FFT Graphing on LCD on a TI C5502 DSP Core

Real Time Digital Signal Processing - University of Nebraska

Daniel Shchur
62096122

Abstract

A frame buffer was implemented to draw pixel data onto the included LCD screen quickly; allowing for a maximum of 32 frames a second. The FFT calculation was normalized to show a great variance of magnitudes based on statistical analysis allowing for easy visibility of amplitude variance between signals on the display. A total of 32 frequency bins are shown on the display allowing for a range of 0 to 6000 Hz of frequency representation on the screen. With the FFT calculation, the display is able to be updated a little under 19 times a second. A demonstration of the functionality is presented in the videos linked in Section 6.1.

Contents

List of Figures

1 Introduction

With the ability to calculate the FFT of the filtered audio signal, a way to present the FFT was needed. The eZdsp board has a 96×16 Organic Light Emitting Diode (OLED) LCD display which makes for a great way to present the calculated FFT.

In this project, the output of the FFT is processed and turned into a bar chart to display on the 96×16 pixel dot matrix display using DSP BIOS to delegate different functionality as threaded tasks as to allow for multitasking on the system. This project builds upon the previous projects that allow for user input and control of filtering and muting while also relaying system state to LEDs. To improve system responsiveness and to allow for full control of the display, the previous parts of the project are changed and improved. The final implementation can be seen in the videos linked in Section 6.1.

2 OLED LCD

The OLED LCD is a 96×16 dot matrix pixel display controlled by a 128×64 display chip. The chip communicates over I^2C protocol and has a lot of configuration and extensibility to make designing a display solution easier. The display's pages are rows of column segments, each an 8 bit value representing a pixel on the screen. While the order can be flipped, the default set up is where the most significant bit represent the lowest of the 8 pixels in a row, and the least significant represents the highest. The screen can also be set up to index into memory in an assortment of ways such as by page, horizontal and down, as well as column wise. The bounds of the frame can also be limited if memory outside the screen is not used for something like scrolling.

2.1 Implementation

The display was implemented in a frame buffer fashion with the screen painted horizontally left to right, then dropping down a row and repeating, similarly to modern displays. The frame was also limited to the size of the physical screen since there was no need for the extra memory.

When drawing pixels on the screen, the whole screen is drawn before the next frame is drawn. This screen tearing as the whole frame is buffered before drawing the next. This does slow down the refresh rate, but it's still fast enough for the purposes of this project. The frame buffer is managed by a task which waits for frame messages from a mailbox to draw. To ensure the I^2C line isn't being utilized by any other processes during the drawing process, the operation is locked using a mutex. The implementation of the task is presented in Section 6.4 and the frame buffer drawing is presented in Section 6.5.

To aid in memory efficiency since the lowest bit count of a variable on the C5502 processor is a 16bit word, frames are bit packed into two columns of pixels for every single variable passed to the frame buffer. The frame buffer unpacks the two column 16bit variables into single column 16bit variables as necessary for communication through the I^2C implementation. This incurs a computation cost, but is only a few hundred cycles at most and save significantly on memory essentially doubling the amount of frames that can be stored in queue waiting to be painted. This also reduces the memory footprint of any task building frames to be drawn since now arrays are half the size since they are only unpacked once during the drawing process.

2.2 Efficiency

The major bottleneck of the whole process is the amount of time it takes to communicate using I^{2}C. That being said, a queued up frame can be drawn in roughly 9,353,000 cycles, or at the rate of 9.353ms per frame if only frames are being drawn on the system. This gives a theoretical maximum refresh rate of about 32 Hz. This is still very efficient. Some care could be taken to try and speed up the process such as only updating parts of the screen that have been changed, but that would take a significantly more complex logic and would mostly likely introduce some form of screen tearing.

When considering full run time of the drawing process, the system takes about 16,000,000 cycles between frames which includes FFT processing, FIR audio processing, and frame buffer drawing. This grants an estimated 18.75 hz refresh rate which is nearly real time with small differences already hard to catch with the human eye. This meets efficiency expectations of the project.

3 FFT Bar Chart

The FFT bar chart is a representation of the calculated FFT. With the frame buffer process decoupled from the FFT process, the FFT process takes care of creating frames of the bar chart to be sent for drawing using a mailbox. The chart was first prototyped in MATLAB to easily and quickly determine what sensible values would need to used when scaling the FFT for the display limitations of the physical device.

3.1 Implementation

Some design considerations needed to be made with how the bar chart was created such that it looked good on the small display, represented a sensible and wide range of frequencies based on the FFT resolution, as well as clearly showed different levels of amplitude of many frequencies that were easily perceivable. The final implementation can be seen in the videos linked in Section 6.1.

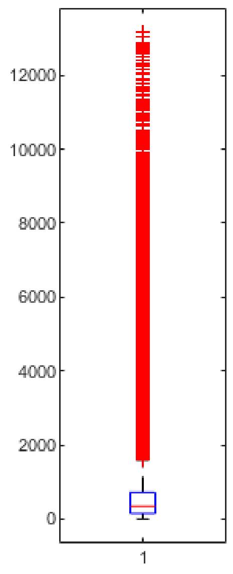
3.1.1 Design and Prototyping

When first implementing the chart in MATLAB, the song War by Edwin Starr was used because it represents a wide range of frequencies and a good simulation of input data. The generated FFT is 256 bins of frequencies in resolution; however, due to Nyquist, only the first 128 values of the FFT actually mattered as the other 128 values are the same but mirrored. This netted bins of roughly 187.5 Hz of difference starting with the lowest bin. The board being as small as it is would need more reduction of resolution however to be able to clearly show the different frequency bins. This is perfectly fine however since most audio is never exceeds 10 kHz of frequency. 32 bins of frequency were ultimately settled on since this would net three pixel columns per bin on the 96 column display. Two columns were used to draw the bars and one column was left blank in between bars to clearly separate the bars on the small screen.

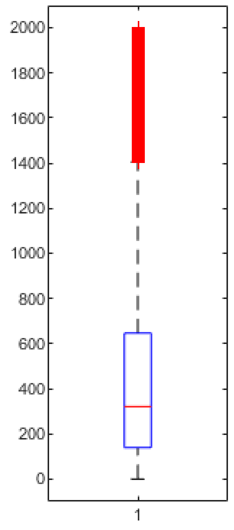
With 32 bins taken from the 128 bin FFT, this netted a linear frequency range of 0 Hz to about 6000 Hz. This is very close to ideal for music and voice data however since most sound is centered around the 80 to 6000 Hz range. Some sibilance is lost however since those sounds are usually above the 6000 Hz range, but that represents a very small portion of audible audio data.

The two filters implemented previously as part of the FIR task are also clearly represented in this range which makes it a very good choice to be able to show off the difference in the audio with and without filtering.

When taking the FFT of a signal, the power is a ratio of the frequencies present in the sample window. This presents an issue where a single clean frequency that can be represented by the FFT has an effective value of "1" while a mix of frequencies leads to all the values being less than "1". With audio data, this can vary greatly depending on the song and what instruments and voices are in the mix. There are many ways to combat this and normalize the data, but many require a good amount of processing time if done in such a way that accurately represents the differences in amplitudes of the different frequencies. Since the project at hand calls for an easy to see representation of the FFT however, a simple quantization with saturation was chosen, implemented with a set of conditional statements for speed. To determine what values should be quantized to the 16 pixels total pixels per vertical column, statistical analysis of the FFT over the entire song of War was performed in MATLAB. The entire song was processed in increments of 256 samples, saving the FFT to an array. This array was later plotted using MATLAB's box plot function. Figure 1(a) shows the plot with the original signal untouched. As can be seen, the bulk of FFTs values are below the range of 2000 with the average being around 450. This meant that the full range was not necessary to convey a fairly accurate representation of the signal. Cutting off all signals above 2000 resulted in the plot in Figure 1(b). The median was 321 with the lower quartile at 191 and the upper at 645. The cut off leads to a loss of 3.5% of the FFT values to saturation. This could be refined further but was left as is.



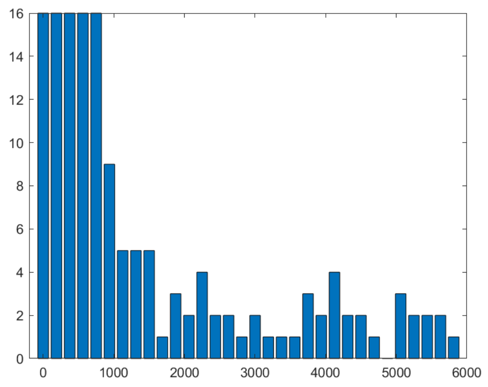
(a) Full values box plot



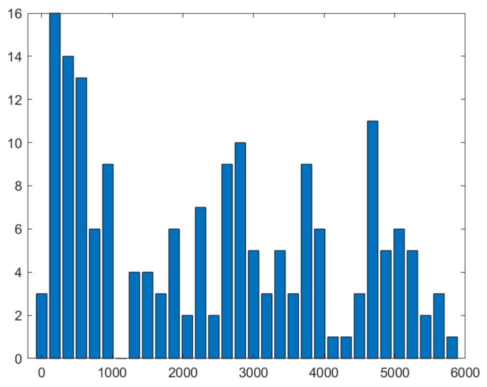
(b) Cut values box plot

Figure 1: Box plots of the total FFTs

To see if the design choices were visually appealing and still showed a great deal of accuracy, a few FFTs were saved and are shown in Figure 2. As can be seen, some signals are fully saturated which shows a loss in accuracy, but on the other hand, all of the frequencies are easily visible making it easy to clearly see which signals are present within a sampling window which is ideal for this design considering the limitations of the physical display.



(a) Normalized FFT showing saturation



(b) Normalized FFT showing variance

Figure 2: Normalized FFT bar charts

3.1.2 Real World Implementation

After all of the prototyping, the same logic was implemented on the board but with care to actually draw the frames as a bar chart instead of simply quantizing and normalizing data. The logic was broken up into a sub-function from the main FFT task to do normalization on an input FFT value turning a 16bit power value into a 16bit column of pixel data which represented the bar chart. Since this was implemented using conditionals, it was very cycle efficient. A snippet of this code is shown in Section 6.3. The FFT task was simply left with looping over 32 values of the FFT and bit packing the pixel data into a frame to be sent off to the display manager task. The implementation of these process is shown in Section 6.2 which shows how the column data was broken apart into a “high” and “low” portion which was doubled on the frame to make full use of the 3 columns allotted per frequency bin.

3.2 Efficiency

The drawing process is fairly efficient taking on average about 8000 cycles to complete a full frame of 96 pieces of pixel data. Some improvements could be made by not bit packing and instead representing frames as a full 192 columns, or by painting the screen vertically instead of horizontally which would mean no bit manipulation would need to be made since a full column could be kept as is; however, this was the best choice to save on memory usage as well as make it significantly easier to draw text on the screen if that functionality were to be used again. In comparison to the frame creation, the FFT takes about 5,050,000 cycles clearly showing that the actually calculation of the FFT takes much longer to complete. If any improvements were to be made, it would be in the algorithm used to generate the FFT.

4 System Improvements and Changes

A few system improvements and changes had to be made to accommodate the memory requirements of the FFT task and display changes.

4.1 Memory

Memory was a constant issue in the project since the coefficiencts of the FIR filters have to be stored in static memory and every frame buffer created would take up a good deal of memory. The FFT task also used a lot of memory because of the requirements of the actually FFT calculation. To make sure not stack overflows were happening the system was analyzed using Code Composer increasing stacks of any tasks which needed more memory space. To make up for the increase in memory requirements, any tasks that did not fully utilize their stacks were reduced in stack size such as display manager and idle tasks. This freed up the memory needed for the system. Another memory improvement which was mentioned in previous sections was the use of bit packing to cut down the memory usage of a frame by half. One of the limitations of the board used in this project is the fact that it has no 8bit variables, the lowest size is a 16bit variable. This leads to a lot of memory waste especially when some memory is simply used as booleans, a 1 bit value. If more memory were to be freed, improvements in the audio queues could be made since a whole 16bits of memory are used to represent whether a sample is 16bits leading to a doubling of memory usage per sample of 16 bit audio.

4.2 User Interface

The user interface task was simplified and improved due to the changes of the project. First, any functionality which manipulated the screen was removed since the FFT was being presented full time on the board. In the future, this could change where a message could be displayed for a short amount of time before switching back to the FFT chart. Another change is the addition of mutexes around the I^{2}C button reads to prevent collision of I^{2}C reads and writes when the display manager or user interface is using the lines. The final change made was adding an 80ms sleep at the start of the task to make sure other tasks of lower priorities were able to run in between polling for button changes. 80ms was chosen since it still felt responsive to the user and gave the most amount of computation time back to the other tasks.

4.3 Task Priorities

There were four tasks running at a time with different levels of priorities. The highest priority task was the audio processing task set to a priority of 8 since keeping the audio real time was the most important part of the project. The next highest task was the user interface task, reading buttons every 80ms. This is to make sure that responsiveness of the buttons were kept high since drawing to screen is a cycle expensive task and the FFT calculation was very expensive as well. After the user interface was the display manager task, drawing frames to the display. This was set higher than the FFT since care needed to be taken to make sure every frame from the FFT, or any other task, would be drawn in a timely manner. The FFT was the lowest priority above the idle task since calculating it and creating the frames to draw wasn't as important as the other tasks. The FFT could take its time and then push the frame onto the frame buffer once it was finished. This meant that the FFT could possibly be less than real time, but it was fine for the use of the project.

5 Summary

Fast and efficient screen drawing and bar chart representation of FFT was implemented and explored on the TI C5502 DSP core. The drawing was implemented in very efficient and visibly easy and pleasant to see way which gave it a high refresh rate, and a perceivable degree of accuracy even with the FFT running in the background. A real time representation of the processed audio data was able to be shown to the user with responsive changes to system state still operational through buttons and LEDs. This was all performed through the use of threaded tasks utilizing the DSP BIOS framework.

6 Appendix

6.1 Video Demonstations on YouTube

6.1.1 Quick Demonstration

https://youtu.be/Yxb0QnMMAUs

6.1.2 Full Presentation With Commentary

https://youtu.be/2YC2P2JKJ0s

6.2 FFT Task

```
void TSK_FFT()
{
    uint16_t i = 0, j = 0, shift = 0;
    uint16_t val, low, high;
    uint16_t fb[96];

    while(1)
    {
        MBX_pend(&mbx_processedAudio, FFT_U.In1, SYS_FOREVER);
        FFT_step();
        j = 0;
        shift=0;
        for(i=0; i<32; i++)
        {
            val = normalize(FFT_Y.Out1[i]);
            low = 0x00ff & val;
            high = 0x00ff & (val >> 8);

            if (shift == 1)
            {
                // Upper half
                fb[j] = (high << 8);
                fb[j+1] = high;
                // Lower half
                fb[j+48] = (low << 8);
                fb[j+49] = low;
                shift = 0;
                j+=2;
            }
            else
            {
                // Upper half
                fb[j] = (high << 8)| high;
                fb[j+1] = 0x0000;
                // Lower half
                fb[j+48] = (low << 8) | low;
                fb[j+49] = 0x0000;
                shift = 1;
                j++;
            }
        }
        if(!MBX_post(&mbx_frameBuffer, fb, 0))
            LOG_printf(&trace, "TSK_FFT: Failed to post to 'mbx_frameBuffer'");
    }
}
```

6.3 Normalization

```
uint16_t normalize(int16_t value)
{
    if(value >= 2000)
    {
        return 0xffff;
    }
    else if(value >= 1875)
    {
        return 0xfeff;
    }
    else if(value >= 1750)
    {
        return 0xfcff;
    }
    else if(value >= 1625)
    {
        return 0xf8ff;
    }
    else if(value >= 1500)
    {
        return 0xf0ff;
    }
...
    else if(value >= 625)
    {
        return 0x00f8;
    }
    else if(value >= 500)
    {
        return 0x00f0;
    }
    else if(value >= 375)
    {
        return 0x00e0;
    }
    else if(value >= 250)
    {
        return 0x00c0;
    }
    else if(value >= 125)
    {
        return 0x0080;
    }
    return 0x0000;
}
```

6.4 Display Manager

```
void TSK_DisplayManager(void)
{
    Uint16 fb[96];
    while(1)
    {
        MBX_pend(&mbx_frameBuffer, fb, SYS_FOREVER);
        LCK_pend(&i2c_lock, SYS_FOREVER);
        osd9616_frameBuffer(fb);
        LCK_post(&i2c_lock);
    }
}
```

6.5 Frame Buffer

```
int16 osd9616_frameBuffer(uint16_t* fb)
{
    uint16_t x, i = 0;
    uint16_t cmd[193];
    // Tell LCD this is pixel data
    cmd[0] = 0x40 & 0x00ff;
    for(x=1; x<193; x+=2)
    {
        // Unpack 16bit into 8 cmds
        cmd[x] = 0x00ff & fb[i];
        cmd[x+1] = 0x00ff & fb[i] >> 8;
        i++;
    }

    // Write bytes to OSD9616
    return EZDSP5502_I2C_write(OSD9616_I2C_ADDR, cmd, 193);
}
```