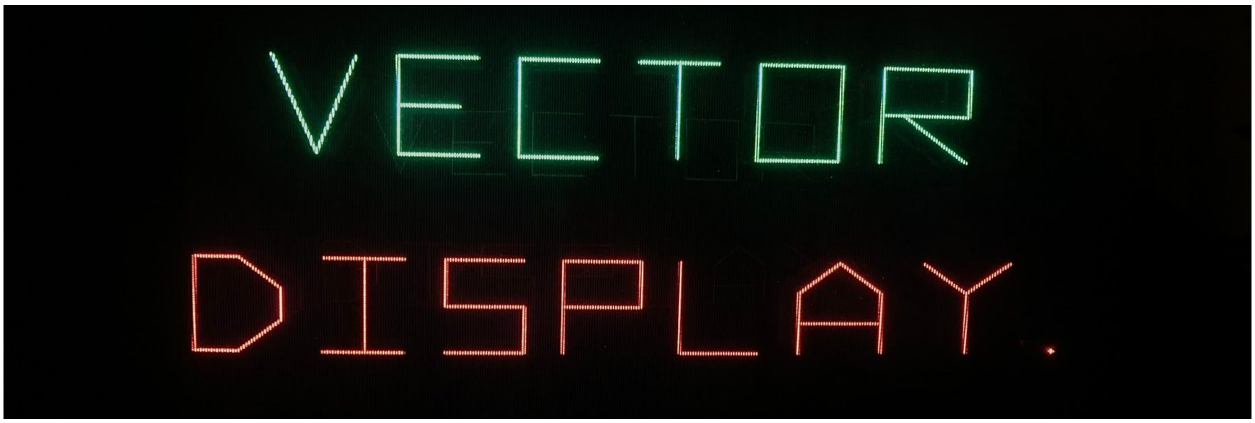


E155 Microprocessors Final Project Report

Ben Bracker and Varun Singh

Abstract

While advances in miniaturizing pixel technology have led to almost all modern displays being rasterized, vector graphics have the aesthetically pleasing property of being able to display objects without any pixelation, fuzziness, aliasing, or other issues that plague lower-resolution rasterized graphics because objects are drawn with continuous lines and curves. The only cost (and the reason raster won out eventually) is that vector graphics leave most of the screen area black. We created a vector display using an ARM microprocessor, dedicated circuitry, and an oscilloscope. We then created an interactive wireframe cube that could be controlled through keyboard input and displayed on the oscilloscope. An update rate of over 70 Hz was achieved. Several stretch goals were achieved, including using 3D instead of 2D graphics, being able to display on a Wells-Gardner 6100 XY vector monitor, and having multiple cubes that can be controlled individually.

Table of Contents

Background

Graphics have typically been displayed in one of two ways: rasterized or vectorized. Raster graphics involve discretizing an image into a 2D grid of pixels, each of which has a discrete number of possible colors. Vector graphics involve connecting points with smooth lines and curves. In the 1970's and early 1980's, vector graphics presented serious competition to raster graphics. One reason for this is that they can achieve a high resolution of possible screen positions using a much smaller amount of memory: whereas raster graphics requires storing color data for a resolution-dependent number of pixels, vector graphics requires storing position data for a resolution-independent number of points. Another advantage of vector graphics is that it greatly simplifies drawing shapes from a set of arbitrary points which is often required for 3D applications: this is why the first 3D raster game "I, Robot" followed the first 3D vector game "Speed Freak" by half a decade[3,4,5].

In more recent eras, when memory and processing power became inexpensive enough to satisfyingly drive raster displays, vector graphics fell out of favor since they do not work well for filling an entire screen with colors. Despite this shortcoming, we decide to drive a vector display because it still carries the intangible benefit of aesthetics and because it presents a nontrivial digital problem to solve.

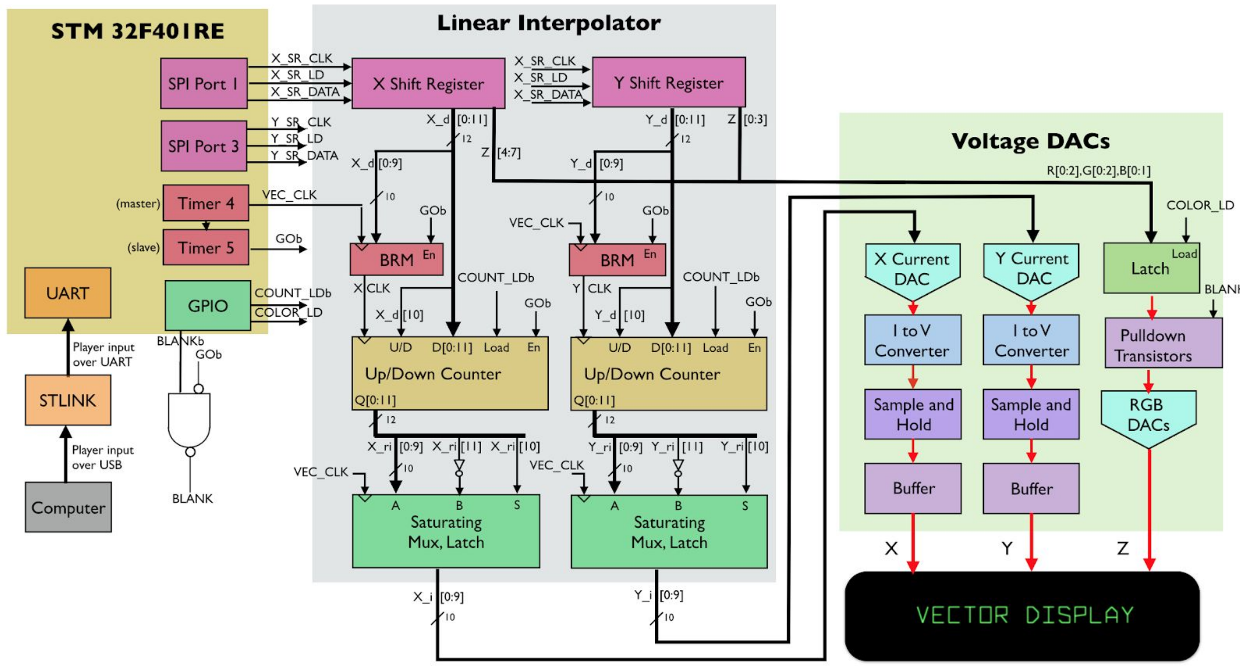
Objective

The goal of this project was to create a vector display using an ARM microprocessor, dedicated circuitry, and a screen. Specifically, a program was created for the STM32F401RE microprocessor to manipulate a wireframe cube through keyboard input. The necessary signals for displaying the cube were then fed to a hardware accelerator which generated analog voltages indicating the X and Y positions to draw lines on the screen. Two different screens were used: Scopy's oscilloscope feature in X-Y mode designed for the ADALM2000 data-acquisition model, and the Wells-Gardner vector monitor originally used by Atari for games such as Tempest, Space Duel, Gravitar, Black Widow, and Star Wars.

System Overview

The system consists of software which calculates the vectors to display and the hardware to actually display them. The information needed by the hardware to draw each vector includes X and Y values between -512 and 512 indicating how much to offset the vector from the current beam location in the X and Y directions on the 1024x1024 screen. A brightness value for the vector is also needed for the screens that can support it.

Block Diagram



Software Design

The software architecture has 4 main parts: initialization, keyboard input, vector generation, and vector transmission.

Initialization

Upon startup, the STM32F401RE microprocessor is first configured to run at its fastest clock speed, 84 MHz. Then, a number of initialization sequences are run to configure various timers, the general-purpose input/output (GPIO) pins, the memory-to-memory direct memory access (DMA) controller, a UART connection to the host laptop, and SPI connections to the hardware accelerator.

Two buffers are then created to store the vector information needed by the hardware. Two buffers are used instead of one so that the software can write to one while the other is being transmitted to the hardware via SPI. This ensures that only entire complete frames are transmitted, and that no artifacts are generated on screen from having vector data changed in the middle of the frame.

The first buffer is initialized with the data needed to display the edges of the wireframe cube and the text “HELLO GAMERS”. DMA is used in memory-to-memory mode to retrieve the vectors corresponding to each letter in the text and copy them in the appropriate order into the buffer.

Once this first buffer is populated, it is transmitted to the hardware (as will be described later) and the second buffer is activated so that it can be written to while the first one is being transmitted. Once initialization is complete, the system continues on to the main program loop, which uses user input to generate the next frame of vectors and transmit it.

Keyboard Input

The user interacts with the system using the keyboard on a laptop connected to the Nucleo. A Python script is run in the terminal to capture keyboard input and transmit valid character commands across the UART connection between the laptop and the Nucleo. An incoming character fires an interrupt on the microprocessor, and that character is stored for use in the main loop. Note that the system only keeps track of the most recent character sent, so all past characters are forgotten regardless of whether they were processed by the main loop. Within the interrupt, a flag is also set to indicate that a new command is available.

Within the main loop, the latest character command is processed to either rotate or scale the cube. The new command flag is also reset to prevent the same command from being executed multiple times. The vector data for the cube's new view are then generated.

Vector Generation

In the software, the cube is represented as the location of its vertices in homogeneous coordinates. Homogeneous coordinates were chosen because all affine transformations (rotation, scaling, translation, shear) can be represented by transformation matrices, which are then multiplied by the vectors indicating the vertex locations. Homogeneous coordinates also offer the advantage of being able to easily generate a 2D orthographic projection of a 3D environment. As a side note, homogeneous coordinates are popular in the computer graphics world for the same reasons that we are using them here - the allow all affine transformations to be linear transformations, and they make it easy to project from a higher dimensional space to a lower dimensional space.

After the new vertex locations of the cube are computed, the edges which define the cube are calculated. They are determined by taking the homogenous difference between the locations of the vertices. Then, these edges are orthographically projected into 2D space. Recall that the edges are represented by the change in X and the change in Y to get from one endpoint to the other.

Up to this point, all math has been done with floating point numbers, but the final calculation involves converting the edge information from floating point to integral values from -512 to 512. These values are placed into the active frame buffer. Any non-cube vectors, such as text on the screen, are static and pre-computed and stored separately. If those vectors on screen need to change, they are also copied into the frame buffer using memory-to-memory DMA. Once all vector data for a particular frame has been written to the active buffer, that buffer is locked and used for the next vector frame transmission, while the second buffer is made active and ready for re-population by the following frame's contents.

Vector Transmission

The vector data is transmitted over SPI from the microprocessor to the X and Y shift registers in the hardware accelerator, and then data is outputted from the shift registers after strobing their latches with GPIO pins. If the beam is to be loaded to an absolute position, the counters' load input is strobed so that they directly load the X,Y position. Otherwise a vector is to be drawn, so a timer in slave mode is used to activate the GOb signal for 1024 pulses of the master vector clock. When the timer reaches the correct count, it uses an interrupt to signal that it is finished and then the process is repeated for the next vector. When all vectors in the frame are drawn, the vectors in the other frame buffer are used to draw the next frame. In the rare event that the next frame has not been fully calculated yet, nothing is transmitted until the frame is ready for transmission.

Hardware Accelerator

Overview

The vector monitor linearly maps the X and Y analog input voltages to positions on the screen, and the DACs linearly map the X and Y digital inputs to analog voltages. At 10-bit resolution, corners of the screen are at the digital positions (0,0), (1023,0), (1023, 1023), (0,1023), and the center of the screen is at (512, 512).

At a high level, the STM microprocessor has two main ways to move the digital (and thus physical) position of the electron beam on the vector display. One way is to make the beam directly go to the absolute position given by (X_d[0:9], Y_d[0:9]). This is accomplished by sending the position data to the shift registers (X_d[10,11],Y_d[10,11] should be held low) and then pulsing the "Load" signal, which causes the counters to perform a parallel load. The other way to move the beam is to move it relative to its current position by

$$\left((-1)^{X_d[10]}X_d[0:9], (-1)^{Y_d[10]}Y_d[0:9]\right) \cdot \frac{\text{GO ticks}}{1024}$$

where "GO ticks" refers to the number of 6MHz clock ticks for which the GO signal is held high. This relative type of change in position makes use of the hardware linear interpolation so that the beam is moved gradually enough to draw a sharp-looking vector.

Other features include a Z signal for brightness control and USB input so that controller data can be acquired on an external computer.

Linear Interpolator

Shift Registers

The shift registers are used to reduce the number of pins needed to interface the linear interpolator with the STM microprocessor. Each shift register block is composed of 2 cascaded 75HC595 chips, so it is able to receive 16 bits from the STM at a time. For each shift register block, 12 of those bits are used for position data, and the rest are used for miscellaneous functions (like the intensity signal).

Binary Rate Multipliers (BRM)

Each binary rate multiplier block is composed of 2 cascaded 7497 chips. The function of the binary rate multiplier is to spread out an integer number of clock pulses (fed to the counters) over a fixed amount of time. Spreading the pulses out over time is important for vector generation because it allows the X and Y axes to change by different average rates which allows arbitrary slopes to be drawn. Otherwise if the X and Y axes received the same clock signal, only -45º, 0º, 45º, 90º angles could be achieved.

Up / Down Counters

Each counter block is a 12 bit up/down counter with parallel load composed of 3 cascaded 74LS191 chips. When used in parallel load mode, the BRMs are bypassed, allowing the microprocessor to directly set the (X,Y) position. When <axis>_d[10] is low, the final output <axis>_i[0:9] is set to <axis>_d[0:9]. When <axis>_d[10] is high, the muxes cause the final output <axis>_i[0:9] to be set to either all 0’s or all 1’s depending on <axis>_d[11].

When used in up/down counting mode, the counters convert the BRMs’ clock pulses to repeatedly adding or subtracting 1 from each axis’s current position. Adding versus subtracting (i.e. the direction of travel in each axis) is decided by <axis>_d[10]. Because each axis increments in steps of -1, 0, or 1 over time, the output (X_ri[0:9], Y_ri[0:9]) is the “raw interpolated” position between the starting position and destination position. The remaining two output bits for each axis <axis>_ri[10] and <axis>_ri[11] are used to detect under/overflows as described in the muxes section.

Saturating Muxes

Each saturating mux block is 3 composed of 74F399 chips. They are needed because a potential problem with feeding the “raw interpolated” <axis>_ri[0:9] values directly to the DAC is that under/overflow of these 10 bits would cause a wraparound condition, where the electron beam might be asked to move to the opposite end of the screen in very little time. To prevent this, the muxes implement saturation arithmetic. In the event of under / overflow of <axis>_ri[0:9], the counter will set <axis>_ri[10] high. If it is an underflow, <axis>_ri[11] will be set; if it is an overflow, <axis>_ri[11] will not be set. So, when under / overflow occurs, <axis>_ri[10] causes the muxes to select the “B” input which is not(<axis>_ri[11]), which is 0’s for underflows and 1’s for overflows. The output of the muxes is considered to be the final “interpolated” output: <axis>_i[0:9] .

Voltage DACs

Z DAC

For vector displays which include RGB intensity inputs, Z DACs can be used to control line brightness. Each DAC consists of resistor networks buffered with op-amps. In addition there is a latch to hold onto the color data because it is often the case that consecutive vectors will share the same colors. There is also a blank input so that the vectors can temporarily be made to have 0 brightness without reloading color data; this is helpful for repositioning the beam without drawing anything. *Note that this is only useful to the Wells Gardner monitor, not the ADALM2000 which lacks Z channels.*

Position DAC

Each position DAC is an AM6012PC. These were selected for their high speed settling time, high speed input frequency (achieved with a parallel interface), ≥10 bit resolution, and DIP packaging. Since they use a slightly modified R2R architecture, their raw output is a bipolar current, and they allow for some glitching when new data is presented. The rest of the analog stages convert this output to a clean voltage output.

I-V Converter

Each one converts a DAC current output to voltage output using a TL082 op-amp and some resistors.

Sample and Hold

Each sample and hold circuit eliminates glitching effects by sampling the voltage from the preceding I-V converter when it is stable and holding onto the sampled voltage while the input voltage might be transitioning with glitches. The sampling versus disregarding the input voltage is accomplished with a DG201B analog switch. The holding onto the voltage is accomplished with a high quality, low leakage 220pF capacitor.

Buffer

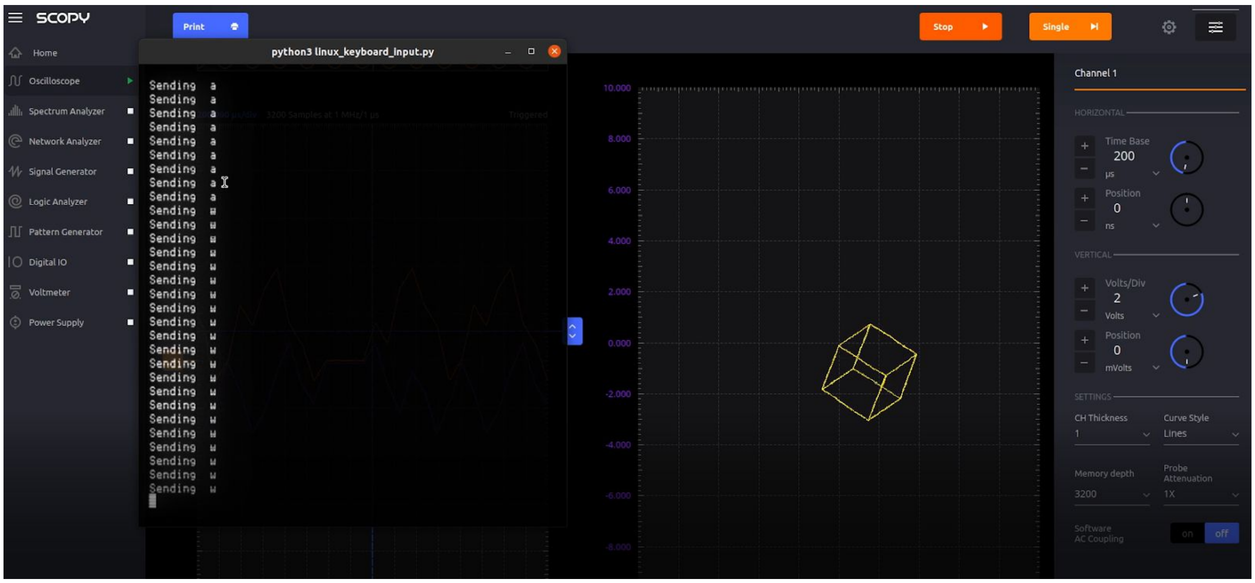
Each buffer is a simple TL082 op-amp buffer. Importantly it has low input leakage so that the hold capacitor does not quickly discharge in holding phases. It also can rescale the output voltage for different displays depending on how its gain is set.

Vector Display

The vector display is a monitor which plots X voltage against Y voltage, and if it's fancy, it will also control the current line brightness according to the Z input. For this project, an ADALM2000 oscilloscope will be used.

Results

We successfully created a vector display using the STM32F401RE microprocessor, dedicated linear interpolation and digital-to-analog circuitry, and the ability to display on ADALM2000 Scopy oscilloscope. We created an interactive wireframe cube that can be controlled through keyboard input, and the cube is displayed along with additional static text. When these images are displayed, frame refresh rate is over 70 Hz.



The interactive cube displayed in Scopy

In addition to the main objectives, we also achieved our stretch objectives of being able to successfully manipulate 3D objects while displaying them in 2D, being able to display on a Wells-Gardner vector monitor, and having multiple individually-controlled cubes.

Two interactive cubes displayed on a Wells-Gardner vector display

References

1. Jed Margolin, “The Secret Life of Vector Generators” (http://jedmargolin.com/vgens/vgmenu.htm)
   - Explains the design considerations and working principles of Atari vector generators. We relied heavily on this source for understanding digital vector generators.
2. Atari, “Asteroids Schematic Package” (https://www.mikesarcade.com/arcade/manuals.html)
   - Shows complete schematics of Asteroids’ digital vector generator, including annotations. We largely copy our linear interpolator and some of the analog output circuitry from this source.
3. The Killer List of Video Games, “Speed Freak” (https://www.arcade-museum.com/game_detail.php?game_id=9707)
   - Shows that Speed Freak was released in 1978 by Vectorbeam and has graphics that visually look 3D.
4. Internet Archive - “Arcade Game Manual: Speed Freak by Vectorbeam” (https://archive.org/details/ArcadeGameManualSpeedfreak/page/n62/mode/1up)
   - Shows that Speed Freak indeed has hardware capable of true 3D graphics (74LS181 bit slice ALU).
5. The Killer List of Video Game, “I, Robot” (https://www.arcade-museum.com/game_detail.php?game_id=8172)
   - Shows that “I, Robot” was released in 1983 by Atari and was the first game to use 3D filled polygon raster graphics.
6. HB Laser, “Laser Show Projects” (https://www.hb-laser.com/en/references/laser-show-projects.html)
   - Shows modern applications of vector graphics in laser shows.

Appendix A: Schematics

We borrowed heavily from the 1979 version 1 Asteroids schematics designed by Howard Delman under Atari. We copied the *X and Y Position Counters* schematic nearly signal for signal, and we referred to the *Video Outputs* schematic for the analog end. We chose to copy these parts because we did not feel we had time to perform R&D on developing a completely new vector generator.

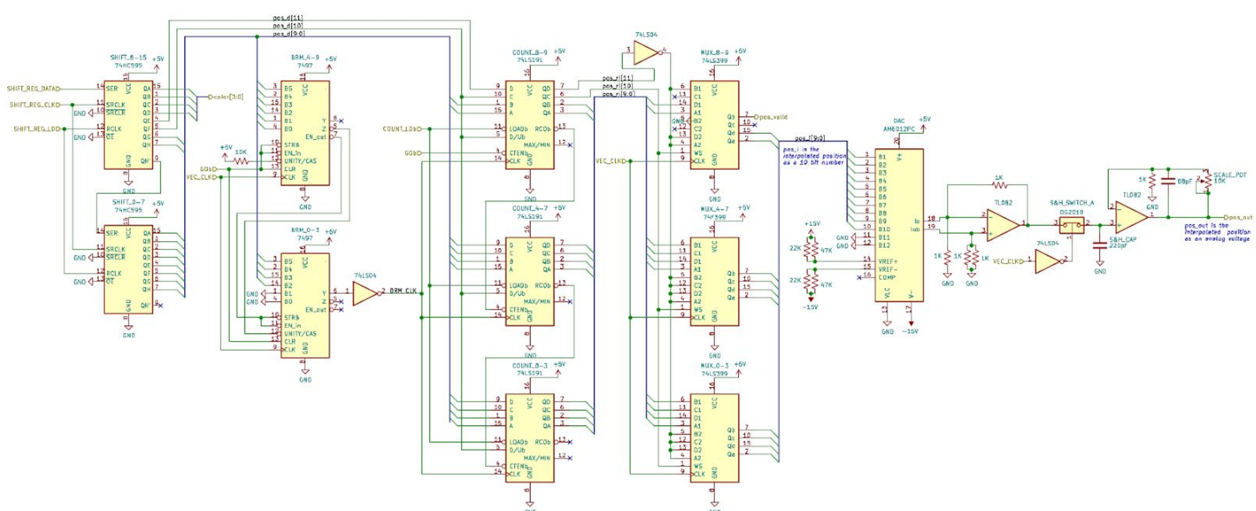
Key deviations from the Asteroids schematic include:

- Using 74HC595 shift registers to interface with the Nucleo64
- Using DG201B analog switch to save us from having to build and external level shifter
- Using AM6012 DAC in place of AD561J DAC, which is expensive to find

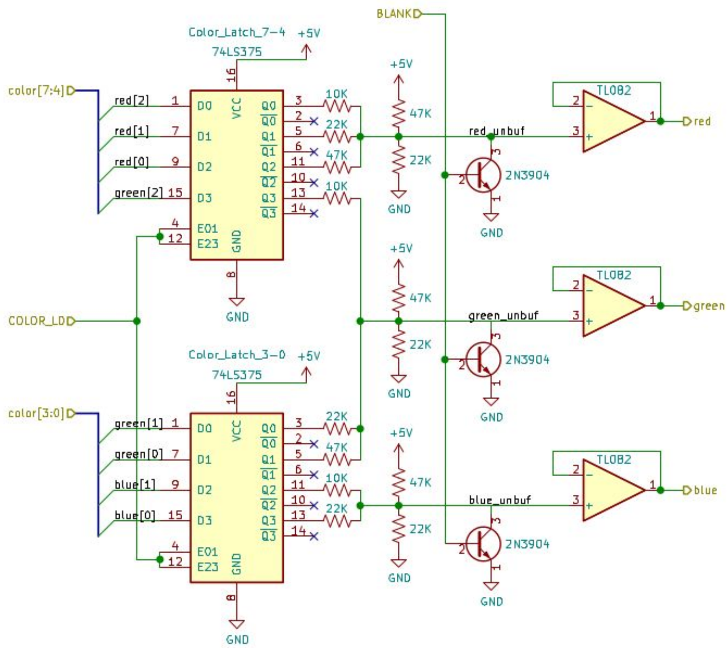
Since the AM6012 uses a slightly different architecture than the original DAC, we used the *Symmetrical Offset Operation* schematic in the DAC's datasheet in order to convert from the current outputs to a bipolar voltage.

Asteroids Schematics:
Asteroids_DP-143-1st-01A.pdf
Asteroids_DP-143-1st-01B.pdf
Asteroids_DP-143-1st-02A.pdf (contains *X and Y Position Counters*)
Asteroids_DP-143-1st-02B.pdf (contains *Video Outputs*)
AM6012 DAC Datasheet: (note that DAC312 is completely equivalent to AM6012)
https://www.analog.com/media/en/technical-documentation/data-sheets/DAC312.pdf

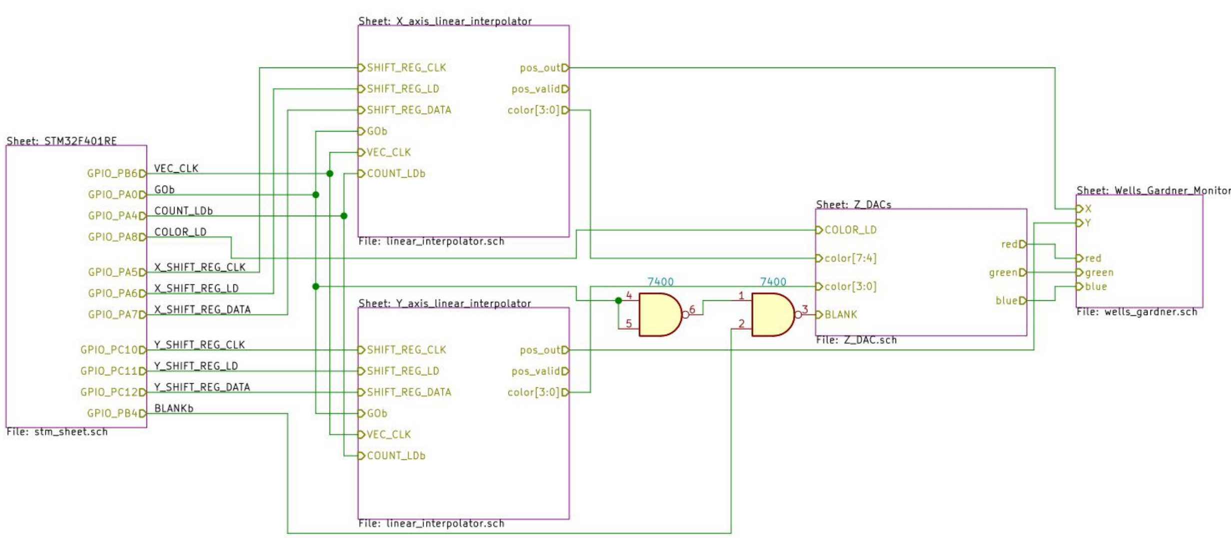
Here is a newly generated schematic that brings all the parts together and uses terminology consistent with this report. It represents the linear interpolator used for the X and Y axes.



Here is the schematic for the Z DACs. In accordance with the specifications of the Wells Gardner 6100 monitor, the color signals range from ~1-4V. The signal is generated by a resistor network and buffered with op-amps. Latches hold the colors so that the processor does not have to keep outputting color data to the shift registers if consecutive vectors have the same color. A blank input was also added to give the ability to set the color to black whenever a vector is not actively being drawn. The color is specified by an 8-bit RGB value. Three bits are all allocated to the red and green channels each, and two bits are used for the blue channel because the human eye is least sensitive to variations in blue light.



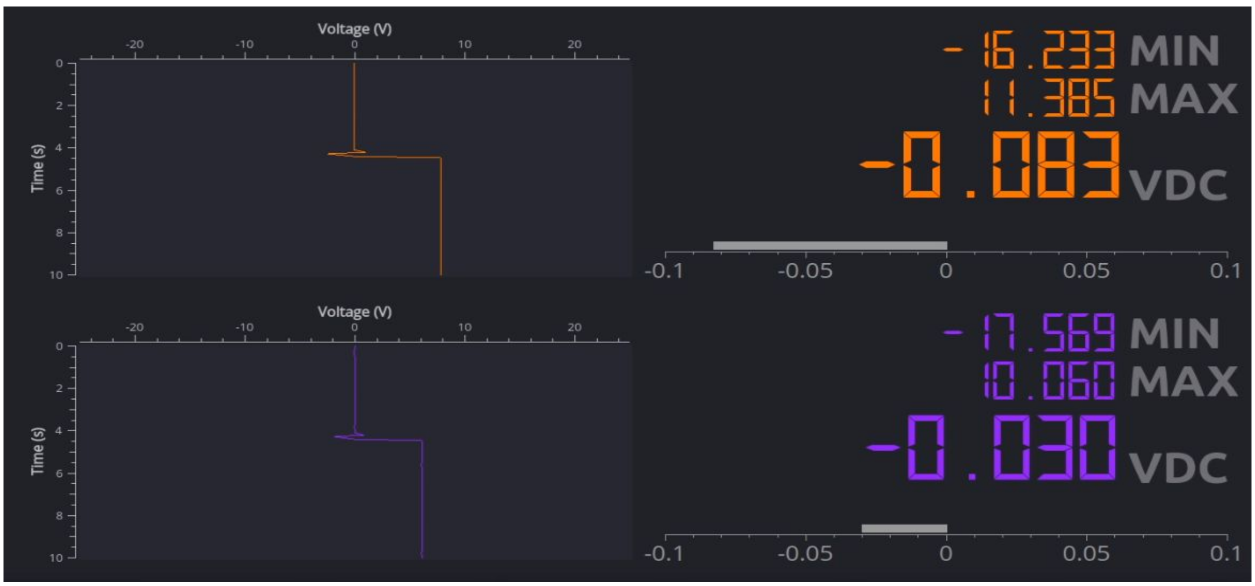
Lastly here is the overall schematic.



Appendix B: Circuit Testing

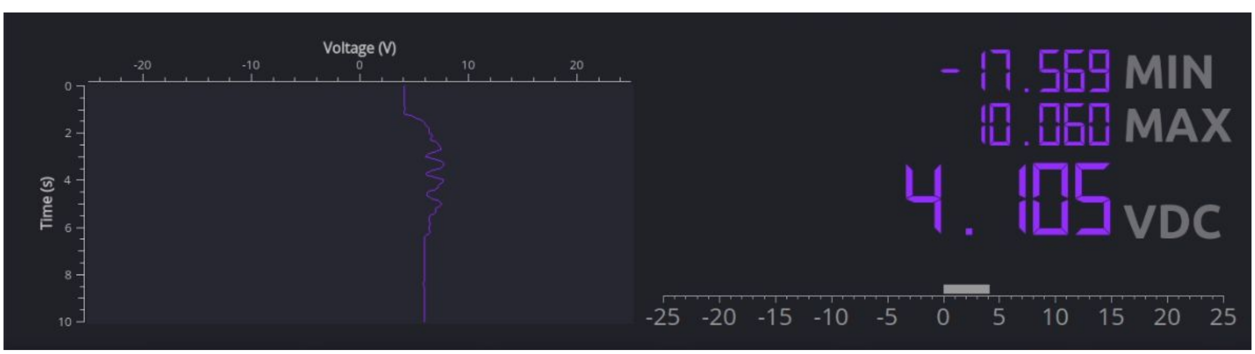
Centered Outputs

Values can directly be loaded into counters, and voltage outputs seem centered. When I press "Update BRM" in the UI, the loadCounter function is called (see appendix D), Subsequently a clear change in output voltage is observed on the oscilloscope. Furthermore that voltage behaves as expected. When 512 is directly loaded into the counters, the resulting voltages are nearly 0V. This is what we desire because the position (512,512) corresponds to the center of the screen, which is achieved on an X-Y oscilloscope by reading (0V,0V).



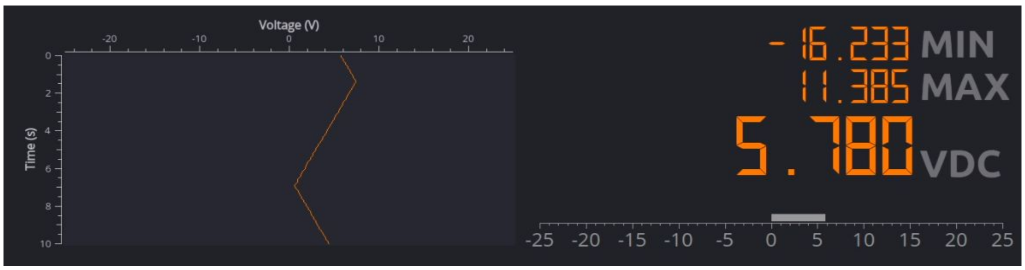
Full-Scale Outputs

Directly loading 1023 into the counters indeed maximizes output voltage. Furthermore, full scale voltage output can be controlled by the potentiometer. This waveform was generated by turning the Y-axis pot up and down over a few seconds.'



Linear Interpolation

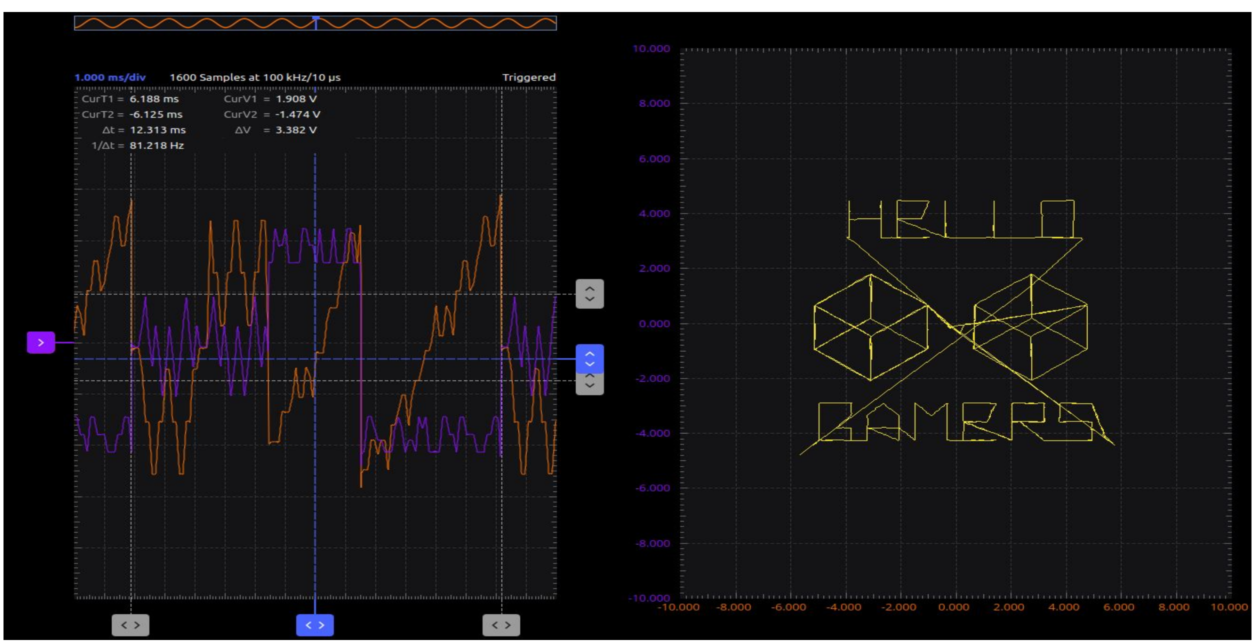
Linear interpolation is observed when the binary rate multipliers are enabled. When “Update BRM” is pressed with the “delta” checkbox checked in the UI, the runBRM function is called (see appendix D). Note that for this test, an input frequency of 256Hz is used so that the change could be humanly observable in real time. (When we actually go to draw images, we will use 6MHz). As expected, rate of change observably depends on the input value and the direction indeed depends on the 11th bit (“x10”) which controls the counters’ up/down input.



Drawing Images with Scopy and Frame Rate Measurement

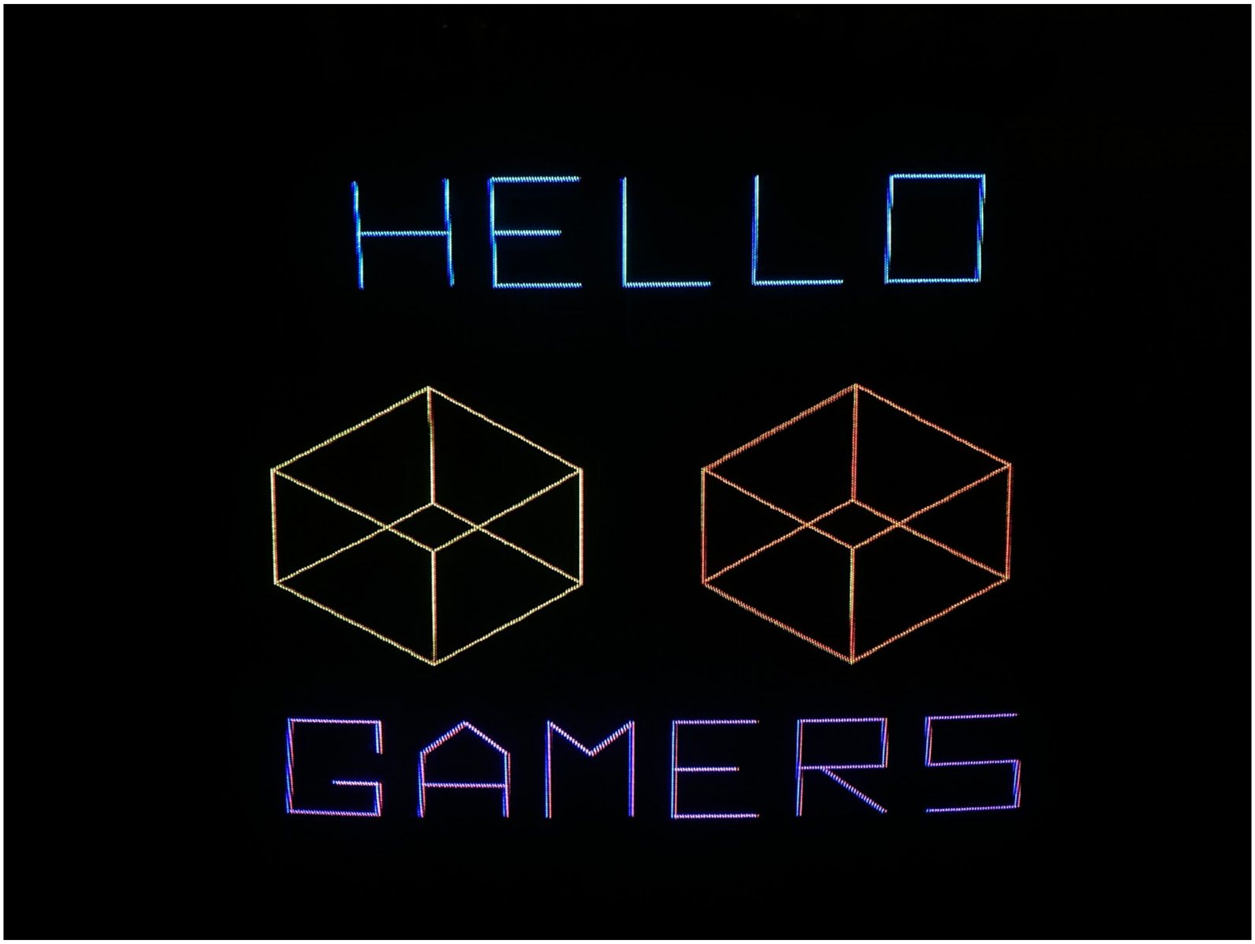
By putting Scopy in XY mode, we are able to see the “HELLO GAMERS” text and the two cubes. Because the Scopy software does not offer any Z inputs in XY mode, we also see lines representing beam movements that are used for repositioning and not intended to be seen as vectors.

Scopy also gives us the ability to check the frame rate. By aligning the cursors where the signal repeats, we are able to see that the screen refreshes at a rate of about 80Hz. Note that this frame rate does depend on how many vectors there are onscreen and how large they are.



Drawing Images with the Wells Gardner 6100 Monitor

Here we see the same image as was displayed on Scopy, except now the beam movements used for repositioning are not visible. We also see the color bits successfully control the RGB inputs to create a number of different colors. This is done by combining the channels in different ratios of intensities.



Appendix C: Bill of Materials

	Part	Link	Unit Cost	Quantity	Total Cost
Position Counting	74F399 quad 2 to 1 clocked MUX	https://www.jameco.com/shop/ProductDisplay?catalogId=10001&langId=-1&storeId=10001&productId=2298207	$0.59	6	$3.54
	SN7497N binary rate multiplier	https://www.jameco.com/shop/ProductDisplay?catalogId=10001&langId=-1&storeId=10001&productId=50796	$2.95	4	$11.80
	74LS191 up-down binary counter	https://www.jameco.com/z/74LS191-Major-Brands-IC-74LS191-Synchronous-4-Bit-Up-Down-Binary-Counter-5-Volt_47001.html?CID=MERCH	$1.09	6	$6.54
DAC and Sample-Hold	AM6012PC	https://www.arcadepartsandrepair.com/store/integrated-circuits/ttl-chips/am6012pc/	$7.50	2	$15.00
	DG202B	https://www.jameco.com/z/DG202BDJ-E3-Vishay-IC-DG202B-Quad-Analog-Switch-SPST-Normally-Open-16-pin-PDIP_239291.html	$2.09	1	$2.09
	220pf Sample and Hold Capacitor	https://www.jameco.com/z/CM05FD221K03-Cornell-Dubilier-Mica-Capacitor-220pF-10-500V-Radial-Through-Hole_2306831.html	$1.95	2	$3.90
	C110 output cap 68 pf	https://www.jameco.com/z/DC68-Capacitor-Ceramic-Disc-68pF-50V-5-_15552.html	$0.25	2	$0.50

	TL082 op-amp	https://www.jameco.com/z/TL082CP-Major-Brands-IC-TL082CP-Low-Power-JFET-Input-Operational-Amplifier-18V-100mW_33241.html	$0.79	2	$1.58
	10K pots	https://www.jameco.com/z/3362H-1-103VP-Trimmer-Potentiometer-10k-Ohm-1-2-Watt-10-Single-Turn-Top-Adjust_770371.html	$0.79	2	$1.58
Other Parts	74HC595 Shift Reg	https://www.jameco.com/z/74HC595-Major-Brands-IC-74HC595-8-Bit-Shift-Register-Output-Latches-and-Eight-3-State-Outputs_46105.html	$0.49	4	$1.96
	15V Regulator	https://www.jameco.com/z/7815T-Major-Brands-IC-7815T-15V-1A-Positive-Voltage-Regulator_51377.html	$0.39	1	$0.39
	-15V Regulator	https://www.jameco.com/z/7915T-Major-Brands-IC-7915T-15V-1-5A-Negative-Voltage-Regulator_51502.html	$0.59	1	$0.59
				Total	**$49.47**

Note that though they were already purchased, an ADALM2000 oscilloscope, Dell Laptop, and Atari Gravitar machine (specifically the power supply and Wells Gardner monitor) were also used for the project.

Appendix D: Clock Library

Header File: clock.h

```
#ifndef __CLOCK_H__
#define __CLOCK_H__

#include "stm32f4xx.h"

void configure84MHzClock();

#endif
```

Source Code: clock.c

```
#include "clock.h"

void configureFlash() {
    // Set to 2 waitstates
    FLASH->ACR &= ~(FLASH_ACR_LATENCY);
    FLASH->ACR |= FLASH_ACR_LATENCY_2WS;

    // Turn on the ART
    FLASH->ACR |= FLASH_ACR_PRFTEN;
}

void configure84MHzPLL() {
    /*
     Set clock to 84 MHz
     Output freq = (src_clk) * (N/M) / P
          84 MHz = (8 MHz) * (336/8) / 4
     M:8, N:336, P:4
      Use HSE as src_clk; on the Nucleo, it is connected to 8 MHz ST-Link
clock
    */

    RCC->CR &= ~(RCC_CR_PLLON); // Turn off PLL
     RCC->CR &= ~(RCC_CR_PLLI2SON); // Turn off the I2S PLL too (it shares
M and src_clk)
```

```
     while (RCC->CR & RCC_CR_PLLRDY); // Wait till PLL is unlocked (e.g.,
off)

    // Select HSE as src_clk
    RCC->PLLCFGR &= ~(RCC_PLLCFGR_PLLSRC);
    RCC->PLLCFGR |= RCC_PLLCFGR_PLLSRC_HSE;

    // Set M
    RCC->PLLCFGR &= ~(RCC_PLLCFGR_PLLM);
    RCC->PLLCFGR |= (8 << RCC_PLLCFGR_PLLM_Pos);

    // Set N
    RCC->PLLCFGR &= ~(RCC_PLLCFGR_PLLN);
    RCC->PLLCFGR |= (336 << RCC_PLLCFGR_PLLN_Pos);

    // Set P (yes, 0b01 is interpreted to mean a factor of 4)
    RCC->PLLCFGR &= ~(RCC_PLLCFGR_PLLP);
    RCC->PLLCFGR |= (0b01 << RCC_PLLCFGR_PLLP_Pos);

    RCC->CR |= RCC_CR_PLLON; // Turn on PLL
    while (!(RCC->CR & RCC_CR_PLLRDY)); // Wait till PLL is locked
}

void configure84MHzClock() {
     /* Sets system clock to 84 MHz from the PLL which is fed 8 MHz from
HSE */

    configureFlash(); // configure flash to support the higher clock speed

    // Turn on and bypass for HSE from ST-LINK
    RCC->CR |= RCC_CR_HSEBYP;
    RCC->CR |= RCC_CR_HSEON;
    while(!(RCC->CR & RCC_CR_HSERDY));

    // Configure and turn on PLL
     // Note that this will have the side effects of turning off the I2S
PLL
    // and possibly changing its input configuration.
    configure84MHzPLL();

```

```
    // Select PLL as clock source
    RCC->CFGR &= ~(RCC_CFGR_SW);
    RCC->CFGR |= RCC_CFGR_SW_PLL;
    while(!(RCC->CFGR & RCC_CFGR_SWS_PLL));

    // Set AHB (system clock) prescalar to 0 so we get full speed!
    RCC->CFGR &= ~(RCC_CFGR_HPRE);
    // Set APB2 (high-speed bus) prescaler to no division
    // (this will let our clocks receive the full SYSCLK freq)
    RCC->CFGR &= ~(RCC_CFGR_PPRE2);
     // Set APB1 (low-speed bus) to divide by 2 (because APB1 should not
exceed 42 MHz)
    RCC->CFGR &= ~(RCC_CFGR_PPRE1);
    RCC->CFGR |= RCC_CFGR_PPRE2_DIV2;
     // Note that clocks on APB1 will still get full 84 MHz if APB1 at 42
MHz
}
```

Appendix E: GPIO Library

Header File: gpio.h

```
#ifndef __GPIO_H__
#define __GPIO_H__

#include "stm32f4xx.h"

// Logic Levels
#define GPIO_LOW    0
#define GPIO_HIGH   1

// Arbitrary GPIO functions for pinMode()
#define GPIO_INPUT  0
#define GPIO_OUTPUT 1
#define GPIO_ALT    2
#define GPIO_ANALOG 3

// Pin definitions for every GPIO pin
#define GPIO_PA0    0
#define GPIO_PA1    1
#define GPIO_PA2    2
#define GPIO_PA3    3
#define GPIO_PA4    4
#define GPIO_PA5    5
#define GPIO_PA6    6
#define GPIO_PA7    7
#define GPIO_PA8    8
#define GPIO_PA9    9
#define GPIO_PA10   10
#define GPIO_PA11   11
#define GPIO_PA12   12
#define GPIO_PA13   13
#define GPIO_PA14   14
#define GPIO_PA15   15

#define GPIO_PB0    0
#define GPIO_PB1    1
```

```
#define GPIO_PB2     2
#define GPIO_PB3     3
#define GPIO_PB4     4
#define GPIO_PB5     5
#define GPIO_PB6     6
#define GPIO_PB7     7
#define GPIO_PB8     8
#define GPIO_PB9     9
#define GPIO_PB10    10
#define GPIO_PB11    11
#define GPIO_PB12    12
#define GPIO_PB13    13
#define GPIO_PB14    14
#define GPIO_PB15    15

#define GPIO_PC0     0
#define GPIO_PC1     1
#define GPIO_PC2     2
#define GPIO_PC3     3
#define GPIO_PC4     4
#define GPIO_PC5     5
#define GPIO_PC6     6
#define GPIO_PC7     7
#define GPIO_PC8     8
#define GPIO_PC9     9
#define GPIO_PC10    10
#define GPIO_PC11    11
#define GPIO_PC12    12
#define GPIO_PC13    13
#define GPIO_PC14    14
#define GPIO_PC15    15

void pinMode(GPIO_TypeDef * GPIOx, int pin, int function);
void alternateFunctionMode(GPIO_TypeDef * GPIOx, int pin, int alt_func);
int digitalRead(GPIO_TypeDef * GPIOx, int pin);
void digitalWrite(GPIO_TypeDef * GPIOx, int pin, int val);
void togglePin(GPIO_TypeDef * GPIOx, int pin);

#endif
```

Source Code: `gpio.c`

```c
#include "gpio.h"

void pinMode(GPIO_TypeDef * GPIOx, int pin, int function) {
    switch(function) {
        case GPIO_INPUT:
            GPIOx->MODER &= ~(0b11 << 2*pin);
            break;
        case GPIO_OUTPUT:
            GPIOx->MODER |= (0b1 << 2*pin);
            GPIOx->MODER &= ~(0b1 << (2*pin+1));
            break;
        case GPIO_ALT:
            GPIOx->MODER &= ~(0b1 << 2*pin);
            GPIOx->MODER |= (0b1 << (2*pin+1));
            break;
        case GPIO_ANALOG:
            GPIOx->MODER |= (0b11 << 2*pin);
            break;
    }
    GPIOx->OSPEEDR |= GPIO_OSPEEDER_OSPEEDR11; // speedy
}

void alternateFunctionMode(GPIO_TypeDef * GPIOx, int pin, int alt_func) {
    if (pin < 8) {
        GPIOx->AFR[0] &= ~(0b1111 << 4*pin);
        GPIOx->AFR[0] |= (alt_func << 4*pin);
    } else {
        GPIOx->AFR[1] &= ~(0b1111 << 4*(pin-8));
        GPIOx->AFR[1] |= (alt_func << 4*(pin-8));
    }
    pinMode(GPIOx, pin, GPIO_ALT);
}

int digitalRead(GPIO_TypeDef * GPIOx, int pin) {
    return ((GPIOx->IDR) >> pin) & 1;
}

void digitalWrite(GPIO_TypeDef * GPIOx, int pin, int val) {
```

```
    if (val) (GPIOx->BSRR) |= (1 << pin);         // set
    else     (GPIOx->BSRR) |= (1 << pin << 16);  // reset
}

void togglePin(GPIO_TypeDef * GPIOx, int pin) {
    digitalWrite(GPIOx, pin, !digitalRead(GPIOx, pin));
}
```

Appendix F: SPI Library

Header File: spi.h

```
#ifndef __SPI_H__
#define __SPI_H__

#include "stm32f4xx.h"

void configureSPI(SPI_TypeDef * SPIx);
uint16_t sendReceiveSPI(SPI_TypeDef * SPIx, uint16_t data);
void doubleSendSPI(SPI_TypeDef* SPIx, SPI_TypeDef* SPIy, uint16_t dataX,
uint16_t dataY);


#endif
```

Source Code: spi.c

```
#include "spi.h"

void configureSPI(SPI_TypeDef * SPIx) {
     SPIx->CR1 &= ~(SPI_CR1_SPE); // Disable SPI so we can change all the
settings
                                    // I assume user calls this on startup,
but if you want to change
                                   // midway, there may be more complicated
shutdown procedures
    // Baud Rate: divide by 4
    SPIx->CR1 &= ~(SPI_CR1_BR);
    SPIx->CR1 |= (0b001 << SPI_CR1_BR_Pos);

    SPIx->CR1 &= ~(SPI_CR1_CPOL); // Set polarity to idle low
    SPIx->CR1 &= ~(SPI_CR1_CPHA); // Set phase to leading edge

    SPIx->CR1 |= SPI_CR1_DFF; // 16-bit data
    SPIx->CR1 |= SPI_CR1_LSBFIRST; // LSB first
```

```
     SPIx->CR1  &=  ~(SPI_CR1_BIDIMODE);  //  For  full-duplex,  we  use  both
wires, and each is unidirectional
     SPIx->CR1  &=  ~(SPI_CR1_RXONLY);  //  For  full-duplex,  we  don't  want
receive only
    SPIx->CR1 |= SPI_CR1_MSTR; // Make STM32 act as the master
    SPIx->CR1 |= SPI_CR1_SSM; // To give software control of nSS...
     SPIx->CR1  |= SPI_CR1_SSI;  //  ...and  set  internal  nSS.  You  could  also
set SSOE instead.
     SPIx->CR1  &= ~(SPI_CR1_CRCEN); // Disable CRC calculation by default;
if you need it, write your own lib!
    SPIx->CR1 |= SPI_CR1_SPE; // Enable SPI
}

uint16_t sendReceiveSPI(SPI_TypeDef * SPIx, uint16_t data) {
    // Transmit
    while (!(SPIx->SR & SPI_SR_TXE)); // Wait until TX buffer is ready for
data to be written
    SPIx->DR = data; // load data into TX buffer to begin transfer

    // Receive
     while  (!(SPIx->SR  &  SPI_SR_RXNE));  //  Wait  until  RX  buffer  is  ready
for data to be read
    uint16_t message = SPIx->DR; // Read data from RX buffer
    return message;
}

void  doubleSendSPI(SPI_TypeDef*  SPIx,  SPI_TypeDef*  SPIy,  uint16_t  dataX,
uint16_t dataY) {
    /* Small optimization for sending to two SPI ports */
    SPIx->DR = dataX;
    SPIy->DR = dataY;
    /*
    // Transmit
    int xSent = 0; // has x data been sent yet
    int ySent = 0; // has y data been sent yet
    while ((xSent == 0) || (ySent == 0)) {
          if  ((xSent  ==  0)  &&  (SPIx->SR  &  SPI_SR_TXE))  {  //  wait  until  TX
buffer is ready for data to be written
                 SPIx->DR  =  dataX;  //  load  data  into  TX  buffer  to  begin
transfer
```

```
            xSent = 1; // flag x data as sent
        }
          if ((ySent == 0) && (SPIy->SR & SPI_SR_TXE)) { // wait until TX
buffer is ready for data to be written
                SPIy->DR = dataY; // load data into TX buffer to begin
transfer
            ySent = 1; // flag y data as sent
        }
    }*/
}
```

Appendix G: Timer Library

Header File: timers.h

```
#ifndef __TIMERS_H__
#define __TIMERS_H__

#include "stm32f4xx.h"

void configureTimer(TIM_TypeDef * TIMx);
void configureCaptureCompare(TIM_TypeDef * TIMx);

void configureDuration(TIM_TypeDef * TIMx, uint32_t prescale, uint8_t
slave_mode, uint8_t master_src);
void generateDuration(TIM_TypeDef * TIMx, uint32_t duration, uint32_t
compare_val);

void configurePWM(TIM_TypeDef * TIMx, uint8_t master_mode);
void generatePWMfreq(TIM_TypeDef * TIMx, uint32_t freq, uint32_t
duty_inv);

void start_micros(TIM_TypeDef * TIMx, uint32_t us);
void wait_micros(TIM_TypeDef * TIMx);
void delay_millis(TIM_TypeDef * TIMx, uint32_t ms);
void delay_micros(TIM_TypeDef * TIMx, uint32_t us);

#endif
```

Source Code: timers.c

```
#include "timers.h"

void configureTimer(TIM_TypeDef * TIMx) {
    // Set prescaler division factor
    TIMx->PSC = (uint32_t)(84-1); // Assuming 84 MHz
    // Generate an update event to update prescaler value
    TIMx->EGR |= TIM_EGR_UG;
    // Enable counter
    TIMx->CR1 |= TIM_CR1_CEN;
```

```
}

void configureCaptureCompare(TIM_TypeDef * TIMx) {
    /* Disable Counter */
    TIMx->CR1 &= ~(TIM_CR1_CEN);                          // disable counter
    /* Configure Capture/Compare Channel 1 for PWM Output */
    TIMx->CCER &= ~(TIM_CCER_CC1E);                        // disable capture/compare channel 1 so we can fiddle with its settings
    TIMx->CCMR1 &= ~(TIM_CCMR1_CC1S);                     // capture/compare channel to output mode
    TIMx->CCMR1 |= (0b111 << TIM_CCMR1_OC1M_Pos);   // set capture/compare channel 1 to PWM mode
    TIMx->CCER |= TIM_CCER_CC1P;                           // set polarity to active low
    TIMx->CCMR1 |= TIM_CCMR1_OC1PE;                       // set capture/compare channel 1 preload enable
    TIMx->CCER |= TIM_CCER_CC1E;                           // enable capture/compare channel 1
}

void configureDuration(TIM_TypeDef * TIMx, uint32_t prescale, uint8_t slave_mode, uint8_t master_src) {
    /* Configures a timer for outputting large precise delays.
        Given a TIMER_BASE_FREQ of 84 MHz, the resolution (each counter tick) is 1 / 84 MHz * prescale
        Likewise the longest countable duration is resolution * 2^(bits in ARR)
    */

    /* Disable Counter */
    TIMx->CR1 &= ~(TIM_CR1_CEN); // disable counter
    /* Select input clock */
    if (slave_mode) {
        // select which master's trigger to respond to (see table 54 of ref manual)
        TIMx->SMCR &= ~(TIM_SMCR_TS);
        TIMx->SMCR |= (master_src << TIM_SMCR_TS_Pos);
        // select external clock mode (aka output of the trigger selection mux)
        TIMx->SMCR |= (0b111 << TIM_SMCR_SMS_Pos);
```

```
    } else {
        // disable slave mode controller;
        // use instead internal clock CK_INT (aka "TIMx_CLK from RCC")
        TIMx->SMCR &= ~(TIM_SMCR_SMS);
    }

    /* Prescale */
    TIMx->PSC = prescale;

    /* Configure Counter as Upcounter */
      TIMx->CR1  &=  ~(TIM_CR1_CMS);  //  use  edge-aligned  mode  (i.e.  plain
vanilla up or down counting)
    TIMx->CR1 &= ~(TIM_CR1_DIR); // upcounter mode

    /* Allow Automatic Updating and Interrupting on Overflow Events */
    TIMx->CR1 |= TIM_CR1_ARPE;  // "auto-reload preload enabled"
                                    // transfers preload register to shadow
register every update event, meaning
                                      // we can change ARR and PSC without
having to manually request an update
    TIMx->ARR = 1;  // set max count to 1 as a default so that
                         // we generate update events quickly when we turn
timer on
     TIMx->CR1  |= TIM_CR1_OPM;    // stop counter at update events; we want
to time out just one pulse
    //TIMx->DIER |= TIM_DIER_UIE; // enable interrupts upon updating

    /* Implement Settings */
      TIMx->CR1  |=  TIM_CR1_URS;     //  let  only  counter  under/overflows
generate interrupts
                                    // so that when we manually generate an
update, it doesn't make an interrupt
       TIMx->EGR  |=  TIM_EGR_UG;      //  manually  generate  an  update  to
initialize all shadow registers
}

void  generateDuration(TIM_TypeDef  *  TIMx,  uint32_t  duration,  uint32_t
compare_val) {
    /* Generates an update after the specified duration.
```

```
         * The actual time duration is resolution (determined by
configureDuration) * duration
       * Note that duration should not exceed 2^31-1 (TIM2,5) or 2^15-1
(TIM3,4) */
        TIMx->CR1 |= TIM_CR1_UDIS;    // Disable update events because
apparently it would be bad
                                   // if the timer by chance tried to update
the shadow registers
                                    // while we're writing new values in the
preload registers.
    TIMx->ARR = duration;
    TIMx->CCR1 = compare_val;

    TIMx->CR1 &= ~(TIM_CR1_UDIS);   // enable update events;
                                    // (i.e. send update signal every time
counter under/over-flows)
                                             // In duration mode, this will
generate an interrupt.

    TIMx->EGR |= TIM_EGR_UG;  // manually generate an update to initialize
all shadow registers
    TIMx->CR1 |= TIM_CR1_CEN; // enable counter
}

void configurePWM(TIM_TypeDef * TIMx, uint8_t master_mode) {
    /* Configures a timer for outputting musical tones (or more generally,
PWM stuff)*/

    /* Disable Counter */
    TIMx->CR1 &= ~(TIM_CR1_CEN); // disable counter

    /* Select internal clock at maximum frequency */
    TIMx->SMCR &= ~(TIM_SMCR_SMS);  // disable slave mode controller;
                                       // use instead internal clock CK_INT
(aka "TIMx_CLK from RCC")
    TIMx->PSC = 0; // do not prescale; let the counter receive full CK_INT
frequency

    /* Configure Counter as Upcounter */
```

```
      TIMx->CR1  &=  ~(TIM_CR1_CMS);  //  use  edge-aligned  mode  (i.e.  plain
vanilla up or down counting)
    TIMx->CR1 &= ~(TIM_CR1_DIR); // upcounter mode

    /* Allow Automatic Updating on Overflow Events */
    TIMx->CR1 |= TIM_CR1_ARPE;  // "auto-reload preload enabled"
                                   // transfers preload register to shadow
register every update event, meaning
                                     // we can change ARR and PSC without
having to manually request an update
    TIMx->ARR = 1;  // set max count to 1 as a default so that
                        // we generate update events quickly when we turn
timer on
    TIMx->CR1 &= ~(TIM_CR1_OPM); // do not stop counter at update events

    /* If Master, output trigger signal upon update events */
    if (master_mode) {
        // set update event at as trigger output (TRGO)
        TIMx->CR2 &= ~(TIM_CR2_MMS);
        TIMx->CR2 |= (0b010 << TIM_CR2_MMS_Pos);
    }

    /* Start Running */
    TIMx->EGR |= TIM_EGR_UG;  // manually generate an update to initialize
all shadow registers
    TIMx->CR1 |= TIM_CR1_CEN; // enable counter
}

void generatePWMfreq(TIM_TypeDef * TIMx, uint32_t freq, uint32_t duty_inv)
{
      /*  Generate  a  signal  of  frequency  <freq>  Hz  and  1/<duty_inv>  duty
cycle
       *  This  is  designed  for  applications  where  frequency  matters  most
(e.g. tone generation).
     *
     * <timer> TIM2, TIM3, TIM4, or TIM5
       *  <freq>  in  Hz,  can  range  from  TIMER_BASE_FREQ  /  (ARR  size)  to
TIMER_BASE_FREQ
      *                  though it should be noted that division rounding
error approaches
```

```
     *               a maximum of ~50% as freq approaches TIMER_BASE_FREQ
     * <duty_inv> is 1/duty, and it is also subjected to divison rounding
error
     *               as duty approaches TIMER_BASE_FREQ / freq
     *
     * TIM2 and TIM5 have 32 bit ARR, which at 84MHz base freq, will not
generate freqs below 0.196Hz
     * TIM3 and TIM4 have 16 bit ARR, which at 84MHz base freq, will not
generate freqs below 1282Hz
     */

       TIMx->CR1  |=  TIM_CR1_UDIS;    //  Disable  update  events  because
apparently it would be bad
                                  // if the timer by chance tried to update
the shadow registers
                                   // while we're writing new values in the
preload registers.
    if (freq != 0) {
        uint32_t num_ticks = 84000000UL/freq;
                                     // Both frequencies are in Hz, so the
ratio
                                    // represents number of periods of base
freq per period of tone freq.
                                    // Integer division shouldn't plague us
with too much rounding error
                                     // if TIMER_BASE_FREQ is sufficiently
large relative to max freq,

        TIMx->ARR  = num_ticks;          // Set value we count up to.
          TIMx->CCR1 = num_ticks/duty_inv; // Set the count threshold at
which capture/compare channel 1 output is high/low.
                                          // Dividing by duty_inv is what
implements the duty cycle
    } else {
        TIMx->ARR = 1;  // set to 1 to that it's constantly updating
        TIMx->CCR1 = 0; // set output to resting state
    }

    TIMx->CR1 &= ~(TIM_CR1_UDIS); // enable update events;
```

```
                                                     // (i.e. send update signal every time
counter under/over-flows)
}

inline void start_micros(TIM_TypeDef * TIMx, uint32_t us) {
    TIMx->ARR = us;                    // Set timer max count
    TIMx->EGR |= TIM_EGR_UG;           // Force update
    TIMx->SR &= ~(TIM_SR_UIF);         // Clear UIF
    TIMx->CNT = 0;                     // Reset count
}

inline void wait_micros(TIM_TypeDef * TIMx) {
    while(!(TIMx->SR & TIM_SR_UIF)); // Wait for UIF to go high
}

void delay_millis(TIM_TypeDef * TIMx, uint32_t ms) {
  start_micros(TIMx, ms*1000);
  wait_micros(TIMx);
}

void delay_micros(TIM_TypeDef * TIMx, uint32_t us) {
  start_micros(TIMx, us);
  wait_micros(TIMx);
}
```

Appendix H: Vector Transformations Library

Header File: transformations.h

```
#ifndef __TRANSFORMATIONS_H__
#define __TRANSFORMATIONS_H__

static const float sin1deg =
0.0174524064372835128194189785163161924722527203071396426836124276f;
static const float cos1deg =
0.9998476951563912391570115588139148516927403105831859396583207145f;

void multiply4x4byVector(float A[4][4], float b[4]);
void translate(float vector[4], float tx, float ty, float tz);
void scale(float vector[4], float sx, float sy, float sz);
void rotateX(float vector[4], int deg);
void rotateY(float vector[4], int deg);
void rotateZ(float vector[4], int deg);
void normalizeHomogenousCoordinates(float coordinates[4]);
void projectOrthogonally(float homogenousCoordinates[4], float
xyCoordinates[2]);

#endif
```

Source Code: transformations.c

```
#include "transformations.h"

/**
 * Using homogenous coordinates so that all affine transformations on a
vector
 * are the product of a transformation matrix and that vector
 */

void multiply4x4byVector(float A[4][4], float b[4]) {
    float result[4];

    for (int i = 0; i < 4; ++i) {
        float dotProduct = 0.0f;
```

```
        for (int k = 0; k < 4; ++k) {
            dotProduct += A[i][k] * b[k];
        }
        result[i] = dotProduct;
    }

    for (int i = 0; i < 4; ++i) {
        b[i] = result[i];
    }
}

void translate(float vector[4], float tx, float ty, float tz) {
      //  Translates  a  vector  by  tx  in  the  x  direction,  ty  in  the  y
direction, and tz in the z direction
    float translationMatrix[4][4] = {
        { 1,  0,  0,  tx},
        { 0,  1,  0,  ty},
        { 0,  0,  1,  tz},
        { 0,  0,  0,  1}
    };
    multiply4x4byVector(translationMatrix, vector);
}

void scale(float vector[4], float sx, float sy, float sz) {
     // Scales a vector by sx in the x direction, sy in the y direction,
and sz in the z direction
    float scalingMatrix[4][4] = {
        {sx,  0,  0,  0},
        { 0, sy,  0,  0},
        { 0,  0, sz,  0},
        { 0,  0,  0,  1}
    };
    multiply4x4byVector(scalingMatrix, vector);
}

void rotatePositiveX(float vector[4]) {
    // Rotates a vector about the x axis in the positive direction
    float rotationMatrix[4][4] = {
        {1,    0    ,      0    ,  0},
        {0, cos1deg, -sin1deg,  0},
```

```
        {0, sin1deg,  cos1deg,  0},
        {0,    0   ,     0   ,  1}
    };
    multiply4x4byVector(rotationMatrix, vector);
}

void rotateNegativeX(float vector[4]) {
    // Rotates a vector about the x axis in the negative direction
    float rotationMatrix[4][4] = {
        {1,     0   ,    0   ,  0},
        {0,  cos1deg, sin1deg,  0},
        {0, -sin1deg, cos1deg,  0},
        {0,     0   ,    0   ,  1}
    };
    multiply4x4byVector(rotationMatrix, vector);
}

void rotateX(float vector[4], int deg) {
    // Rotates a vector about the x axis
    if (deg == 0) return;
    else if (deg > 0) {
        for (int i = 0; i < deg; ++ i) {
            rotatePositiveX(vector);
        }
    } else {
        for (int i = 0; i < -deg; ++ i) {
            rotateNegativeX(vector);
        }
    }
}

void rotatePositiveY(float vector[4]) {
    // Rotates a vector about the y axis in the positive direction
    float rotationMatrix[4][4] = {
        { cos1deg, 0, sin1deg,  0},
        {    0   , 1,    0   ,  0},
        {-sin1deg, 0, cos1deg,  0},
        {    0   , 0,    0   ,  1}
    };
    multiply4x4byVector(rotationMatrix, vector);
```

```
}

void rotateNegativeY(float vector[4]) {
    // Rotates a vector about the y axis in the negative direction
    float rotationMatrix[4][4] = {
        {cos1deg, 0, -sin1deg,  0},
        {   0   , 1,     0   ,  0},
        {sin1deg, 0,  cos1deg,  0},
        {   0   , 0,     0   ,  1}
    };
    multiply4x4byVector(rotationMatrix, vector);
}

void rotateY(float vector[4], int deg) {
    // Rotates a vector about the y axis
    if (deg == 0) return;
    else if (deg > 0) {
        for (int i = 0; i < deg; ++ i) {
            rotatePositiveY(vector);
        }
    } else {
        for (int i = 0; i < -deg; ++ i) {
            rotateNegativeY(vector);
        }
    }
}

void rotatePositiveZ(float vector[4]) {
    // Rotates a vector about the z axis in the positive direction
    float rotationMatrix[4][4] = {
        {cos1deg, -sin1deg,  0,  0},
        {sin1deg,  cos1deg,  0,  0},
        {   0   ,     0   ,  1,  0},
        {   0   ,     0   ,  0,  1}
    };
    multiply4x4byVector(rotationMatrix, vector);
}

void rotateNegativeZ(float vector[4]) {
    // Rotates a vector about the z axis in the negative direction
```

```
    float rotationMatrix[4][4] = {
        { cos1deg, sin1deg,  0,  0},
        {-sin1deg, cos1deg,  0,  0},
        {    0   ,    0   ,  1,  0},
        {    0   ,    0   ,  0,  1}
    };
    multiply4x4byVector(rotationMatrix, vector);
}

void rotateZ(float vector[4], int deg) {
    // Rotates a vector about the z axis
    if (deg == 0) return;
    else if (deg > 0) {
        for (int i = 0; i < deg; ++ i) {
            rotatePositiveZ(vector);
        }
    } else {
        for (int i = 0; i < -deg; ++ i) {
            rotateNegativeZ(vector);
        }
    }
}

void normalizeHomogenousCoordinates(float coordinates[4]) {
    // Normalizes the homogenous coordinates to have a w component of 1.0
    for (int i = 0; i < 3; ++ i) {
        coordinates[i] /= coordinates[3];
    }
    coordinates[3] = 1.0f;
}

// https://en.wikipedia.org/wiki/Orthographic_projection
void     projectOrthogonally(float     homogenousCoordinates[4],     float
xyCoordinates[2]) {
    // Project a vector onto the Y-Z plane
    normalizeHomogenousCoordinates(homogenousCoordinates);
    xyCoordinates[0] = homogenousCoordinates[1];
    xyCoordinates[1] = homogenousCoordinates[2];
}
```

Appendix I: Cube Transformations Library

Header File: `cubetransformations.h`

```
#ifndef __CUBETRANSFORMATIONS_H__
#define __CUBETRANSFORMATIONS_H__

#include "transformations.h"
#include <stdint.h>

#define LEFT_CUBE    0
#define RIGHT_CUBE   1

static const int FRONT_RIGHT_UP_IDX     = 0;
static const int FRONT_RIGHT_DOWN_IDX   = 1;
static const int FRONT_LEFT_DOWN_IDX    = 2;
static const int FRONT_LEFT_UP_IDX      = 3;
static const int BACK_LEFT_UP_IDX       = 4;
static const int BACK_LEFT_DOWN_IDX     = 5;
static const int BACK_RIGHT_DOWN_IDX    = 6;
static const int BACK_RIGHT_UP_IDX      = 7;

static const float origin[4] = {0, 0, 0, 1};

static float leftCubeVertices[8][4] = {
    { 1, -1,  1,  1},
    { 1, -1, -1,  1},
    { 1, -3, -1,  1},
    { 1, -3,  1,  1},
    {-1, -3,  1,  1},
    {-1, -3, -1,  1},
    {-1, -1, -1,  1},
    {-1, -1,  1,  1}
};

static float rightCubeVertices[8][4] = {
    { 1,  3,  1,  1},
    { 1,  3, -1,  1},
    { 1,  1, -1,  1},
```

```
    { 1,   1,   1,   1},
    {-1,   1,   1,   1},
    {-1,   1,  -1,   1},
    {-1,   3,  -1,   1},
    {-1,   3,   1,   1}
};

static float cubeEdges[38][4];

static float cubeVectorDataFloats[38][2];

void translateCube(int cube, float tx, float ty, float tz);
void scaleCube(int cube, float sx, float sy, float sz);
void rotateXCube(int cube, int deg);
void rotateYCube(int cube, int deg);
void rotateZCube(int cube, int deg);
void calculateCubeVectorData(uint16_t vectorData[2][38]);

#endif
```

Source Code: cubetransformations.c

```
#include "cubetransformations.h"

/**
 * Transformations to the cube are performed by transforming each vertex
of the cube.
 */

void translateCube(int cube, float tx, float ty, float tz) {
    // Translate a cube by tx in the x direction, ty in the y direction,
and tz in the z direction
    if (cube == LEFT_CUBE) {
        for (int i = 0; i < 8; ++i) {
            translate(leftCubeVertices[i], tx, ty, tz);
        }
    } else {
        for (int i = 0; i < 8; ++i) {
            translate(rightCubeVertices[i], tx, ty, tz);
        }
```

```
    }
}

void scaleCube(int cube, float sx, float sy, float sz) {
     // Scales a cube by tx in the x direction, ty in the y direction, and
tz in the z direction
    if (cube == LEFT_CUBE) {
        for (int i = 0; i < 8; ++i) {
            scale(leftCubeVertices[i], sx, sy, sz);
        }
    } else {
        for (int i = 0; i < 8; ++i) {
            scale(rightCubeVertices[i], sx, sy, sz);
        }
    }
}

void rotateXCube(int cube, int deg) {
    // Rotates a cube about the x axis
    if (cube == LEFT_CUBE) {
        for (int i = 0; i < 8; ++i) {
            rotateX(leftCubeVertices[i], deg);
        }
    } else {
        for (int i = 0; i < 8; ++i) {
            rotateX(rightCubeVertices[i], deg);
        }
    }
}

void rotateYCube(int cube, int deg) {
    // Rotates a cube about the y axis
    if (cube == LEFT_CUBE) {
        for (int i = 0; i < 8; ++i) {
            rotateY(leftCubeVertices[i], deg);
        }
    } else {
        for (int i = 0; i < 8; ++i) {
            rotateY(rightCubeVertices[i], deg);
        }
```

```
    }
}

void rotateZCube(int cube, int deg) {
     // Rotates the left cube about the x=0,y=-2 axis and the right cube about the x=0,y=2 axis
    translateCube(cube, 0, 2-(4*cube), 0);
    if (cube == LEFT_CUBE) {
        for (int i = 0; i < 8; ++i) {
            rotateZ(leftCubeVertices[i], deg);
        }
    } else {
        for (int i = 0; i < 8; ++i) {
            rotateZ(rightCubeVertices[i], deg);
        }
    }
    translateCube(cube, 0, (4*cube)-2, 0);
}

void subtractCubeVertices(int cube, int idx1, int idx2, float result[4]) {
      // Homogenous coordinate subtraction in order to get the vector between two poitns
    if (cube == LEFT_CUBE) {
        float weight1 = idx1 < 0 ? origin[3] : leftCubeVertices[idx1][3];
        float weight2 = idx2 < 0 ? origin[3] : leftCubeVertices[idx2][3];
        for (int i = 0; i < 3; ++i) {
            float a = idx1 < 0 ? origin[i] : leftCubeVertices[idx1][i];
            float b = idx2 < 0 ? origin[i] : leftCubeVertices[idx2][i];
            result[i] = weight2 * a - weight1 * b;
        }
        result[3] = weight1 * weight2;
    } else {
        float weight1 = idx1 < 0 ? origin[3] : rightCubeVertices[idx1][3];
        float weight2 = idx2 < 0 ? origin[3] : rightCubeVertices[idx2][3];
        for (int i = 0; i < 3; ++i) {
            float a = idx1 < 0 ? origin[i] : rightCubeVertices[idx1][i];
            float b = idx2 < 0 ? origin[i] : rightCubeVertices[idx2][i];
            result[i] = weight2 * a - weight1 * b;
        }
        result[3] = weight1 * weight2;
```

```
    }
}

void calculateCubeEdges() {
     // Calculate the edges of the cube by taking the difference between
consecutive vertices
    // Start and end at the origin

     subtractCubeVertices(LEFT_CUBE,            -1,                      -1,
cubeEdges[0]);
     subtractCubeVertices(LEFT_CUBE,   FRONT_RIGHT_UP_IDX,               -1,
cubeEdges[1]);
                 subtractCubeVertices(LEFT_CUBE,         BACK_RIGHT_UP_IDX,
FRONT_RIGHT_UP_IDX,      cubeEdges[2]);
                subtractCubeVertices(LEFT_CUBE,        BACK_RIGHT_DOWN_IDX,
BACK_RIGHT_UP_IDX,       cubeEdges[3]);
                 subtractCubeVertices(LEFT_CUBE,         BACK_LEFT_DOWN_IDX,
BACK_RIGHT_DOWN_IDX,     cubeEdges[4]);
                  subtractCubeVertices(LEFT_CUBE,          BACK_LEFT_UP_IDX,
BACK_LEFT_DOWN_IDX,      cubeEdges[5]);
                 subtractCubeVertices(LEFT_CUBE,         FRONT_LEFT_UP_IDX,
BACK_LEFT_UP_IDX,        cubeEdges[6]);
                subtractCubeVertices(LEFT_CUBE,        FRONT_LEFT_DOWN_IDX,
FRONT_LEFT_UP_IDX,       cubeEdges[7]);
               subtractCubeVertices(LEFT_CUBE,        FRONT_RIGHT_DOWN_IDX,
FRONT_LEFT_DOWN_IDX,     cubeEdges[8]);
                 subtractCubeVertices(LEFT_CUBE,        FRONT_RIGHT_UP_IDX,
FRONT_RIGHT_DOWN_IDX,    cubeEdges[9]);
                 subtractCubeVertices(LEFT_CUBE,         FRONT_LEFT_UP_IDX,
FRONT_RIGHT_UP_IDX,      cubeEdges[10]);
                subtractCubeVertices(LEFT_CUBE,        FRONT_LEFT_DOWN_IDX,
FRONT_LEFT_UP_IDX,       cubeEdges[11]);
                 subtractCubeVertices(LEFT_CUBE,         BACK_LEFT_DOWN_IDX,
FRONT_LEFT_DOWN_IDX,     cubeEdges[12]);
                  subtractCubeVertices(LEFT_CUBE,          BACK_LEFT_UP_IDX,
BACK_LEFT_DOWN_IDX,      cubeEdges[13]);
                 subtractCubeVertices(LEFT_CUBE,         BACK_RIGHT_UP_IDX,
BACK_LEFT_UP_IDX,        cubeEdges[14]);
                subtractCubeVertices(LEFT_CUBE,        BACK_RIGHT_DOWN_IDX,
BACK_RIGHT_UP_IDX,       cubeEdges[15]);
```

```
            subtractCubeVertices(LEFT_CUBE,       FRONT_RIGHT_DOWN_IDX,
BACK_RIGHT_DOWN_IDX,    cubeEdges[16]);
             subtractCubeVertices(LEFT_CUBE,        FRONT_RIGHT_UP_IDX,
FRONT_RIGHT_DOWN_IDX,   cubeEdges[17]);
         subtractCubeVertices(LEFT_CUBE,                            -1,
FRONT_RIGHT_UP_IDX,     cubeEdges[18]);

   subtractCubeVertices(RIGHT_CUBE,          -1,                    -1,
cubeEdges[19]);
    subtractCubeVertices(RIGHT_CUBE, FRONT_RIGHT_UP_IDX,            -1,
cubeEdges[20]);
                subtractCubeVertices(RIGHT_CUBE,     BACK_RIGHT_UP_IDX,
FRONT_RIGHT_UP_IDX,     cubeEdges[21]);
               subtractCubeVertices(RIGHT_CUBE,     BACK_RIGHT_DOWN_IDX,
BACK_RIGHT_UP_IDX,      cubeEdges[22]);
               subtractCubeVertices(RIGHT_CUBE,     BACK_LEFT_DOWN_IDX,
BACK_RIGHT_DOWN_IDX,    cubeEdges[23]);
                subtractCubeVertices(RIGHT_CUBE,     BACK_LEFT_UP_IDX,
BACK_LEFT_DOWN_IDX,     cubeEdges[24]);
                subtractCubeVertices(RIGHT_CUBE,     FRONT_LEFT_UP_IDX,
BACK_LEFT_UP_IDX,       cubeEdges[25]);
               subtractCubeVertices(RIGHT_CUBE,     FRONT_LEFT_DOWN_IDX,
FRONT_LEFT_UP_IDX,      cubeEdges[26]);
              subtractCubeVertices(RIGHT_CUBE,     FRONT_RIGHT_DOWN_IDX,
FRONT_LEFT_DOWN_IDX,    cubeEdges[27]);
               subtractCubeVertices(RIGHT_CUBE,     FRONT_RIGHT_UP_IDX,
FRONT_RIGHT_DOWN_IDX,   cubeEdges[28]);
                subtractCubeVertices(RIGHT_CUBE,     FRONT_LEFT_UP_IDX,
FRONT_RIGHT_UP_IDX,     cubeEdges[29]);
               subtractCubeVertices(RIGHT_CUBE,     FRONT_LEFT_DOWN_IDX,
FRONT_LEFT_UP_IDX,      cubeEdges[30]);
               subtractCubeVertices(RIGHT_CUBE,     BACK_LEFT_DOWN_IDX,
FRONT_LEFT_DOWN_IDX,    cubeEdges[31]);
                subtractCubeVertices(RIGHT_CUBE,     BACK_LEFT_UP_IDX,
BACK_LEFT_DOWN_IDX,     cubeEdges[32]);
                subtractCubeVertices(RIGHT_CUBE,     BACK_RIGHT_UP_IDX,
BACK_LEFT_UP_IDX,       cubeEdges[33]);
               subtractCubeVertices(RIGHT_CUBE,     BACK_RIGHT_DOWN_IDX,
BACK_RIGHT_UP_IDX,      cubeEdges[34]);
```

```
                  subtractCubeVertices(RIGHT_CUBE,     FRONT_RIGHT_DOWN_IDX,
BACK_RIGHT_DOWN_IDX,     cubeEdges[35]);
                    subtractCubeVertices(RIGHT_CUBE,     FRONT_RIGHT_UP_IDX,
FRONT_RIGHT_DOWN_IDX,    cubeEdges[36]);
             subtractCubeVertices(RIGHT_CUBE,                            -1,
FRONT_RIGHT_UP_IDX,      cubeEdges[37]);
}

void calculateCubeVectorDataFloats() {
    // Calculate the edge values while still using floats
    calculateCubeEdges();
    for (int i = 0; i < 38; ++i) {
        projectOrthogonally(cubeEdges[i], cubeVectorDataFloats[i]);
    }
}

#define NEG (1<<10)
void calculateCubeVectorData(uint16_t vectorData[2][38]) {
    // Calculate the screen coordinate values (-512 to 512) for the edgees
    calculateCubeVectorDataFloats();
    for (int i = 0; i < 38; ++i) {
        for (int j = 0; j < 2; ++j) {
            float val = 100 * cubeVectorDataFloats[i][j];
            if (val >= 0) {
                vectorData[j][i] = (uint16_t)(val);
            } else {
                vectorData[j][i] = NEG|((uint16_t)(-val));
            }
        }
    }
}
```

Appendix J: Source Code: main.c

```
// https://www.tutorialspoint.com/computer_graphics/3d_transformation.htm
#include "stm32f4xx.h"
#include "clock.h"
#include "gpio.h"
#include "cubetransformations.h"
#include "timers.h"
#include "spi.h"

/* Pin Assignments */
#define LED_GPIO GPIOB
#define LED_PIN GPIO_PB10
#define VEC_CLK            GPIO_PB6  // white wire
#define GOb                GPIO_PA0  // green wire
#define COUNT_LDb          GPIO_PA4  // blue wire
#define COLOR_LD           GPIO_PA8  // yellow wire
#define BLANKb             GPIO_PB4  // dark green wire
#define X_SHIFT_REG_CLK    GPIO_PA5  // orange wire
#define X_SHIFT_REG_LD     GPIO_PA6  // red wire
#define X_SHIFT_REG_DATA   GPIO_PA7  // brown wire
#define Y_SHIFT_REG_CLK    GPIO_PC10 // orange wire
#define Y_SHIFT_REG_LD     GPIO_PC11 // red wire
#define Y_SHIFT_REG_DATA   GPIO_PC12 // brown wire

/* Peripheral Assignments */
#define DELAY_TIM        TIM3
#define VEC_MASTER_CLK   TIM4
#define VEC_TIMER        TIM5
#define X_DMA            DMA1
#define Y_DMA            DMA1
#define X_DMA_STREAM     DMA1_Stream1
#define Y_DMA_STREAM     DMA1_Stream2
#define X_SPI SPI1
#define Y_SPI SPI3

/* Global Vector Vars */
// Bits
#define NEG (1<<10)     // applies negative direction; for use with x or y
data
```

```
#define  BLANK  (1<<0)      //  blanks  current  vector  (i.e.  makes  it
transparent); for use with z data
#define LD_COL (1<<1)  // loads new color data; for use with z data
#define LD_POS (1<<2) // loads new absolute position; for use with z data
// Buffer Struct
#define BUFFER_SIZE 200
typedef struct {
    uint16_t x[BUFFER_SIZE]; // stores x data to be sent to hardware
    uint16_t y[BUFFER_SIZE]; // stores y data to be sent to hardware
    uint16_t z[BUFFER_SIZE]; // stores software controls
     unsigned int top;         // points to where the first free chunk of
memory is
     unsigned int anim_index; // points to the first vector of the first
animated object is (used in write mode)
    unsigned int read_index; // points to where the first unread vector is
(used in read mode)
    //
    // Here's a picture to visualize (not to scale).
    //
    //                  X            Y          Z
    //            0 --------    --------    ----
    //               (vector data that does often
    //                not change frame to frame)
    //  anim_index --------    --------    ----
    //               (vector data that does often
    //                change frame to frame)
    //         top --------    --------    ----
    //               (unusued memory)
    // BUFFER_SIZE --------    --------    ----
    //
} vector_buffer;
// Actual Buffers
vector_buffer buff_0_value; // value as in not a pointer
vector_buffer buff_1_value;
// Aliases to Actual Buffers
vector_buffer* buff_r = &buff_0_value; // r for currently being read from
(and drawn to screen)
vector_buffer* buff_w = &buff_1_value; // w for currently being written
// Color State
uint8_t x_color=0; // current color
```

```
uint8_t y_color=0;
// Current Instruction Being Drawn to Screen
uint16_t curr_x=0;
uint16_t curr_y=0;
uint16_t curr_z=0;
// Buffer Switch Request
unsigned int buffer_swap_req=0; // draw-er (TIM5_IRQHandler) issues the
request
                                 // and when comput-er (main loop) is done
calculating the next frame,
                                // the request is granted
unsigned int drawer_halted=0; // lets the comput-er know if it needs to
restart the draw-er
                              // when a buffer swap is performed

/* Font Data (translated from Atari Gravitar ROM data) */
uint16_t space_x[1] = {120};
uint16_t space_y[1] = {0};
uint16_t space_z[1] = {0};

uint16_t zero_x[5] = {0, 80, 0, NEG|80, 120};
uint16_t zero_y[5] = {120, 0, NEG|120, 0, 0};
uint16_t zero_z[5] = {0,0,0,0,BLANK};

uint16_t one_x[3] = {40,0,80};
uint16_t one_y[3] = {120,NEG|120,0};
uint16_t one_z[3] = {BLANK,0,BLANK};

uint16_t two_x[7] = {0,80,0,NEG|80,0,80,40};
uint16_t two_y[7] = {120,0,NEG|60,0,NEG|60,0,0};
uint16_t two_z[7] = {BLANK,0,0,0,0,0,BLANK};

uint16_t three_x[7] = {0,80,0,NEG|80,0,80,40};
uint16_t three_y[7] = {120,0,NEG|120,0,60,0,NEG|60};
uint16_t three_z[7] = {BLANK,0,0,0,BLANK,0,BLANK};

uint16_t four_x[6] = {0,0,80,0,0,40};
uint16_t four_y[6] = {120,NEG|60,0,60,NEG|120,0};
uint16_t four_z[6] = {BLANK,0,0,BLANK,0,BLANK};
```

```
uint16_t five_x[6] = {80,0,NEG|80,0,80,40};
uint16_t five_y[6] = {0,60,0,60,0,NEG|120};
uint16_t five_z[6] = {0,0,0,0,0,BLANK};

uint16_t six_x[6] = {0,80,0,NEG|80,0,120};
uint16_t six_y[6] = {60,0,NEG|60,0,120,NEG|120};
uint16_t six_z[6] = {BLANK,0,0,0,0,BLANK};

uint16_t seven_x[4] = {0,80,0,40};
uint16_t seven_y[4] = {120,0,NEG|120,0};
uint16_t seven_z[4] = {BLANK,0,0,BLANK};

uint16_t eight_x[7] = {0,80,0,NEG|80,0,80,40};
uint16_t eight_y[7] = {120,0,NEG|120,0,60,0,NEG|60};
uint16_t eight_z[7] = {0,0,0,0,BLANK,0,BLANK};

uint16_t nine_x[6] = {80,NEG|80,0,80,0,40};
uint16_t nine_y[6] = {60,0,60,0,NEG|120,0};
uint16_t nine_z[6] = {BLANK,0,0,0,0,BLANK};

uint16_t a_x[7] = {0,40,40,0,NEG|80,80,40};
uint16_t a_y[7] = {80,40,NEG|40,NEG|80,40,0,NEG|40};
uint16_t a_z[7] = {0,0,0,0,BLANK,0,BLANK};

uint16_t b_x[12] = {0,60,20,0,NEG|20,NEG|60,60,20,0,NEG|20,NEG|60,120};
uint16_t                                b_y[12]                                =
{120,0,NEG|20,NEG|20,NEG|20,0,0,NEG|20,NEG|20,NEG|20,0,0};
uint16_t b_z[12] = {0,0,0,0,0,0,BLANK,0,0,0,0,BLANK};

uint16_t c_x[5] = {0,80,NEG|80,80,40};
uint16_t c_y[5] = {120,0,NEG|120,0,0};
uint16_t c_z[5] = {0,0,BLANK,0,BLANK};

uint16_t d_x[7] = {0,40,40,0,NEG|40,NEG|40,120};
uint16_t d_y[7] = {120,0,NEG|40,NEG|40,NEG|40,0,0};
uint16_t d_z[7] = {0,0,0,0,0,0,BLANK};

uint16_t e_x[7] = {80,NEG|80,0,80,NEG|20,NEG|60,120};
uint16_t e_y[7] = {0,0,120,0,NEG|60,0,NEG|60};
uint16_t e_z[7] = {0,BLANK,0,0,BLANK,0,BLANK};
```

```
uint16_t f_x[5] = {0,80,NEG|20,NEG|60,120};
uint16_t f_y[5] = {120,0,NEG|60,0,NEG|60};
uint16_t f_z[5] = {0,0,BLANK,0,BLANK};

uint16_t g_x[8] = {0,80,0,NEG|40,40,0,NEG|80,120};
uint16_t g_y[8] = {120,0,NEG|40,NEG|40,0,NEG|40,0,0};
uint16_t g_z[8] = {0,0,0,BLANK,0,0,0,BLANK};

uint16_t h_x[6] = {0,0,80,0,0,40};
uint16_t h_y[6] = {120,NEG|60,0,60,NEG|120,0};
uint16_t h_z[6] = {0,BLANK,0,BLANK,0,BLANK};

uint16_t i_x[6] = {80,NEG|80,80,NEG|40,0,80};
uint16_t i_y[6] = {0,120,0,0,NEG|120,0};
uint16_t i_z[6] = {0,BLANK,0,BLANK,0,BLANK};

uint16_t j_x[5] = {0,40,40,0,40};
uint16_t j_y[5] = {40,NEG|40,0,120,NEG|120};
uint16_t j_z[5] = {BLANK,0,0,0,BLANK};

uint16_t k_x[5] = {0,60,NEG|60,60,60};
uint16_t k_y[5] = {120,0,NEG|60,NEG|60,0};
uint16_t k_z[5] = {0,BLANK,0,0,BLANK};

uint16_t l_x[4] = {0,0,80,40};
uint16_t l_y[4] = {120,NEG|120,0,0};
uint16_t l_z[4] = {BLANK,0,0,BLANK};

uint16_t m_x[5] = {0,40,40,0,40};
uint16_t m_y[5] = {120,NEG|40,40,NEG|120,0};
uint16_t m_z[5] = {0,0,0,0,BLANK};

uint16_t n_x[4] = {0,80,0,40};
uint16_t n_y[4] = {120,NEG|120,120,NEG|120};
uint16_t n_z[4] = {0,0,0,BLANK};

uint16_t o_x[5] = {0, 80, 0, NEG|80, 120};
uint16_t o_y[5] = {120, 0, NEG|120, 0, 0};
uint16_t o_z[5] = {0,0,0,0,BLANK};
```

```
uint16_t p_x[5] = {0,80,0,NEG|80,120};
uint16_t p_y[5] = {120,0,NEG|60,0,NEG|60};
uint16_t p_z[5] = {0,0,0,0,BLANK};

uint16_t q_x[8] = {0,80,0,NEG|40,NEG|40,40,40,40};
uint16_t q_y[8] = {120,0,NEG|80,NEG|40,0,40,NEG|40,0};
uint16_t q_z[8] = {0,0,0,0,0,BLANK,0,BLANK};

uint16_t r_x[7] = {0,80,0,NEG|80,20,60,40};
uint16_t r_y[7] = {120,0,NEG|60,0,0,NEG|60,0};
uint16_t r_z[7] = {0,0,0,0,BLANK,0,BLANK};

uint16_t s_x[6] = {80,0,NEG|80,0,80,40};
uint16_t s_y[6] = {0,60,0,60,0,NEG|120};
uint16_t s_z[6] = {0,0,0,0,0,BLANK};

uint16_t t_x[5] = {0,80,NEG|40,0,80};
uint16_t t_y[5] = {120,0,0,NEG|120,0};
uint16_t t_z[5] = {BLANK,0,BLANK,0,BLANK};

uint16_t u_x[5] = {0,0,80,0,40};
uint16_t u_y[5] = {120,NEG|120,0,120,NEG|120};
uint16_t u_z[5] = {BLANK,0,0,0,BLANK};

uint16_t v_x[4] = {0,40,40,40};
uint16_t v_y[4] = {120,NEG|120,120,NEG|120};
uint16_t v_z[4] = {BLANK,0,0,BLANK};

uint16_t w_x[6] = {0,0,4,4,0,4};
uint16_t w_y[6] = {120,NEG|120,40,NEG|40,120,NEG|120};
uint16_t w_z[6] = {BLANK,0,0,0,0,BLANK};

uint16_t x_x[4] = {80,NEG|80,80,40};
uint16_t x_y[4] = {120,0,NEG|120,0};
uint16_t x_z[4] = {0,BLANK,0,BLANK};

uint16_t y_x[6] = {40,0,NEG|40,80,NEG|40,80};
uint16_t y_y[6] = {0,80,40,0,NEG|40,NEG|80};
uint16_t y_z[6] = {BLANK,0,0,BLANK,0,BLANK};
```

```
uint16_t z_x[5] = {0,80,NEG|80,80,40};
uint16_t z_y[5] = {120,0,NEG|120,0,0};
uint16_t z_z[5] = {BLANK,0,0,0,BLANK};

// Cube Vars
const unsigned int CUBE_VECTOR_DATA_SIZE = 76; // words
#define ARRAY_SIZE 38
uint16_t cubeVectorData[2][ARRAY_SIZE];
uint16_t cubeZData[38] = {
    BLANK, BLANK, 0, 0, 0, 0, 0, 0, 0, 0,
    0, BLANK, 0, BLANK, 0, BLANK, 0, BLANK, BLANK, BLANK,
    BLANK, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    BLANK, 0, BLANK, 0, BLANK, 0, BLANK, BLANK
};

// Keyboard Input Vars
volatile int newCommandAvailable = 0;
volatile uint16_t lastCommand;

/////////////////////////////////////////////////////////////////////////////
/////
// Configuration Functions
/////////////////////////////////////////////////////////////////////////////
/////

void configureGPIOs() {
    // Enable clock to GPIOs
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN;
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOCEN;
    // Timer Signals
    alternateFunctionMode(GPIOB, GPIO_PB6, 2); // VEC_CLK (white wire) alt
func VEC_CLK_CH1
     alternateFunctionMode(GPIOA, GPIO_PA0, 2); // GOb (green wire) alt
func VEC_TIMER_CH1
    // USART2 GPIOA Signals
    alternateFunctionMode(GPIOA, GPIO_PA2, 7);
    alternateFunctionMode(GPIOA, GPIO_PA3, 7);
    // Miscellaneous Controls
```

```
    pinMode(GPIOA, GPIO_PA4, GPIO_OUTPUT); // COUNT_LDb (blue wire)
    pinMode(GPIOA, GPIO_PA8, GPIO_OUTPUT); // COLOR_LD (yellow wire)
    pinMode(GPIOB, GPIO_PB4, GPIO_OUTPUT); // BLANKb (dark green wire)
    // X SPI Signals
        alternateFunctionMode(GPIOA, GPIO_PA5, 5);    // X_SHIFT_REG_CLK
(orange wire) alt func SPI1_SCK
     pinMode(GPIOA, GPIO_PA6, GPIO_OUTPUT);       // X_SHIFT_REG_LD    (red
wire)
    alternateFunctionMode(GPIOA, GPIO_PA7, 5);  // X_SHIFT_REG_DATA (brown
wire)  alt func SPI1_MOSI
    // Y SPI signals
        alternateFunctionMode(GPIOC, GPIO_PC10, 6);  // Y_SHIFT_REG_CLK
(orange wire) alt func SPI3_SCK
     pinMode(GPIOC, GPIO_PC11, GPIO_OUTPUT);      // Y_SHIFT_REG_LD    (red
wire)
    alternateFunctionMode(GPIOC, GPIO_PC12, 6); // Y_SHIFT_REG_DATA (brown
wire)  alt func SPI3_MOSI
    // external debugging LED
    pinMode(LED_GPIO, LED_PIN, GPIO_OUTPUT);
}

void configureDelayTimer() {
    // Enable clock to timer 3
    RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;
    // Configure timer 3
    configureTimer(TIM3);
}

void configureVectorTimers() {
    // Enable clock to timers
    RCC->APB1ENR |= RCC_APB1ENR_TIM4EN;
    RCC->APB1ENR |= RCC_APB1ENR_TIM5EN;
    // Configure VEC_CLK Timer as a master clock
    configureCaptureCompare(VEC_MASTER_CLK);
    configurePWM(VEC_MASTER_CLK, 1);
    generatePWMfreq(VEC_MASTER_CLK, 10000000U, 2U);
    // Configure VEC_TIMER Timer for controlling GOb
    // as a function of the number of pulses that VEC_CLK makes
    configureCaptureCompare(VEC_TIMER);
    configureDuration(VEC_TIMER, 0, 1, 0b010);
```

```
       VEC_TIMER->DIER  |=  TIM_DIER_UIE;  //  enable  interrupt  req.  upon
updating
     NVIC_EnableIRQ(TIM5_IRQn); // enable the interrupt itself; we use it
to feed new vectors
}

void configureUSART2() {
    // Enable clock to USART 2
    RCC->APB1ENR |= RCC_APB1ENR_USART2EN;
    // Config Registers
    USART2->CR1 |= (USART_CR1_UE); // Enable USART
    USART2->CR1 &= ~(USART_CR1_M); // M=0 corresponds to 8 data bits
    USART2->CR2 &= ~(USART_CR2_STOP); // 0b00 corresponds to 1 stop bit
    USART2->CR1 &= ~(USART_CR1_OVER8); // Set to 16 times sampling freq
    // Baud Rate Register
    USART2->BRR |= (45 << USART_BRR_DIV_Mantissa_Pos);
    USART2->BRR |= (0b1001 << USART_BRR_DIV_Fraction_Pos); // 9/16
    // Interrupts
    USART2->CR1 |= (USART_CR1_RXNEIE); // Enable the receive interrupt req
    NVIC_EnableIRQ(USART2_IRQn); // Enable the actual receiver interrupt
    // Turn it on
    USART2->CR1 |= (USART_CR1_RE); // Enable the receiver
}

void configureDMA() {
    // Using DMA 2 Stream 0 for memory-to-memory transfer
    //   Note that in memory-to-memory mode,
    //   the "peripheral" address and controls are for the source
    //   and the "memory" address and controls are for the destionation
    // Enable clock
    RCC->AHB1ENR |= RCC_AHB1ENR_DMA2EN;
    // Disable stream
    DMA2_Stream0->CR &= ~DMA_SxCR_EN;
     while  (DMA2_Stream0->CR  &  DMA_SxCR_EN_Msk);  //  wait  until  stream  is
off
    // Reset all configuration bits
    // - Select channel 0 (channel does not matter for memory-to-memory)
    // - Single transfer mode
    // - Disable double buffer mode
    // - Priority level low (0/4) - will be set to high later
```

```
    // - Memory and peripheral data size are 8 bits - will be set to 16
bits later
    // - Disable memory and peripheral increment mode - will be enabled
later
    // - Disable circular mode
    // - Peripheral-to-memory mode - will be set to memory-to-memory later
    // - Set DMA as the flow controller
    // - Disable DMA interrupts - some will be enabled later
    // - Disable the stream (should already be disabled)
    DMA2_Stream0->CR = 0;
    // Enable the configuration bits we need
    DMA2_Stream0->CR |= (0b10 << DMA_SxCR_PL_Pos);     // high priority
(3/4)
    DMA2_Stream0->CR |= (0b01 << DMA_SxCR_MSIZE_Pos); // 16-bit memory
data size
    DMA2_Stream0->CR |= (0b01 << DMA_SxCR_PSIZE_Pos); // 16-bit peripheral
data size (is this even needed?)
    DMA2_Stream0->CR |= (0b10 << DMA_SxCR_DIR_Pos);   // memory-to-memory
mode
    DMA2_Stream0->CR |= DMA_SxCR_MINC;                  // enable memory
increment mode
    DMA2_Stream0->CR |= DMA_SxCR_PINC;                  // enable peripheral
increment mode
    // We'll re-enable the stream when we are ready to send something
}

//////////////////////////////////////////////////////////////////////////
/////
// Helper Functions
//////////////////////////////////////////////////////////////////////////
/////

void USART2_IRQHandler() {
    // For receiving user control data from a computer
    if (USART2->SR | USART_SR_RXNE) {
        uint16_t message = USART2->DR;
        lastCommand = message;
        newCommandAvailable = 1;
    }
}
```

```
void runDMA(uint16_t * source, uint16_t * destination, unsigned int
numberOfDatas) {
    // Disable stream
    DMA2_Stream0->CR &= ~DMA_SxCR_EN;
     while (DMA2_Stream0->CR & DMA_SxCR_EN_Msk); // wait until stream is
off
    // Clear Status Registers
    DMA2->LIFCR |= (DMA_LIFCR_CTCIF0 | DMA_LIFCR_CHTIF0 | DMA_LIFCR_CTEIF0
|
                    DMA_LIFCR_CDMEIF0 | DMA_LIFCR_CFEIF0);
    // Configure source and destination
    DMA2_Stream0->PAR  = (uint32_t) source;
    DMA2_Stream0->M0AR = (uint32_t) destination;
    // Data transfer length
    DMA2_Stream0->NDTR = (uint32_t) numberOfDatas;
    // Enable stream
    DMA2_Stream0->CR |= DMA_SxCR_EN;
}

void swapBuffers() {
    if (buff_r==&buff_0_value) {
        buff_r = &buff_1_value;
        buff_w = &buff_0_value;
    } else {
        buff_r = &buff_0_value;
        buff_w = &buff_1_value;
    }
    buff_r->read_index=0;
    buff_w->top=buff_w->anim_index; // we consider anything that needs to
be animated as unwritten to begin with
}

void addVectorsToBuffer(uint16_t* x_data, uint16_t* y_data, uint16_t*
z_data, unsigned int length) {
    // Adds an array of <length> vectors to write buffer
    //
    // use DMA to do busywork of copying
    runDMA(x_data, &buff_w->x[buff_w->top], length);
    runDMA(y_data, &buff_w->y[buff_w->top], length);
```

```
    runDMA(z_data, &buff_w->z[buff_w->top], length);
    // increment allocated space
    buff_w->top+=length;
}

void addLoadToBuffer(uint16_t x_pos, uint16_t y_pos, unsigned int red,
unsigned int green, unsigned int blue) {
    // Adds a load instruction to write buffer.
      // A load instruction directly sets beam to an absolute position
onscreen.
    // And while we're at it, it can also set the current color.
    // red:   3 bits
    // green: 3 bits
    // blue:  2 bits
    //
        // write hardware control data (stuff actually sent to shift
registers)
    buff_w->x[buff_w->top] = ((green<<14) | (blue<<12) | x_pos);
    buff_w->y[buff_w->top] = ((red<<13) | ((green>>2)<<12) | y_pos);
    // write software control data
    buff_w->z[buff_w->top] = LD_COL | LD_POS | BLANK;
    // increment allocated space
    buff_w->top++;
}

void fetchNextVector() {
    // Fetch next vector
    curr_x = buff_r->x[buff_r->read_index];
    curr_y = buff_r->y[buff_r->read_index];
    curr_z = buff_r->z[buff_r->read_index];
    buff_r->read_index++;
    if (buff_r->read_index >= buff_r->top) {
        buff_r->read_index = 0;
        buffer_swap_req=1; // draw-er makes the request
    }
}

void beginDrawing() {
    // load in initial vector to get things started
```

```
     // this first vector should probably load a starting position and
color
    X_SPI->DR = buff_r->x[buff_r->read_index];
    Y_SPI->DR = buff_r->y[buff_r->read_index];
    // now generate an update to start the interrupt cycle
    VEC_TIMER->DIER |= TIM_DIER_UIE;
     // Since we aren't drawing a vector yet, we need to manually ensure
the data have time to be loaded in,
    // and the beam has time to settle.
     // Note that the compare value CCR1 is as high as the auto-reload
value, so the output GOb never activates
    // and draws a vector accidentally.
    //
    generateDuration(VEC_TIMER, 100, 100);
}

// This is the main function for handling drawing stuff
void TIM5_IRQHandler() {
    // Clear interrupt flag
    VEC_TIMER->SR &= ~(TIM_SR_UIF);
     // If we're waiting on the comput-er (main loop) to finish the next
frame, stop drawing
    if (buffer_swap_req) {
            drawer_halted=1; // lets comput-er know it needs to restart
draw-er
        return; // stop drawing
    }
    // Output the previous vector's data by strobing shift reg latch
    digitalWrite(GPIOA, X_SHIFT_REG_LD, GPIO_HIGH);
    digitalWrite(GPIOC, Y_SHIFT_REG_LD, GPIO_HIGH);
    digitalWrite(GPIOA, X_SHIFT_REG_LD, GPIO_LOW);
    digitalWrite(GPIOC, Y_SHIFT_REG_LD, GPIO_LOW);
    // Strobe color latch if we need to
    if (curr_z&LD_COL) {
        digitalWrite(GPIOA, COLOR_LD, GPIO_HIGH);
        digitalWrite(GPIOA, COLOR_LD, GPIO_LOW);
    }
    // Strobe counter parallel load if we need to
    // This moves the beam to absolute position (X,Y)
    if (curr_z & LD_POS) {
```

```
        // Blank colors for absolute loads
        digitalWrite(GPIOB, BLANKb,GPIO_LOW);
        // Strobe the counter parallel load
        digitalWrite(GPIOA, COUNT_LDb, GPIO_LOW);
        digitalWrite(GPIOA, COUNT_LDb, GPIO_HIGH);
        // Grab the next vector
        fetchNextVector();
        // Give some time for the beam to settle
          // but set compare value CCR1 so that GOb never activates and
accidentally draws a vector
        generateDuration(VEC_TIMER, 1028, 1028);
    } else {
        // Blank colors if we need to
        if (curr_z&BLANK) {
            digitalWrite(GPIOB,BLANKb,GPIO_LOW);
        } else {
            digitalWrite(GPIOB,BLANKb,GPIO_HIGH);
        }
        // Grab the next vector
        fetchNextVector();
        // Activate GOb for 1024 pulses
         // This runs the BRM's and Up/Down counters so that they draw out
a line
         // The compare functionality needs a few cycles to get ready after
the timer is enabled,
        // hence the offset by 5 ticks.
        generateDuration(VEC_TIMER, 1028, 5);
    }
    // Send out the next vector over SPI
    X_SPI->DR = curr_x;
    Y_SPI->DR = curr_y;
     // Note that we are expecting the SPI to finish sending before 1028
pulses of VEC_CLK
    // This is reasonable; it takes time to draw
}

void helloGamerText() {
    addLoadToBuffer(260, 800, 0b000, 0b100, 0b10);
    addVectorsToBuffer(h_x,h_y,h_z,6);
    addVectorsToBuffer(e_x,e_y,e_z,7);
```

```
    addVectorsToBuffer(l_x,l_y,l_z,4);
    addVectorsToBuffer(l_x,l_y,l_z,4);
    addVectorsToBuffer(o_x,o_y,o_z,5);
    addLoadToBuffer(185, 150, 0b010, 0b000, 0b10);
    addVectorsToBuffer(g_x,g_y,g_z,8);
    addVectorsToBuffer(a_x,a_y,a_z,7);
    addVectorsToBuffer(m_x,m_y,m_z,5);
    addVectorsToBuffer(e_x,e_y,e_z,7);
    addVectorsToBuffer(r_x,r_y,r_z,7);
    addVectorsToBuffer(s_x,s_y,s_z,6);
    buff_w->anim_index=buff_w->top;
}

void vectorDisplayText() {
    addLoadToBuffer(260, 800, 0b000, 0b100, 0b10);
    addVectorsToBuffer(v_x,v_y,v_z,4);
    addVectorsToBuffer(e_x,e_y,e_z,7);
    addVectorsToBuffer(c_x,c_y,c_z,5);
    addVectorsToBuffer(t_x,t_y,t_z,5);
    addVectorsToBuffer(o_x,o_y,o_z,5);
    addVectorsToBuffer(r_x,r_y,r_z,7);
    addLoadToBuffer(185, 150, 0b010, 0b000, 0b10);
    addVectorsToBuffer(d_x,d_y,d_z,7);
    addVectorsToBuffer(i_x,i_y,i_z,6);
    addVectorsToBuffer(s_x,s_y,s_z,6);
    addVectorsToBuffer(p_x,p_y,p_z,5);
    addVectorsToBuffer(l_x,l_y,l_z,4);
    addVectorsToBuffer(a_x,a_y,a_z,7);
    addVectorsToBuffer(y_x,y_y,y_z,7);
    buff_w->anim_index=buff_w->top;
}

////////////////////////////////////////////////////////////////////////////
/////
// Main
////////////////////////////////////////////////////////////////////////////
/////

int main(void) {
    /* Configure All Peripherals */
```

```
    // peripheral configurations from other libraries
    configure84MHzClock();
    RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;
    configureSPI(SPI1);
    RCC->APB1ENR |= RCC_APB1ENR_SPI3EN;
    configureSPI(SPI3);
    __enable_irq();
    // peripheral configurations defined here in main
    configureGPIOs();
    configureDelayTimer();
    configureVectorTimers();
    configureUSART2();
    configureDMA();

    /* Initialize Frame Buffers */
    // store static images in first frame buffer
    helloGamerText();
    // store static images in second frame buffer
    swapBuffers();
    helloGamerText();

    /* Initialize Vector Generator */
    // drives GOb high when it is done
    // so that we aren't drawing anything when we perform the initial load
     VEC_TIMER->DIER &= ~(TIM_DIER_UIE); // we don't want to jump into the
interrupt cycle just yet
    generateDuration(VEC_TIMER, 1, 2);
    // now we are ready to start drawing
    beginDrawing();

    /* Initialize 3D Calculations */
    rotateZCube(LEFT_CUBE, 45);
    rotateZCube(RIGHT_CUBE,45);
    rotateYCube(LEFT_CUBE, 45);
    rotateYCube(RIGHT_CUBE,45);
    calculateCubeVectorData(cubeVectorData);

    /* Initialize Debugging LED */
     digitalWrite(GPIOA, LED_PIN, 1); // Turn on by default to shows signs
of life
```

```
/* Main Loop */
while (1) {
    // Respond to Keyboard Input
    if (newCommandAvailable) {
        newCommandAvailable = 0;
        switch (lastCommand) {
            case ((uint16_t)'w'):
                rotateYCube(LEFT_CUBE,  1);
                break;
            case ((uint16_t)'s'):
                rotateYCube(LEFT_CUBE,  -1);
                break;
            case ((uint16_t)'a'):
                rotateZCube(LEFT_CUBE,  1);
                break;
            case ((uint16_t)'d'):
                rotateZCube(LEFT_CUBE,  -1);
                break;
            case ((uint16_t)'q'):
                scaleCube(LEFT_CUBE,  1.01, 1.01, 1.01);
                break;
            case ((uint16_t)'e'):
                scaleCube(LEFT_CUBE,  0.99, 0.99, 0.99);
                break;
            case ((uint16_t)'i'):
                rotateYCube(RIGHT_CUBE, 1);
                break;
            case ((uint16_t)'k'):
                rotateYCube(RIGHT_CUBE, -1);
                break;
            case ((uint16_t)'j'):
                rotateZCube(RIGHT_CUBE, 1);
                break;
            case ((uint16_t)'l'):
                rotateZCube(RIGHT_CUBE, -1);
                break;
            case ((uint16_t)'u'):
                scaleCube(RIGHT_CUBE, 1.01, 1.01, 1.01);
                break;
```

```
                case ((uint16_t)'o'):
                    scaleCube(RIGHT_CUBE, 0.99, 0.99, 0.99);
                    break;
                case ((uint16_t)'v'):
                    togglePin(LED_GPIO, LED_PIN);
                    break;
                case ((uint16_t)'n'):
                    togglePin(LED_GPIO, LED_PIN);
                    break;
            }
            calculateCubeVectorData(cubeVectorData);
        }
        // Add animated images to frame buffer
        buff_w->top=buff_w->anim_index;
          addLoadToBuffer(512, 512, 0b010, 0b100, 0b00); // recenter and
color yellow

addVectorsToBuffer(cubeVectorData[0],cubeVectorData[1],cubeZData,19);
          addLoadToBuffer(512, 512, 0b011, 0b011, 0b00); // recenter and
color orange

addVectorsToBuffer(&cubeVectorData[0][19],&cubeVectorData[1][19],cubeZData
,19);
        // if draw-er (TIM5_IRQHandler) has requested the next buffer,
         // the comput-er (this main loop right here) can now oblige that
request
        if (buffer_swap_req) {
            buffer_swap_req=0; // lower flag
            swapBuffers();
            // restart draw-er if it was stopped
            if (drawer_halted) {
                drawer_halted=0; // lower flag
                beginDrawing();
            }
        }
    }
    return 0;
}
```