LCD Controller Design

Background

Figure 1: The Bargin that started all of this :)

A while back I picked one of these displays up from Electronics Goldmine, figuring I might be able to have a Launchpad drive one of these. Unfortunately I realized that the MSP's didn't have enough memory for a frame buffer of this size, so the display sat

gathering dust. Enter the $4.99 Stellaris Launchpad. Finally a device that could handle the required buffer, and with a USB interface to boot:) But how to get the pixels from the buffer to the display ??? Well, as it turns out, a MSP430G2452 makes a great companion to the Stellaris for this project.

System Overview

Here is a high level view of how this system is connected together :

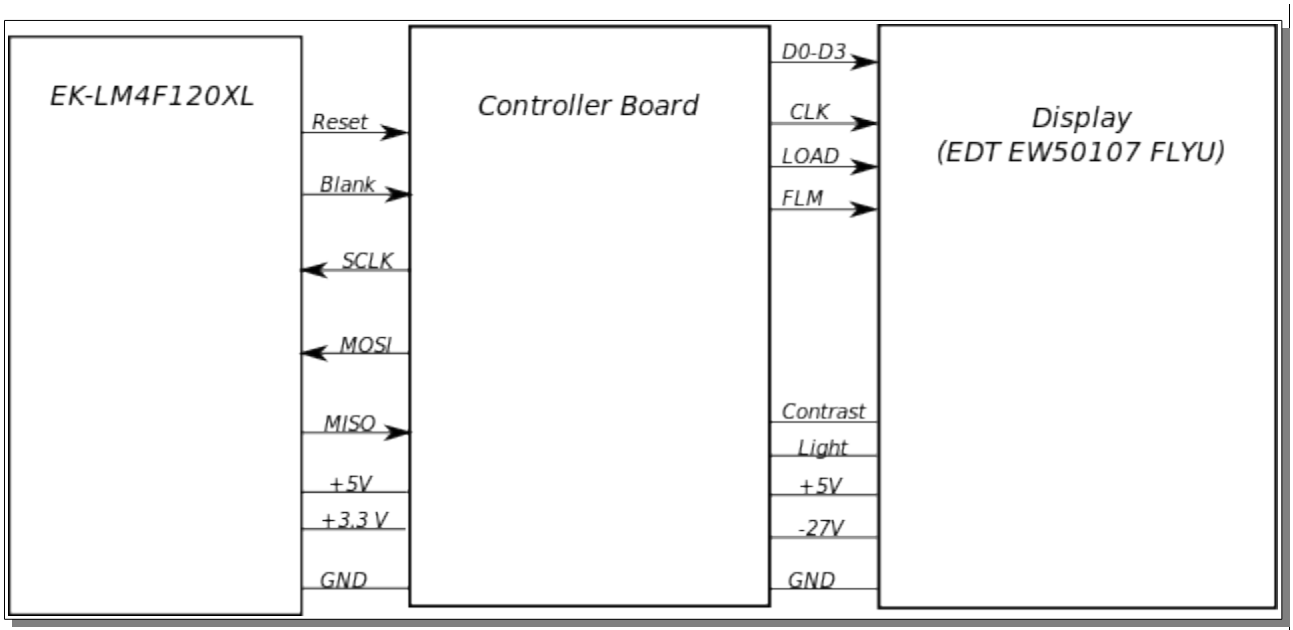

Figure 2:

Name	Type	Funciton
Reset	GPIO	Reset the MSP on the Controller Board
Blank	GPIO	Signal the MSP to stop SPI and turn off the display. Also used to start the SPI transfer after the Stellaris is ready
SCLK,MOSI,MISO	SPI	SPI bus pins. Controller has jumpers to select one of the four controllers on the Stellaris
+5V,+3.3V,GND	Power	Power Supply to controller and display

Table 1: Stellaris Pin Functions

Hardware Design

The controller board has three functions :

1. Host the MSP and the Glue logic to control the display
2. Genrate the **-27V** Bias voltage for the LCD
3. Act as a baseboard to hold the Stellaris Launchpad when attached to the back of the display unit.

Controller Section

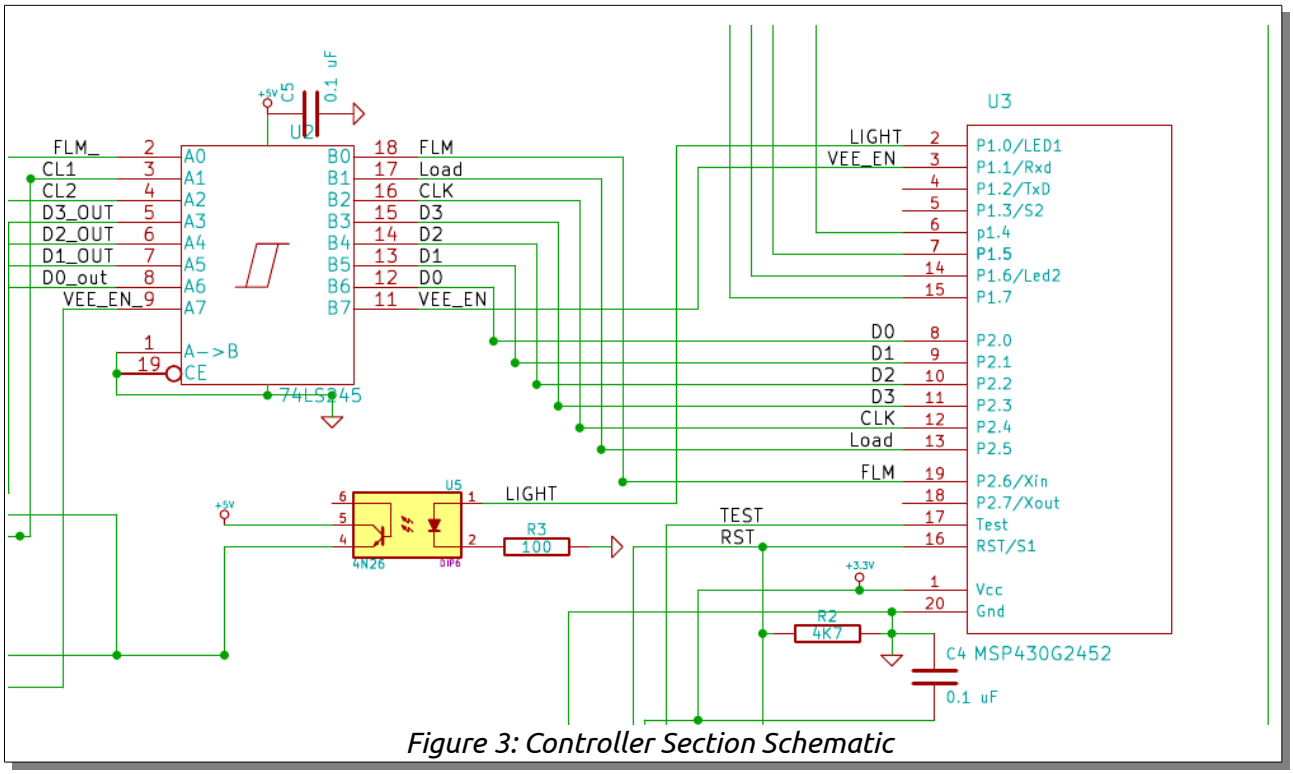

Figure 3: Controller Section Schematic

The Display contoller consists of the MSP430G2452 and a 74HC245 8 bit bus tranciever. The 74HC245 is used to drive the display logic at 5V and to provide a little protection to the MSP. The 4N26 phototransistor is used to switch the 5V supply to the LCD and the Backlight off when the display is blanked. I used this part mainly because I have a bunch in my partsbin and it seemed like a very safe way to interface the MSP to a 5V system.

The MSP430G2452 was used because the USI module can be configured to run as a 16bit SPI Master. This feature is important, since it allows us to do 4 pixel transfers for each SPI transfer. As it turns out, when running at a 8Mhz SPI clock, this allows the transfer to complete in about 2 pixel transfers or ½ of the Mainloop. This part also comes as the alternate chip with the newer revisions of the Launchpad:) Any of the other MSP430G2XX2 parts will work as well.

Bias Supply

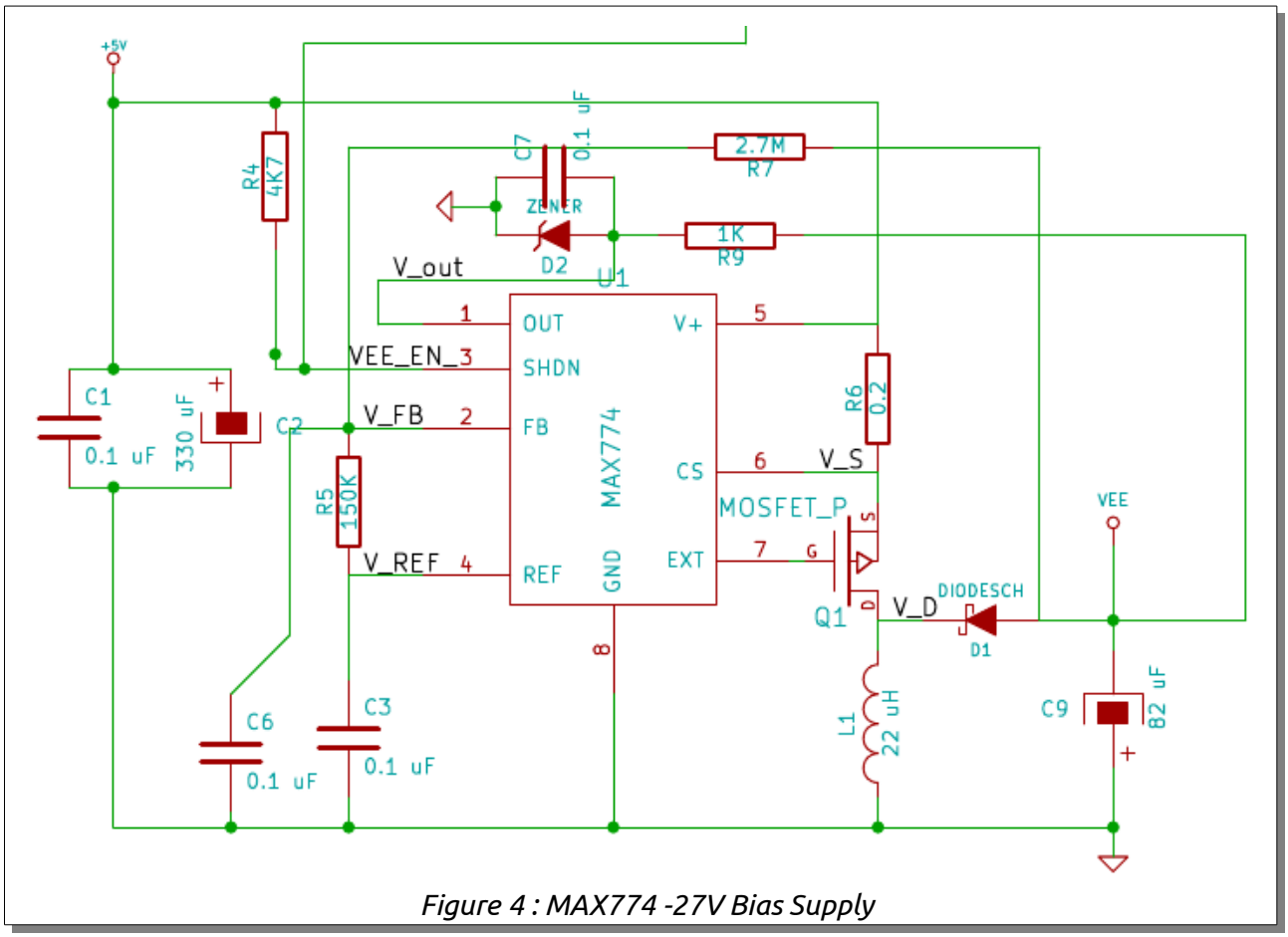

Figure 4 : MAX774 -27V Bias Supply

This turned out the the more difficult part of this project. After several unsuccessful attempts at charge pump based designs, I settled on a design based arount a MAX774 chip configured as an adjustable negative voltage supply. This circuit is designed around the MAX774 Datasheet. D2 is a 12V Zener Diode used to provide a negative bias for the Mosfet. I originally had OUT tied to GND, but the Fet sometimes would not fully switch.

Software Design

There are two pieces of software working together to get this project working :

1. The firmware for the MSP processor on the baseboard. This little guy is running like mad to fetch pixels from the stellaris, send them to the display and repeat the job.
2. The main application on the Stellaris. This consists of :
    - Setup routines to initialize UART0, the SPI port and uDMA
    - the interupt routines to handle the uDMA transfers
    - a command line interface for testing.

The stellaris code currently is a skeleton that should be usable by many other applications. The display data structure is fully compatible with grlib, so any application that wants to display monochrome graphics can now do so on a BIG 320x240 display. :)

MSP430

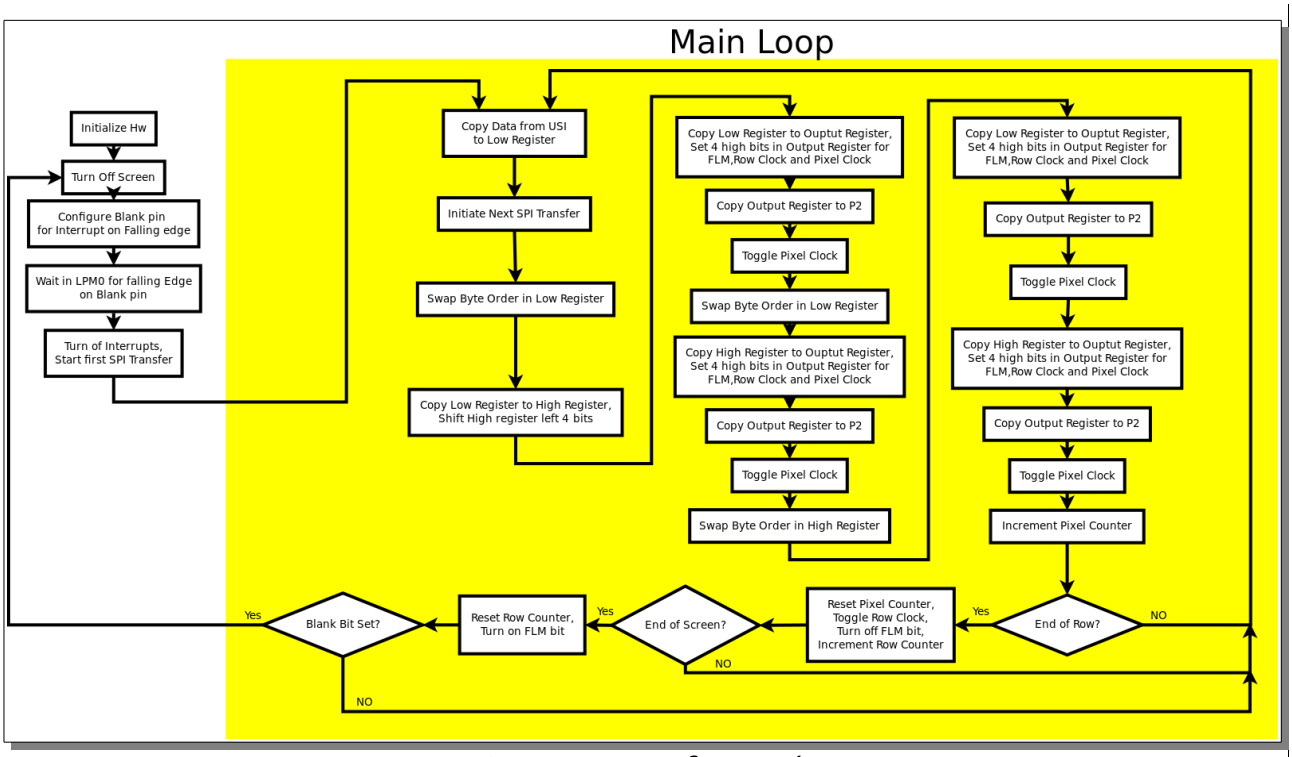

Figure 5: MSP Software Flow

The main goal of the MSP430 software is to allow the mainloop to run as fast as possible. This is accomplished in a couple of ways :

- Most of the operations inside the mainloop use Register variables to save clock cycles.
- The software was written in Assembly to make sure that there were no compiler issues wasting clock cycles
- I used the Naken Assembler since it gives cycle counts in the listing file, helping with finding inefficent sections in the code and making sure I could stay within my cycle budget.

The end result is a Mainloop that can send pixel data to the display at ~1Mhz, resulting in a ~51Hz Refresh rate for the display when running at a 16Mhz DCO.

Stellaris

Buffer Memory Layout

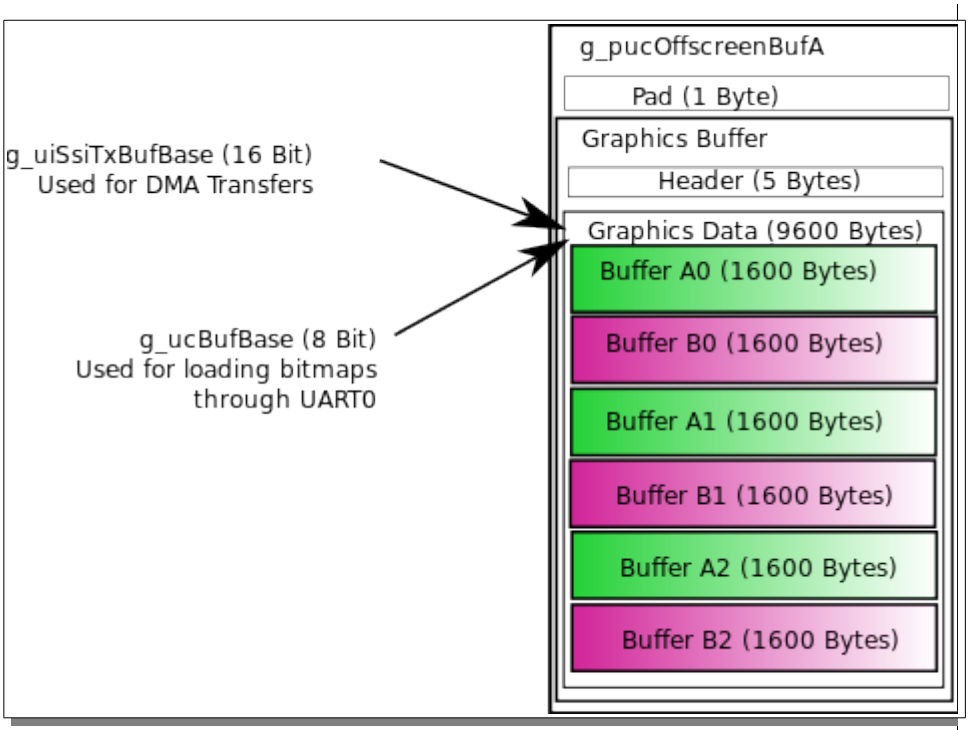

Figure 6: Graphics Buffer Layout

The display buffer layout is shown in Figure 6. In order for the uDMA transferrs to work correctly, the Graphics data needs to be aligned on an even address. The grlib 1bpp format adds a 5 byte headder in front of the actual graphics data. In order get the alignment correct, the g_pucOffscreenBufA array is defined using the *__attribute__ ((aligned(2)))* gcc directive, forcing the array to be on an even boundry. In addition, we leave a 1 byte pad before the grlib image structure.

Data Transfer Design

The Stellaris is set up as a 16bit SPI slave using the uDMA controller to keep the buffer filled and to discard the recieved data into a dummy buffer. The uDMA transfers are set up as a ping pong buffer. See Figure 6 for the details of the buffer layout. After initialization, the primary uDMA buffer is set to Buffer A0 and the secondary buffer is set to Buffer B0. When the A0 buffer has been transfered, the uDMA controller switches to the B0 buffer and genrates an interrupt. The interrupt handler detects that Buffer A0 is done and sets up Buffer A1 as the new primary buffer. When Buffer B0 is empty, the transfers now come from Buffer A1 and the secondary buffer is set to B1. Next Buffers A2/B2 are used. On the next interrupt the sequence starts back at Buffers A0/B0.

The uDMA transferrs are triggerd when the SSI FIFO buffer has 4 empty slots and is configured as a burst transfer of 4 16bit values. This keeps CPU overhead to a minimum.