PERIPHERAL EQUIPMENT
2023 – 2024

PRACTICE 3

By Adrián Sánchez López

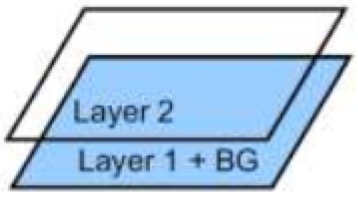
1. Program architecture and data structures

For the data acquisition in phase2 I already created an interrupt for the DMA stream that transferred the samples from memory to memory, and there I got the mean of the 8 samples. Therefore, I added 2 global variables, numSamples and numTransfers and in the interrupt of the DMA, numtransfers is incremented by 1 each time I get the 8 samples and calculate the mean. So when numTransfers is 10, I increment numSamples, set numTransfers to 0 and call the function displaySamples that updates the waveforms of both accelerations. In addition, when numSamples is 200 a flag that tells the functions to display the accelerations that we already received 200 samples is set. Finally, each time that the number of samples modulus 10 is equal to 0, the function that displays the bitmap in the x-y graph is called.

Regarding the display, the new average of the sample is passed as a parameter to the function displaySamples. In addition there is a global array that stores 201 Positions, which is a structure that stores the position corresponding to sampleAverage passed for both accelerations. DisplaySamples firstly checks if the flag I previously mentioned that tells the program if at least 200 samples have been received. If it has received less, then it calculates the position of the new sample for both graphics, stores it in the next free position of the array storing them and displays the pixel in the calculated position for both waveforms. On the other hand, if 200 samples have been received, then the new sample positions are calculated and stored in the array in the last position(position 201), and then a for loop goes through the whole array shifting the positions of the array to the left. While doing so it unsets the previous pixel position and sets them in their new position.

Regarding the x-y graph, when in the previous function the positions were calculated for both waveforms depending on the sample received, it also stored that sample in an array of 10 samples, which works like a LIFO. As mentioned before, when numSamples modulus 10 is equal to 0(100 ms have passed), the function that displays the bitmap is called, this functions calculates the average value of the 10 samples stored in that array that works like a LIFO and calculates the position of the value. Once calculated, it calls the function DrawBitmap to clear the previous bitmap and calls the DrawBitmap again to paint the new Bitmap. Finally, the position painted is stored in a variable that stores the previous bitmap position, so the next time this function is called, the bitmap painted now can be cleared.

2. Video subsystem configuration



As seen in the following diagram, 2 layers have been used, the first being the background that has rectangles that enclose both waveforms and the x-y graph painted in black and the rest painted in white. The second being a transparent layer where the waveforms and the bitmaps are painted and clear and updated, so no need for repainting the whole layer is needed, only the painted pixels to be updated are cleared and the painted in their new position. The first layer which is the background uses the address 0xD0000000 and the second one uses the address 0xD0050000. As said just now, the second layer is transparent except for the pixels needed for the waveforms and bitmap, that way the background doesnt need to be repainted at all and the few painted pixels from layer2 are the ones that are individually cleared and repainted. To do so, the alpha value has been used at the start to set all the pixels of the layer as transparent and to set as transparent the pixels painted that need to be cleared.

3. Implementation

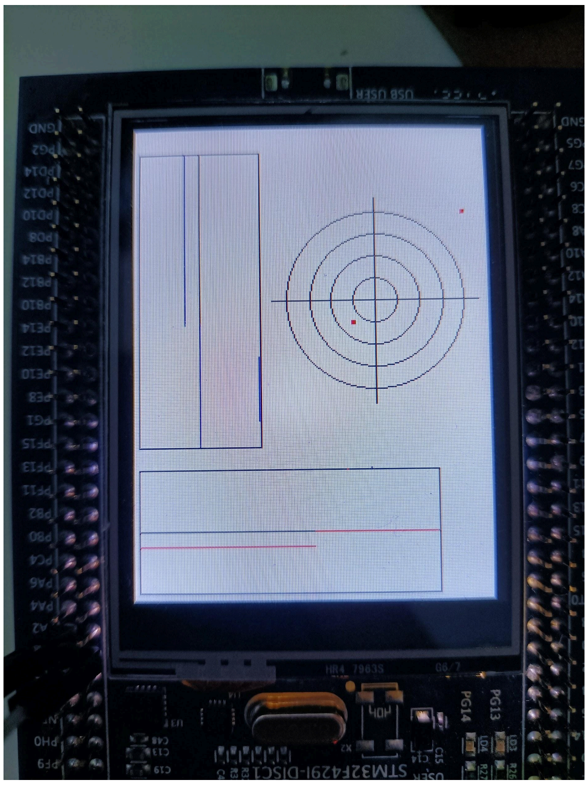
The requested pixel format is ARGB4444, which means that it uses 4 bits for Alpha value, 4 for the red value, 4 for the green value and 4 for the blue value. Therefore in total each pixel is made of 16 bits(2 bytes), so if I want to access the pixel in the row 5 and column 6, then I calculate the number of pixels(6 rows * pixelsPerRow + 6 columns(6 pixels)) and then the number of pixels is multiplied by the size of each pixel(2 bytes). So if the width is 240 pixels, then 6 * 240 + 6 = 1446 pixels * 2 bytes per pixel = 2892 bytes, this will be the offset that needs to be added to the address of the layer I want to draw on. Another interesting point is how to set up the 2 bytes conforming the pixel, as the functions received ARGB in a format of 8 bits and I want them in a format of 4 bits, I shift each of them 4 bits to the right so I only keep the 4 most significant bits. Then, with logical ORs I add the alpha value shifted to the left 12 bits, the red value shifted to the left 8 bits, the green value shifted to the left 4 bits and the green value without shifting, with that I conform the pixel in the ARGB4444 format.

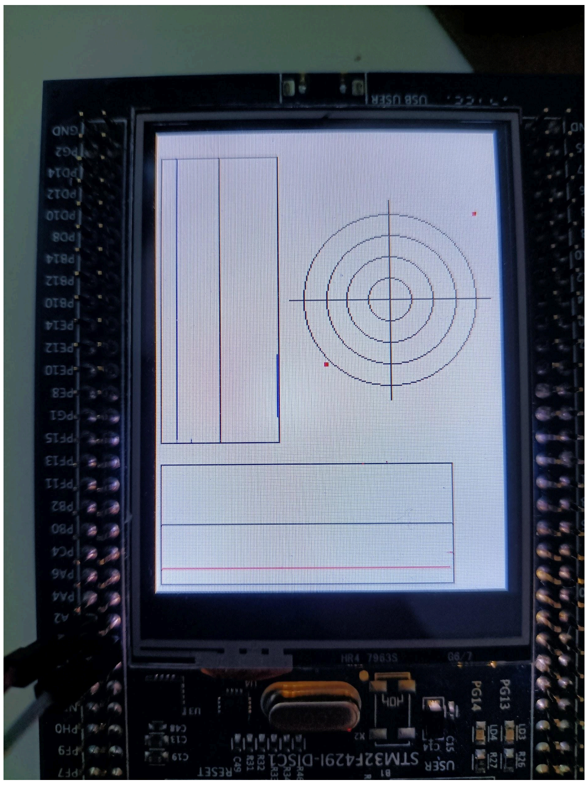
```
((alpha >> 4) << 12) | ((red >> 4) << 8) | ((green >> 4) << 4) | (blue >> 4);
```

I previously explained how to access a pixel to modify it, but that was to modify the whole pixel, if you only want to make the pixel transparent for instance. Then you only require the alpha value, as it is going to be transparent the RGB values will not be used for anything. In this case you will have to calculate the address of the desired pixel as said before, but as I only want to modify the alpha I will only set the 4 bits of the alpha value. If for instance I wanted to change only the green value, the as it is located after the 8 MSB, then I will calculate the address as explained, but I will add an extra 8 bits which are the 8 bits of the alpha + the red which are located before the green, and then set the 4 bits of the green.

Finally, for drawing the horizontal or vertical lines, the procedure is as follows: it will first check if the starting pixel and the ending pixel are inside the screen. If they aren't, then it returns a NO_OK. If they are, then it checks if the row/col_start is bigger than the row/col_end, if the start is bigger then it switches both. Afterwards, a for loop calls the function setPixel to set the pixels from the start point until the endpoint to the desired ARGB. If I wanted to be more efficient, then instead of calling the function set pixel that will calculate the address each time, then I could calculate the offset for the starting pixel and then multiply it by 2 for each of the following pixels until the end point. That way it would waste less time calculating the address.

4. LCD





5. Conclusions

Overall, this practice has helped me deepen my knowledge in data visualization devices and the architecture of their controllers. Developing a simple graphic library has made me understand even more how pixels work, ways to optimize different procedures such as accessing a pixel, drawing a line, drawing a circle… Not only that, but this has also been the first time I implemented a graphic library, though basic, it has laid the foundations towards making more complex graphic procedures in the future.