAmpL) was read and it showed a value of 10, it would then be stored in oldValL. Then a second new value of maxAmpL was read and it was 100. Since the new value is much larger it would tell specific lights to turn on. Both of these methods allow the code to set thresholds based on the values of different songs and these automated thresholds make for a better and more fluent display on the LED disk.

Touch Screen

The touch screen was the last part of the code that we added to our main.c file. This part of the code uses functions from the eGFX library files. We first tried creating the touchscreen using the Touch GFX designer program. We got the display how we originally wanted it but realized that we didn’t have enough time to learn the operating system it used, freeRtos. We scrapped that method and started making our own code instead of Touch GFX generating code for us based off of our display. In lines 1142 to 1187 is where we are clearing the screen and dumping the screen buffer as well as placing the images and text on the screen. Make sure the clear plane and buffer lines are in that order, we had trouble with the screen when it was turned off and then back on without that code being there. Before you can can the eGFX_blit function to place an image on the screen you need to first convert the .png file into data that the screen can handle. To do this, go into the eGFX folder in your project file then click into the Sprites folder. Drop the .png files you want into the Sprites folder. Once you have done that go back to the eGFX folder and click the GenSprites.bat file. This will convert all of your .png files into the correct file for the eGFX structure. We placed an image of the LED disk for each of the three designs with the name of the design above the images. Then inside the infinite while loop we added if statements to react to a touch on one of the images, which causes the design on the LED disk to change to whatever the user clicked on the screen.

Display Patterns on LED Disk

This project has three set display patterns. The display patterns were given the names pieChunks, stopLight and lightning based off of what they look like. The pieChunks display is one of the three displays and this is what the program defaults to when the board is powered up. The pieChunks display/function turns on the LEDs that form a cross on the disk whenever the bass or low frequency range has a peak amplitude that is greater than the previous peak amplitude plus 400. The four symmetric quadrants of the disk fluctuate with the mid frequency peak amplitudes by comparing the present amplitude with the last computed power average. The greater the present value is than the previous power average, the more LEDs will light up. This makes the disk look like it is fluctuating from the inner to the outer ring of LEDs the louder these frequencies get. The second design is the stopLight function. The inner three rings light up green when the low frequencies are larger than the previous value. The next three rings do the same thing but light up yellow for the mid frequencies and the outer three light up red based on the high frequency amplitudes. The third and last display is lightning. This display has a set background to resemble a stormy sky and therefore no LEDs are ever off. The larger the amplitude of the low frequencies gets compared to the power average the more LEDs light up red towards the center and then a group of LEDs that look like lightning are lit up on the disk for the highest bass hits. All of these functions use variable thresholds for turning LEDs on and off. This makes the displays go well with the music no matter how different one song is from another.

Results

Overall, our project performed as intended. The main goals of our project were to output designs on the LED disk in response to an audio input, have multiple designs, and allow the user to select from those designs through use of the touchscreen LCD. All of these were accomplished. The *showleds* function created to write to the disk worked particularly well in simplifying and speeding up the process of making new designs. The disk itself, while requiring a lot of power, also displayed bright, clear colors for each design. Some of our other goals did change slightly over the course of this project. We had originally planned to enable audio input through the board’s digital microphone but as we progressed, we decided that this goal was supplemental at best and so we excluded it from our scope. One goal that we fell short on was the FFT display on the LCD. We had hoped to be able to, in addition to the design selection, display the real-time FFT power spectrum on the LCD. This was not accomplished because of a setback when designing the touchscreen LCD. We ran out of time but given a few more days we would have accomplished this as well.

Conclusion

Completing this project was a satisfying experience. We hit a lot of road bumps but always made progress and eventually met our end goal. Going back though, there would be a few things we could do differently given a second go at it. Most of the changes we could have made would be a result of inexperience and knowledge gained with this kind of a project. For instance, we could set up our peripheral function using drivers instead of doing it all manually, but even with just the knowledge we gained about setting up peripherals we could save a good bit of time. We spent a lot of time too on a software for the touchscreen that turned out to be less useful than we originally thought, so we would skip that and save more time. Future improvements to our project would obviously include displaying the FFT power spectrum on the LCD as mentioned earlier. We would also like to develop more designs and more ambitiously, we could even pursue a Bluetooth option.

Links

[1] Setup Instruction: https://community.nxp.com/docs/DOC-333654

[2] Code Files: https://github.com/Clemons11/Music-Controlled-LED-Disk

[3] Youtube Video: https://www.youtube.com/watch?v=3DTKbYZYn9U