3DPOV

3D Persistence of Vision Display

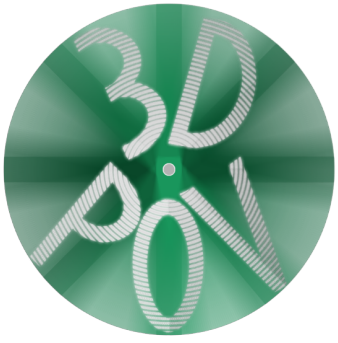
SYRACUSE UNIVERSITY ENGINEERING & COMPUTER SCIENCE

Philip D. Geramian
Computer Engineering

Alexander E. Julian
Electrical Engineering

Lyle J. Moffitt
Computer Engineering

Roman S. Olesh
Electrical Engineering



April 29, 2015

Abstract

The 3DPOV display is a three dimensional display that enhances 3D printing workflows by providing an organic representation of any 3D model. The system exploits the *persistence of vision effect* to form a perceived static image from layered motion (similar to how a spinning fan blade appears as a flat disc).

Layered segments of pulsing LEDs spin around a central shaft at 1800 rpm. The LEDs' illumination is synchronized with specific points of the blades' rotation to create the appearance of a 3D image.

Introduction

It is important to be able to visualize what you are working on, especially when 3D printing. In the standard 3D printing workflow, the designer creates a 3D model using CAD software. Using traditional display systems, the designer can only truly get a pseudo-three dimensional view of the object. With the 3DPOV display you can accurately represent any printable object before you proceed with the print. Being able to see what the object looks like in true three-dimensional space before it is printed saves time and materials (and consequently, money).

To get a "Persistence of Vision" (POV) effect, we built a system that rotates LEDs at 1800 rpm. During each rotation, LEDs will light up at programmed positions displaying an image. We used brushless DC motors to drive the system, and a copper ring and carbon graphite brush are used to transfer power to the rotating LEDs through the central shaft. Arduino-based microprocessors were built into each of the 8 layers to control the LEDs via ASIC LED drivers.

The concept of POV is that the human brain is capable of keeping an image in the mind for 1/16th of a second. This allows for society to enjoy things like film, and television where (in the NTSC standard) a new frame is generated every 1/30th of a second. POV makes the sequence of flashing images appear as a moving video. This effect is commonly seen with helicopter blades and fan blades, which are perceived as solid disks at high speed.

The Serial Peripheral Interface Bus, or SPI bus, is a communication protocol used to interface different systems together in a unified manner. A single master device can controll any number of slave devices. This allows for one to have a single master unit that may cost a lot of money, but is very good at processing data and have it interface with many, sometimes cheaper, slave units that are very good at doing a single task. In our system this is exemplified by the SRAM chips and the LED drivers that control our displays. In the SPI protocol every full duplex device has four wires going to it for just SPI purposes. These four wires are: Clock, Master out Slave in (MOSI), Master in Slave out (MISO), and Chip Select (/CS).

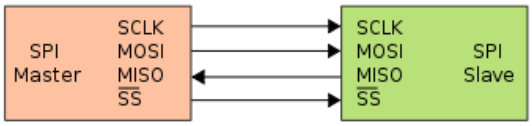


Figure 1 : SPI Basic Functionality

It should be noted that the standard has chip select as an active low line. The basic functionality is as follows: when the ~CS pin is pulled low by the master it then generates a clock signal and on every clock cycle it shifts out one byte of data at a time over MOSI (Master-Out Slave-In), while the slave device shifts

out one byte at a time on MISO (Master-In Slave-Out). Once the transfer is completed the clock stops and the chip select line is pulled high. Multiple devices can be similarly connected, each sharing the same MOSI, MISO, and SCK lines. However, each device must have its own CS (chip select). In this way, the master dictates who is receiving and who is sending.

Methods

The 3DPOV display is a very complex system containing many interacting subsystems. The 3DPOV display can be broken into four distinct categories: mechanical design, fabrication, and verification, hardware design and verification, software design and verification, and PCB design and fabrication.

Mechanical Design, Fabrication, & Verification

We did extensive design and fabrication with our mechanical system because it is one of the most integral parts of the system. In order to achieve persistence of vision, we need to maintain (approximately) 1800 rpms and the mechanical vibrations need to be minimized.

Our design uses a DC brushless motor in order to drive the system. We chose a Turnigy d2386-11 750 KV DC brushless motor to drive the system. The KV measurement for DC brushless motors stands for RPM/volt, where a lower number generally indicates the motor outputs more torque, but at the expense of speed. The 750 KV DC brushless motor was a good middle ground between torque and speed. Additionally, DC brushless motors are generally very stable because they have internal sensors that communicate back to a controller.

In order to control the DC brushless motor, we needed to use an electronic speed controller [ESC] with which we could control the speed with a simple servo signal. The interaction looks as follows:

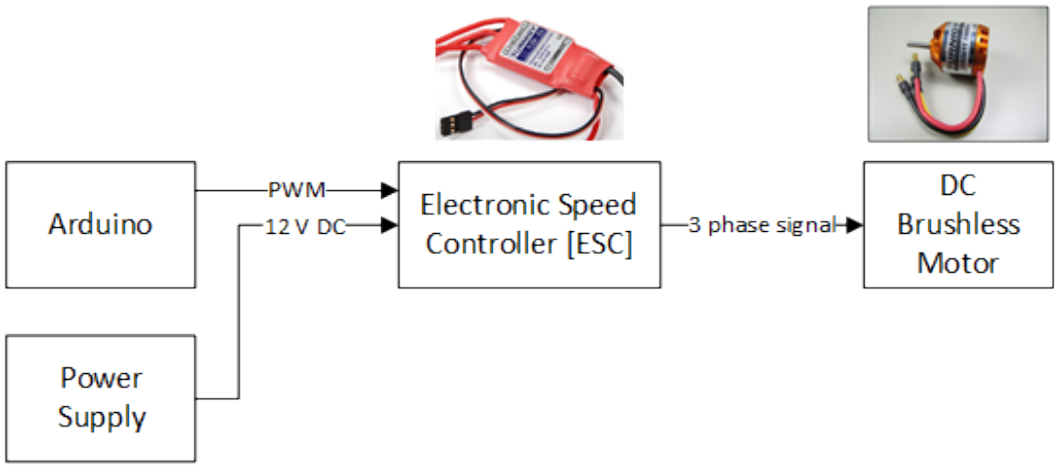


Figure 2 : Motor Control Diagram

We realized quickly that the brushless motor wouldn't be able to deliver enough torque. We also needed to consider how to build a very precise shaft that rotates completely on axis. To accomplish we decided on a timing belt pulley system that would drive a take apart DC motor. With the timing belt system we can use a 1:3 configuration that yields 3 times the torque at the output at the cost of 1/3 the speed. Therefore, the motor needs to spin at 5400 rpm so that the main shaft can spin at 1800 rpm.

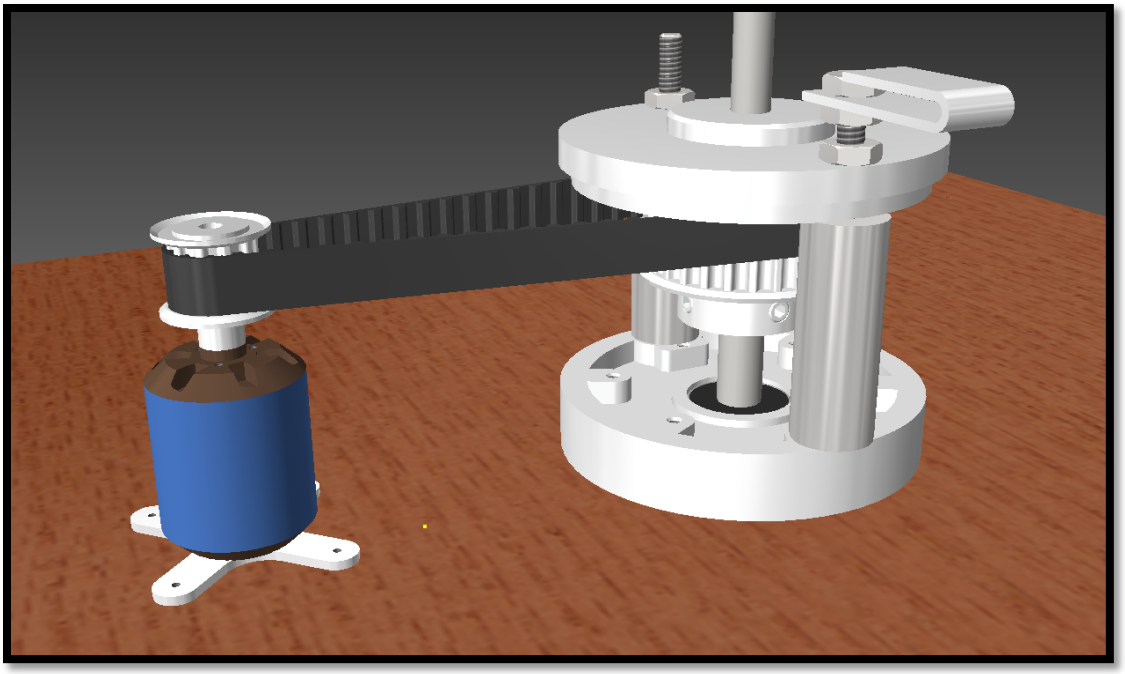


Figure 3 : 3D Model of drive system

Pictured above is a 3D model of our drive system. We fabricated the main shaft (pictured to the right) by taking apart a DC brushed motor. We took the DC brushed motor and removed the casing by unscrewing the two through bolts. Next, we carefully removed the windings with a hack saw and then removed the internal wirings and the carbon graphite brushes. We made sure to keep the brushes intact so that they could be reused.

The final step was to machine the metal frame and shaft so that it set at the appropriate dimensions we needed. This involved utilizing our machine shop to cut off the extra lip on the base portion of the motor casing and using a lathe to machine the shaft to spec. Then, a timing belt gear is attached to the shaft so that it can couple to the motor.

Since we removed the casing that supports the motor in place, we needed to make support pieces to brace the motor. We used the machine shop to lathe to cylindrical support pieces (visible in the diagram above) to fully support the frame.

With the drive system integrated, we needed a method of attaching the wings to the shaft. After some debate and designing, we decided that we could couple a carriage bolt to the lower shaft using a custom machined coupling and then affix the wings to the blade using nuts and rubber washers.

Pictured below is the original shaft (left), the shaft coupling, a bearing coupling, and the carriage bolt. The carriage bolt is a 5/16” threaded carriage bolt that can be found at any hardware store. In order to fabricate the couplings, we used a lathe to bore out the inner diameter of the lower shaft to approximately 3/8” and the inner diameter of the upper portion was lathed to approximately 7mm.

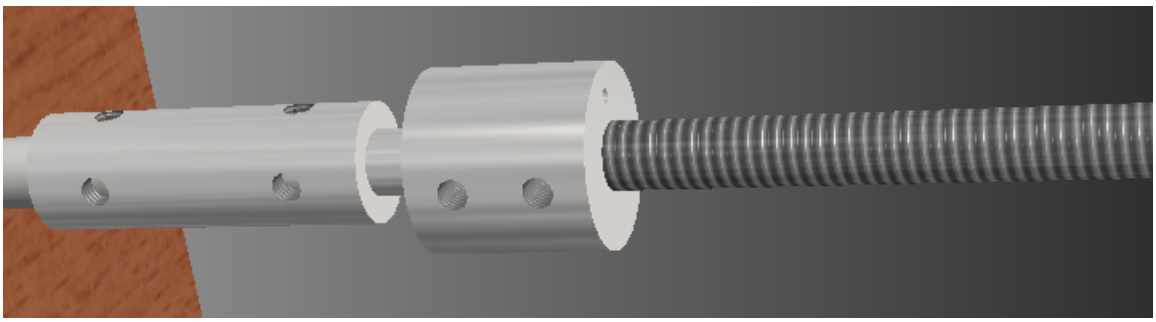


Figure 4 : 3D Model of coupling system

The upper coupling sits on the carriage bolt and sits inside of a bearing that is attached to the upper layer. This bearing acts as a method of pinching off any imperfections in the lower shaft that would cause the shaft to rotate off axis, creating bad instability. This bearing is crucial in reducing noise and ensuring smooth rotations.

The following is a visualization of the bearing attached to the upper layer:

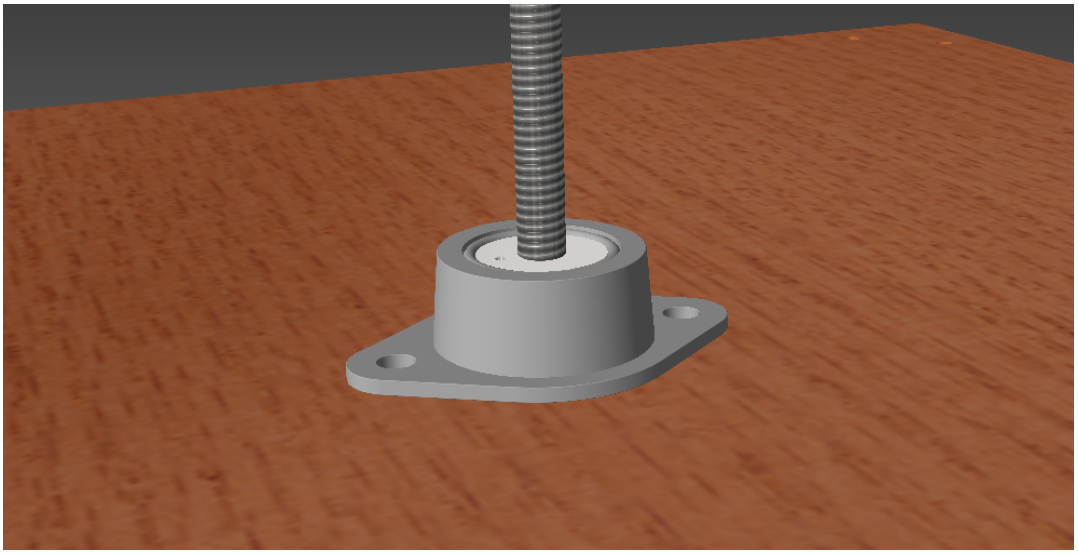


Figure 5 : 3D Model of bearing system

As can be seen, the coupling sits firmly in the three quarter inch bearing. We needed to build a separate coupling for the carriage bolt because we needed a clever way of getting the power up to the boards. We had brainstormed methods of doing so and a standard slip ring didn't seem to be a feasible solution both economically and for size constraint reasons. Therefore, we re-used the carbon graphite brushes to transfer power into a machined super conductive copper alloy ring. This ring would spin with the central shaft and a wire attached to the ring delivers power to the boards.

The copper ring system looks as follows:

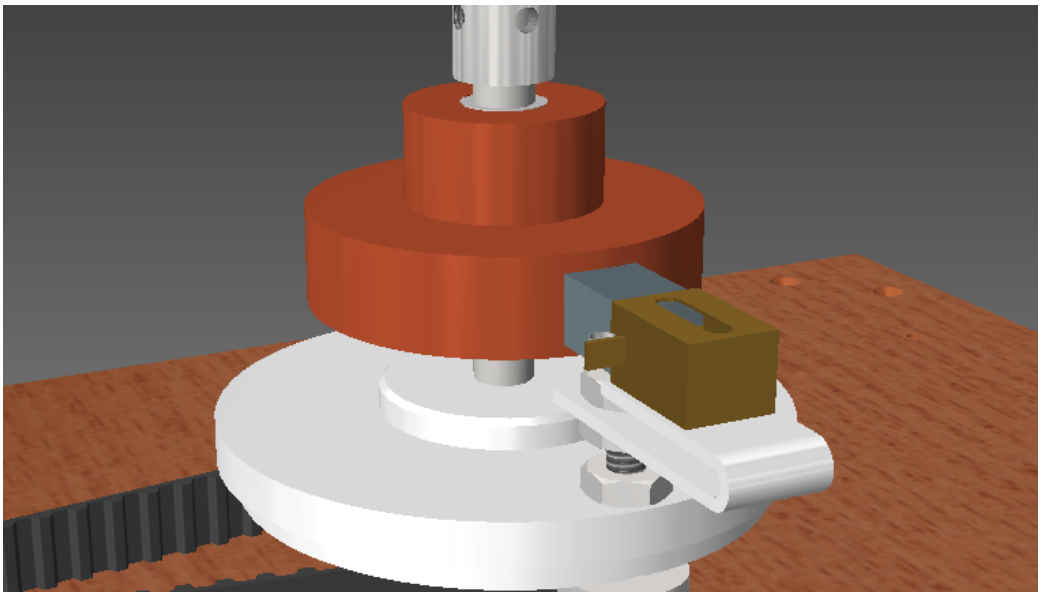


Figure 6 : 3D Model of copper ring power transfer system

The coupling that sits inside the bearing freely spins with the shaft and an 18 gauge wire can comfortably fit inside of a whole drilled into the coupling allowing connection to be made. It might be difficult to see in the diagram, but there is a plastic sheathing inside the copper ring to isolate it from the shaft. We did this so that we could send ground up through the aluminum/steel chassis and the boards could then simply pull ground off the carriage bolt.

In the next round of fabrications, we focused on building a fully functional wooden platform with a detachable safety peripheral. The wooden based was constructed with ¾" thick solid oak wood. We cut two squares for the two layers. We made the top layer slightly smaller than the bottom layer. The bottom layer was cut to 17 in^2 and the top layer was cut to approximately 16.625 in^2 using a table saw.

Holes were drilled to countersink standard sheet rock screws into the boards so that we could attach legs to both layers. Additionally, holes were drilled in the center to allow the DC motor assembly to sit flush and for the bearing and coupling to comfortably fit. 7/8" angle brackets were used to affix the legs of the top layer to the bottom layer.

With the base layers put together, we needed a method of ensuring that the system wouldn't slide all over the place when it is sitting on a plane table. To solve this problem, we took a sheet of gasket rubber from the hardware store and cut out squares that would fit on the bottom of the feet attached to the bottom board. This allows the board to get a really good grip on any table and stay firmly in place.

At this point, our base with the mechanical assembly looked as follows:

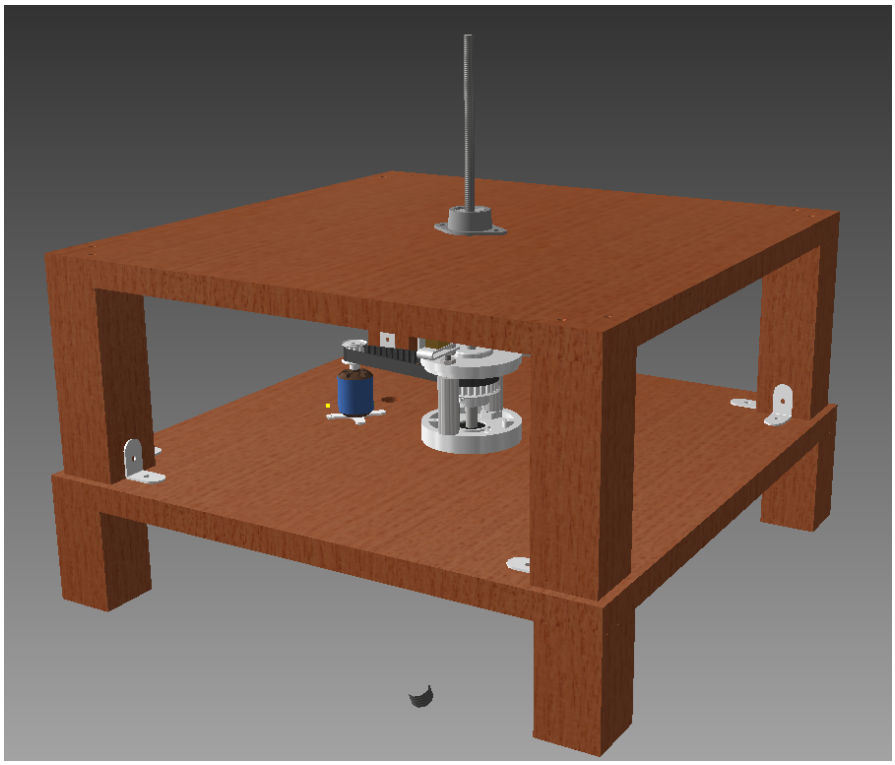


Figure 7 : 3D Model of base assembly with mechanical pieces

The next piece that needed to be fabricated was our safety shield. We ordered a large quantity of 1/8" thick clear acrylic. We cut it to size using a table saw with a special blade for cutting plastics. With the sides and top cut, we used a combination of super glue and silicon caulk to adjoin the pieces. Additionally, we cut the carriage bolt to size and fitted the electronics boards to the system.

Finally, we needed to fabricate a piece that would sit in front of the board that has all the control devices attached to it. We used left-over pieces from the build to make a nice looking control panel. The final result looked as follows:

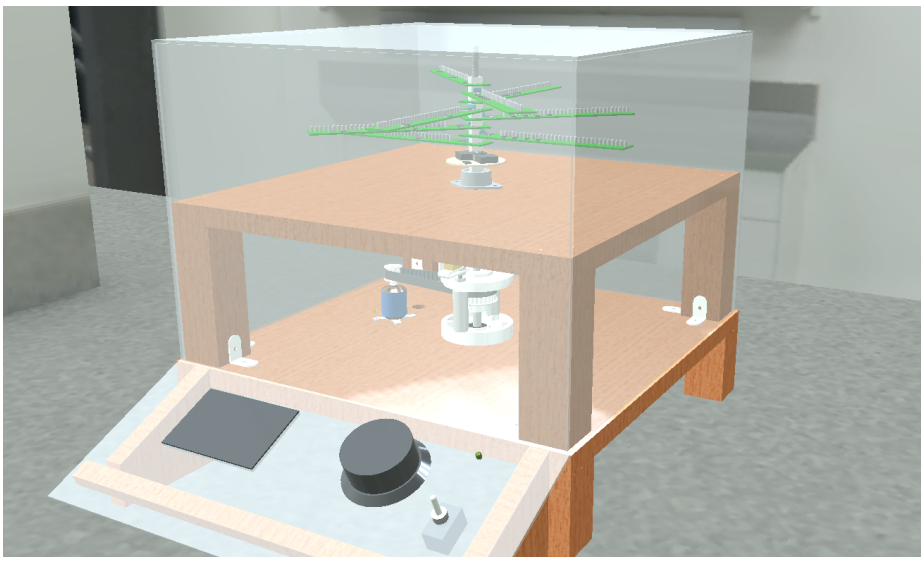


Figure 8 : 3D Model of complete assembly

Hardware Design and Verification

We started our hardware design with a set of criteria for the system. We quickly decided on logistics for the display such as how many layers (8), how many LEDs on a board (32), LED size (3 mm), and LED color (white).

With the logistics set out, we needed to determine the best solution for storing a bunch of data and creating a complete cohesive system. From our proof of concept, we knew we wanted to stick with the Maxim MAX6971 LED driver IC. This IC is a 16 output, 36V constant current LED driver that is interfaced to with SPI.

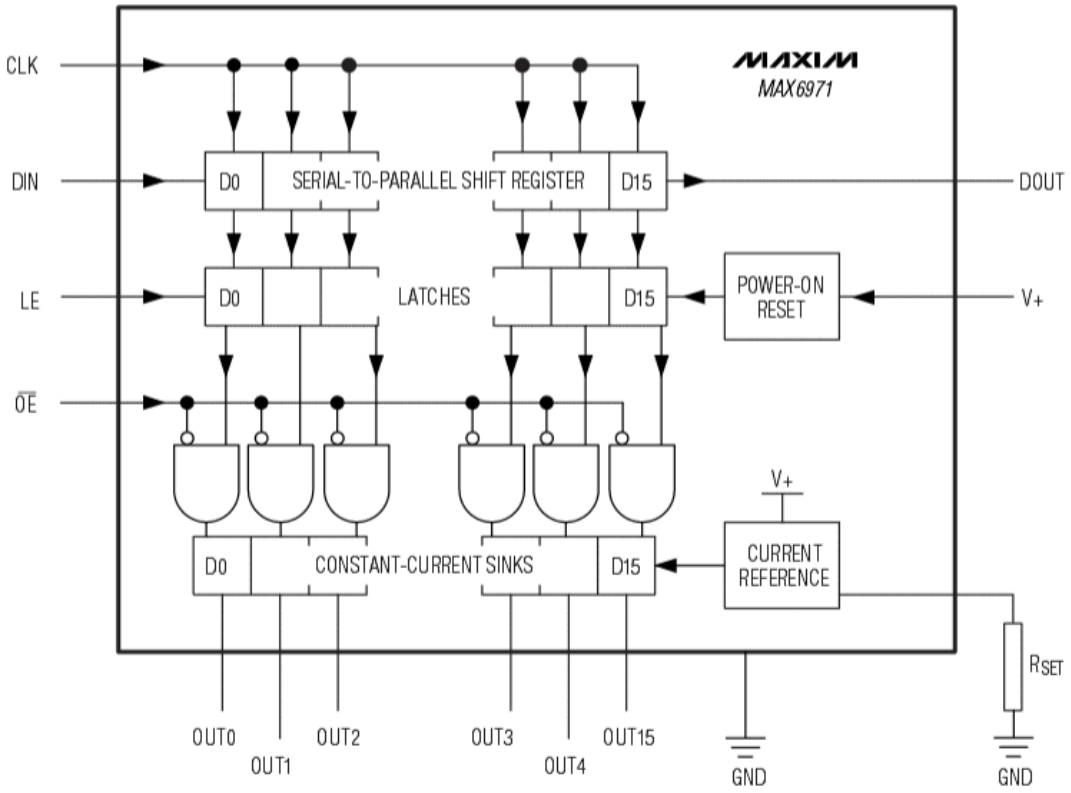


Figure 9 : MAX6971 block diagram

The MAX6971 has a simple 16-bit shift register on its input that will always shift the next bit through regardless of the state of the latches (OE and LE). This means that you can't feed back the data out to the master otherwise you'll have noise on the data bus. By sending the latch enable (LE) high, the contents of the shift register are dumped out in parallel to the output latches. When the output enable (OE) is set low, the output drivers go from high impedance to being on, display whatever is in the output latches. Data is shifted into the shift registers MSB first and a resistor is needed to set the current reference for the constant-current sinks.

We chose a 680Ω resistor which gave us the following current to each LED:

$$I(mA) = \frac{18000}{R_{set}} = \frac{18000}{680} \approx 26.5\ mA$$

In a worst case scenario, this means we would have to source $32 \times 8 \times 26.5mA = 6.784\ A$ of current. So we knew early on it would be important to have a power supply that could source enough current for us.

Before we addressed this issue, we sought to pick a suitable option for memory. We needed something we could talk to quickly with an 8-bit microcontroller and had an easy hardware interface. We had multiple options to test including: SD card (FAT file system, RAW binary data), EEPROM, and SRAM (volatile and non-volatile).

After doing testing on the SD card, we knew it wouldn't be possible to get it to work quickly enough with an 8-bit microcontroller. Additionally, SD cards require 3.3V to run and the rest of our system was using 5V, so this was a consideration as well. An interesting thing we found though, was that the SD card was quick enough for entire revolution, but not for each individual theta position.

The EEPROM memory IC we were looking at was a MicroChip 25LC1024 1 Mbit SPI interface EEPROM chip. The following is the block diagram:

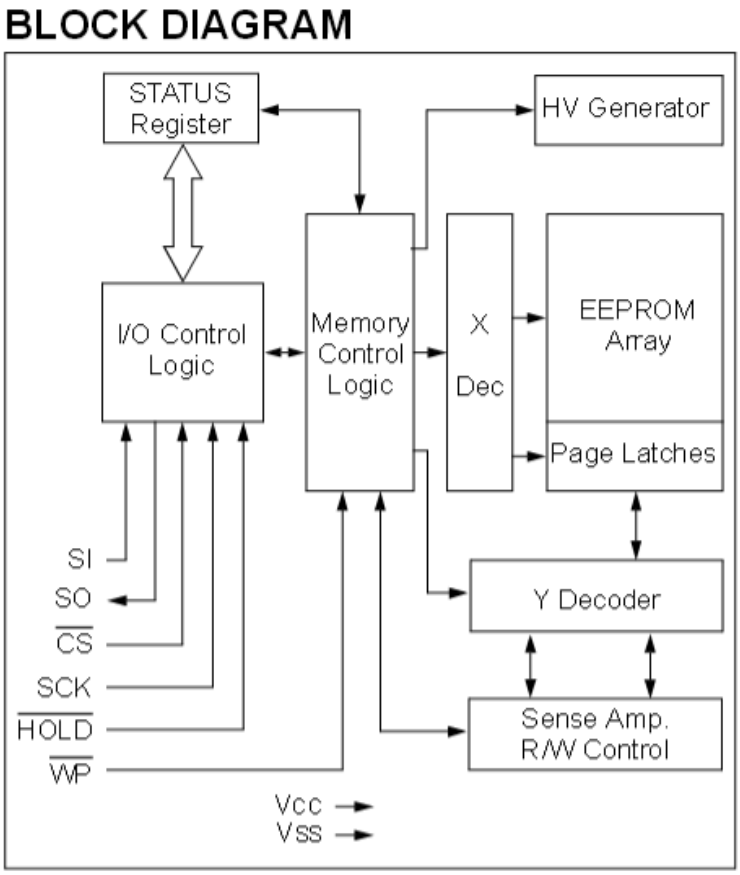


Figure 10 : 25LC1024 block diagram

In short, to use the EEPROM memory chip, the user sends over an 8-bit instruction via SPI. Then depending on the instruction, additional SPI transfers are needed specify memory addresses and to send/receive data from the system. The great part about the EEPROM chip is that it is very fast. With 8-bit SPI we are able to read 32 bits of data in approximately 16 µs. The one drawback to EEPROM is that it has to go into a write cycle at the end of each "page" of data (where a page is every 256 bytes). This write-cycle lasts several milliseconds which is not ideal.

Finally, the SRAM chips we tested were a MicroChip 23LC1024 1 Mbit SPI, SDI, and SQI SRAM chip. The block diagram for the SRAM is nearly identical to that of the EEPROM, however, since the SRAM is volatile we can get the same speed results, but we can write indefinitely without ever having to go into a read cycle.

To interface to SRAM chips, we needed the standard SPI connections (MISO, MOSI, and SCK), as well as an active low chip select line and an active low hold line. The chip select allows you to put multiple memory chips on the same bus and then simply select which chip you want to talk to. The hold signal allows you to suspend the SPI transmission. This is actually incredibly useful because it will ignore any signal on the input/output until hold is raised back to a high logic level. If chip select is raised, however, the device does exit the transmission. This feature allows us to continuously read/write to the device and then pause and write to the LED drivers and then continue to read/write to the memory chips without having to resend the instruction and address.

The final piece of hardware that needed to be tested was the Hall Effect sensor. In our proof of concept, we used a Melexis MLX92241 Hall Effect sensor that would detect the magnetic field and through hysteresis, change at the output. The issue with this device is that it was a current output device, when we needed to have a voltage output device.

After ordering a bunch of different sensors, we ended choosing the Melexis MLX92212LSE-ABA-000 unipolar switching digital output Hall-Effect sensor. The wiring diagram looks as follows:

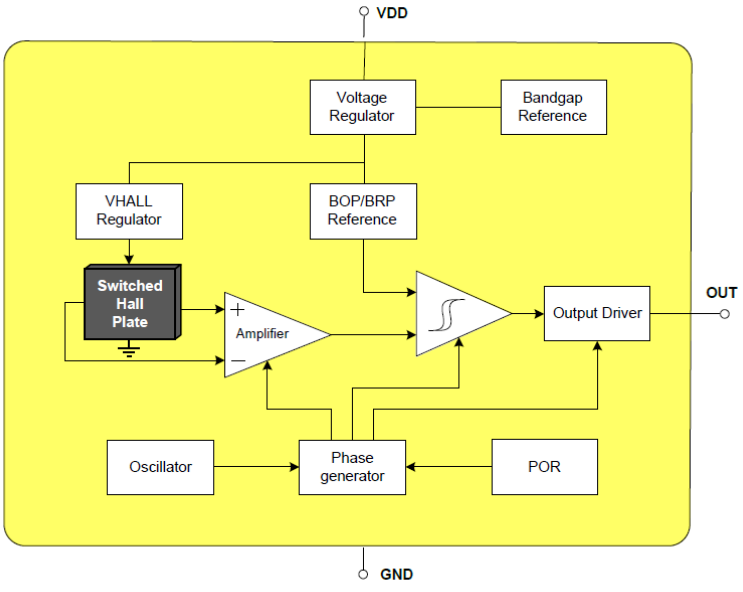


Figure 10 : Hall Effect sensor functional diagram

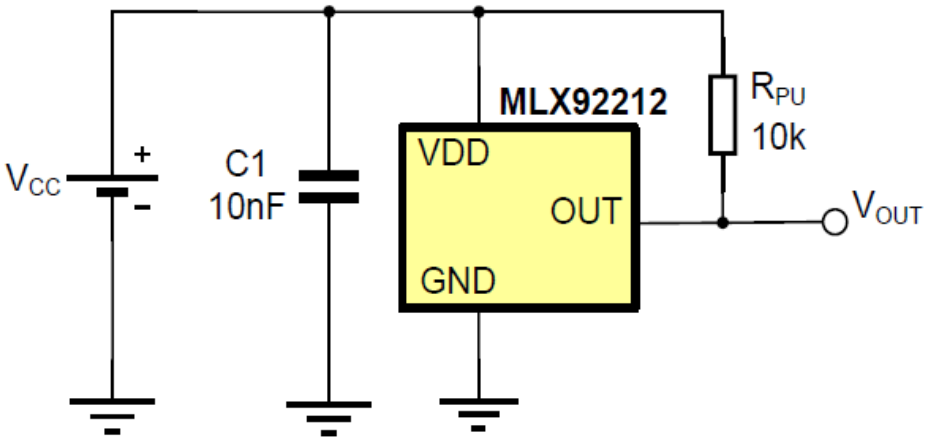


Figure 11 : Hall Effect Sensor wiring diagram

The MLX92212 only needs a decoupling capacitor across power and ground and a pull-up resistor off the output to work. Additionally, it can switch on and off at up to 10 kHz which is much quicker than our requirements call for (30 Hz). Since the sensor is a unipolar switching device, it only detects one pole (South Pole in this case), so the direction of the magnet matters. Upon seeing a magnet field the output drops from 5V to approximately 10 mV and stays there until the magnetic field goes away. Thus, we can look for a falling edge on our input and interrupt on that condition.

With all the pieces in place, we could put together a functional diagram of the entire system:

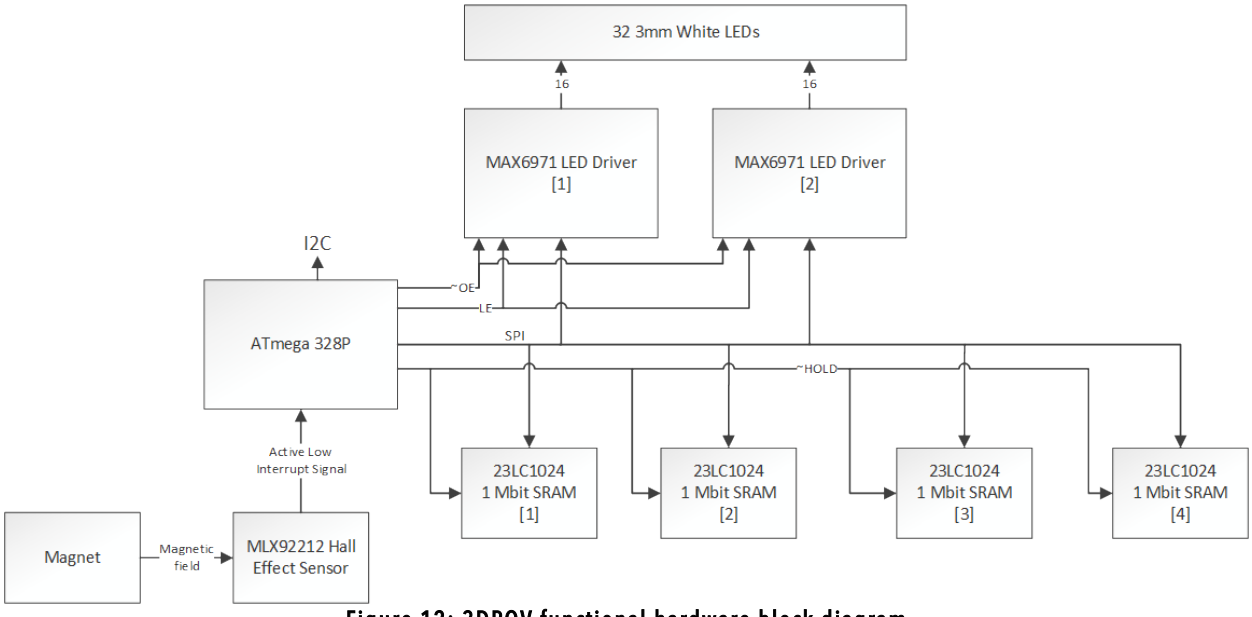


Figure 12: 3DPOV functional hardware block diagram

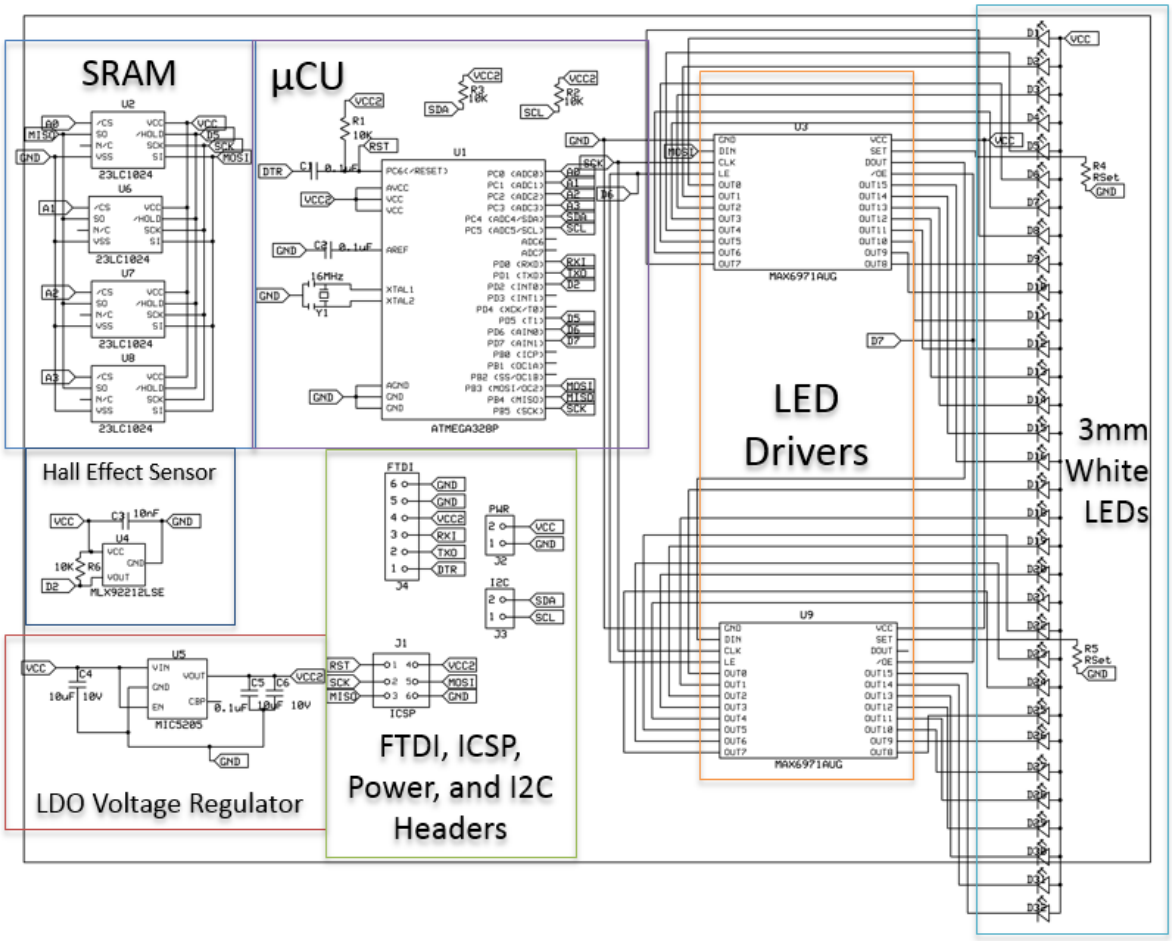


Figure 13: 3DPOV functional hardware schematic

The schematic is a synthesis of all of our components and has some new components that were added when we decided to have an Arduino built into our PCBs. The low drop off (LDO) voltage regulator is there to ensure that the ATmega 328p gets a solid 5V with an input of 5-12V. Additionally, the FTDI and ICSP headers were added in so we could program the Arduinos and burn the bootloader.

PCB Design and Fabrication

In the effort to make our end product look as professional as possible, we wanted to have our own custom PCBs printed. We used expressPCB to print our products because they have a very nice turnaround time and aren't too expensive.

Our first revision of the boards had a lot of errors, so we needed to go back through and fix everything so that we could have functional boards. The following is a picture of the revised board layout:

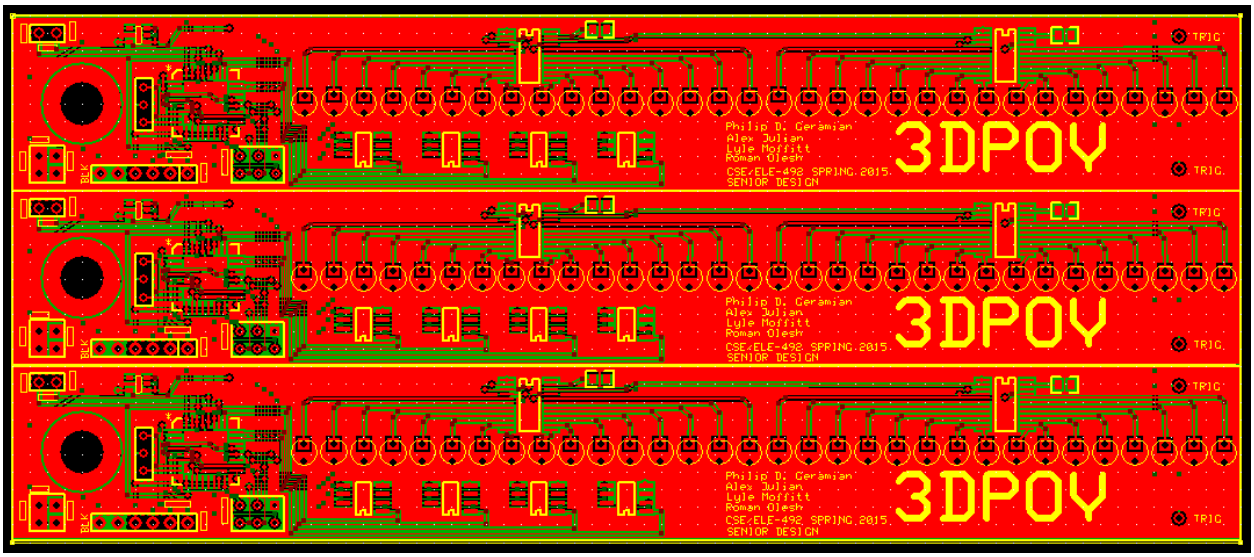


Figure 14: PCB layout as seen in expressPCB

With boards printed, we then had to populate each board with a combination of surface mount and through-hole components.

Software Design

The original design specification for our project called for an RF comm. chip and microSD card on every rotating blade. This original choice was made for reasons of simplicity; a parallel and asynchronous design mean that all of our components could operate in isolation. Having no communication dependencies gave us a very flexible, and adaptable design. From the very beginning, we knew that our implementation had to abide by a few simple limitations:

- Given that each image frame created by a blade's rotation would have 32 pixels along its radius and 256 radial positions in each rotation, we knew that each image-frame displayed by a layer would require 1 KiB of information.
- The microprocessor on each blade has a data storage capacity of 2 KB RAM and 32 KB ROM.
- We planned to spin the blade(s) at 30 Hz. This induced two further (complimentary requirements): one, we would need to satisfy a minimum data throughput of 240 Kib/s per layer (~2 Mib/s for 8 vertical layers); and two, that we would need to store 240 KiB of frame data for each second of our 3D animation (30 KiB per layer).
- Rotating at 30 Hz with 256 positions in each rotation meant that our system had a maximum theta-frame duration of 130 µs. Accordingly, this meant our LED drivers would need to operate at a refresh rate of 7.68 KHz.
- Using a 8 bit, 16 MHz microprocessor to drive the operations of each blade adds a further limitation that the data-path must have an end-to-end length of no more than 520 instructions. That is, the total computational overhead of all data management processes must not exceed a ratio of 520 instructions per byte transferred through it.

Software Verification

With an initial design in place and design criteria identified, we began the process of design verification with a series of prototypes. All the components of the design were broken into functional units based on minimal hardware dependency. We separated all the minimal sub systems in the data layer so that we could test each individually, whilst simultaneously testing the software driving each system. This 'Agile Development' approach allowed us to create the system piece by piece, making sure that each worked as it was built. Breaking the design into units meant that if any sub-system did not stand up to the design requirements, we could replace it with minimal redesign.

The first component system(s) tested was the RF communicator and the SD card reader. Both unit tests were spawned from readily available code, found online [1] [2]. We wired the devices up (one at a time) as directed and additionally connected a logic-analyzer to the ~CS, MISO, and MOSI (both are SPI devices). This setup allowed us to measure the performance metrics of each device, giving us a rough base-line[3] to compare against our design requirements.

The results measured were less than favorable, so we looked into possible optimizations. Referring to the technical documentation for the SD card[4] and the RF communicator[5], we found that there was indeed a lot of room for improvement. However, we also discovered that the RF's transmission protocol had a required delay that exceeded our max theta-duration. This was the first indicator that a significant redesign was necessary.

The biggest gain in the SD card's performance came from a major overhaul of its driver library. The original called operated the SD card as a FAT file system. That was stripped out, and we instead used the card as a raw block-device. After repeating the above test, we found that it was still too slow, so further software optimizations were implemented. To wit, the SPI and Digital I/O libraries were completely rewritten from scratch to take advantage of compile-time optimization techniques. Once this was completed, performance tests were run again. The SD access theta-delay was still much too long, but we found that the data throughput was still excellent. This knowledge, coupled with the first indicator spurred a major redesign of both hardware and software in our project.

Because of data storage requirements, we could not simply remove the SD card from each blade. Having an RF communicator on every blade was overly redundant, but it could still not be removed altogether in order to preserve a line of communication between the blades and the Beagle Bone front-end controller.

To resolve the former, we replaced the SD card on each blade with a set of SRAM blocks. These had a much simpler interface (and therefore shorter instruction-path length) , as well as a faster data latching technology (SD cards use EMMC). Both combined to give us data access times well under the theta-delay maximum.

To resolve the latter problem, the previously symmetric, isolated-parallel design was replaces with a concurrent, asynchronous master-slave model. The RF communicator and SD card were consolidated into one module (the master). A new I2C bus was added to connect it to the other blades (the slaves).

---

1 https://github.com/maniacbug/RF24

2 https://github.com/jbeynon/sdfatlib

3 See results: "SD Measurements" , and "RF Measurements"

4 http://www.circlemud.org/jelson/sdcard/SDCardStandardv1.9.pdf

5 http://www.mpja.com/download/RF24L01P.pdf

With the new design in place, we began another round of verification. Since we had compartmentalized the testing earlier, we did not need to verify the SD card and RF communicator again -- only the I2C bus and SRAM memory blocks needed to be tested. To maintain a maximum level of efficiency, the device libraries for the SRAM and I2C were written from the ground up using compile-time optimization techniques.

Software Integration

With all the component sub-systems verified, the next step was to integrate them all together into a coordinated, unified system. Normally, in a sequential-synchronous system, all the above components would need to be combined, ordered, and given intricate timing procedures in order to maintain stability. Furthermore, all of the sub components would be sliced up and interlaced, thereby voiding the assurances of all previous verifications. In order to avoid all of the above pitfalls, an asynchronous, declarative, event-driven model was used instead.

Events in this system take the form of hardware interrupts paired with stateful contexts. Microprocessor hardware triggers a fast context switch from the microprocessor's current program position into an Interrupt Service Routine (ISR) specific to the hardware context. The ISRs themselves are stateless actors who select and apply deterministic procedures based on software context. The procedures in turn operate on data context.

In order to set up our system, we create a list of general actions that need to be accomplished. These are our actors. Since our system is asynchronous, we need to make sure that none of these actions can result in a race condition. This behavior is usually implemented with the help of semaphores. In order to avoid the overhead of using semaphores to control context switching, we must make all our actions atomic through RAII. This is naturally taken care of for use for most of the interrupts, which cannot interrupt each other. Everywhere else, RAII is implemented using the side-affects of constructors and destructors, which effectively implement scope-based semaphores on the stack.

The actions in any dependency sequence (where later actions depend on earlier actions) can be categorized roughly into two type: contiguous and non-contiguous. If we define a sequence of dependent actions (A->B->C), the sequence is contiguous if action B can only be performed *once* in between actions A and C.

In an asynchronous system, we can have no way of knowing in what order those actions occur. Seemingly it would be impossible to implement an inherently serial process (like the data streaming necessary to implement our project) in without a guarantee of sequence. However, because the system is deterministic and atomic, we can think of each iteration of a given action as a tick on a progress meter.

For non-contiguous sequence operations, we use the meter like a queue in a consumer-producer kind of model. That is, we guarantee proper sequencing of such actions by simply placing requirements on the relative progress of each meter. For contiguous sequence operations, we can use the meter like a token (via function call-back or mutex) or we can directly modify the dependent action's meter.

Now that we have the above explained, the software design of the system backend can be properly explained. At right is a diagram of the basic structure. As mentioned before, all processes start with an *event*. The following columns could be roughly classified "first movers" and "core actions".

The former lists (in order of priority) each of the hardware interrupts that the system reacts to, except for the last (and lowest priority), which is composed entirely of control logic. They can collectively be labeled as "first movers" because they dictate when actions happen.

The "cores actions", obviously are what happens once the action starts. Together they make up each step of the data flow in every data path though the system.

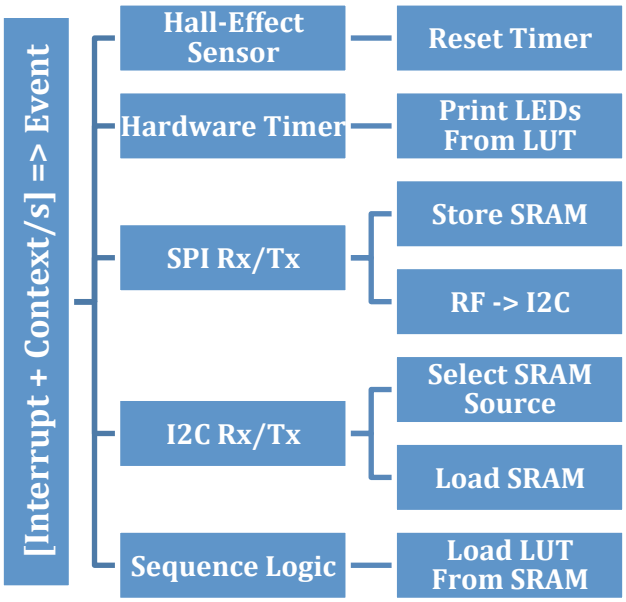


Figure 14 : Backend System Declarative Process Tree

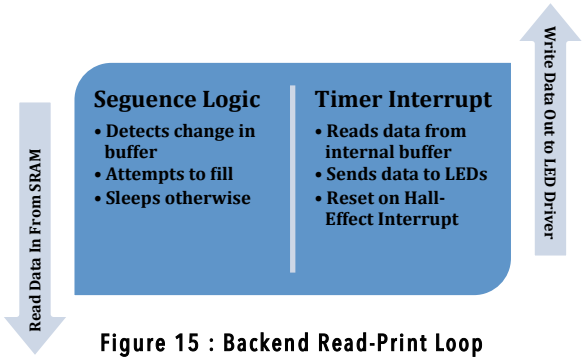


Figure 15 : Backend Read-Print Loop (Sequential Representation)

The data routing points in this system are varied in their nature, multifaceted in their use, and complex in their interaction. To wit, it takes both of the following diagrams to capture the actions as sequences. Conversely, all the actions in these two diagrams are encapsulated by the second column of Figure 14.

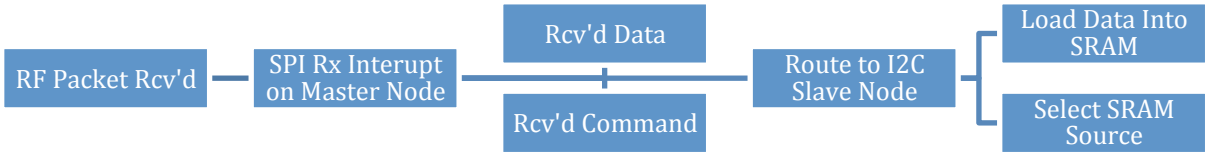


Figure 16 : Backend Data Flow (Sequential Representation)

Finally, that brings us to the last piece of the puzzle. As mentioned before, the processes are stateless; all they do is operate on (and in response to) contexts. The software context consists of the stateful data used to organize related actions. The data context consists of the messages that get passed in between nodes over SPI and I2C. While action is incomplete, packet is stored internally, along with progress state.

Software Interface

From the very beginning, the goal had been to display a 3D image. We planned for a typical use case to be a 3D-printing hobbyist, who wants to view his print object before committing to the print. Seems simple enough, but it in fact raises many questions that we had to answer, chief amongst them:

How do we turn a 3D-print file into a data format that works for our display?

[and]

How do we get the data loaded onto the backend?

We approached solving the first question from 2 perspectives. First, we looked for popular, open-source, libraries related to importing and manipulating 3D objects. Second, we looked for public repositories that hosted 3D object files and looked for common formats that coincided with the supported formats of the first approach. As we continued to search, other selection criteria came into play. For example:

- From the very beginning, we knew that performing Cartesian to Polar transforms would be key to the data conversion process. Therefore, we needed tools that would facilitate that.
- We should preference smaller, ready to go tools over custom implementation.
- Converting a 3D object to 2D slices requires a special tool that implements the route-planning algorithms necessary.

As the search drew out, many pieces of the tool chain began to solidify.

- It was decided that that STL was the best file format because of it's ubiquity and open-source nature.
- Instead writing any library elements for interpreting 3D objects and slicing them, it was decided that the combination of the tools Sli3r and Ghostscript would serve well enough for our purposes.

These tools were selected because of their robust and multifaceted nature. Slic3r has tons of options for scaling and rotating 3D images as part of its preprocessing routine. This allows us to pre-render an animation as a series of linear transforms. It's ability to output an SVG image is especially a bonus, because it allows us to plug that straight into Ghostscript.

GS was a great fit for our tool chain for a number of reasons. Aside from its many, many options for image conversion, it has specialized algorithms built in for up-scaling and down-sampling input images to optimally convert to a given resolution and pixel density. This was a key feature because it implemented a key step in our conversion process.

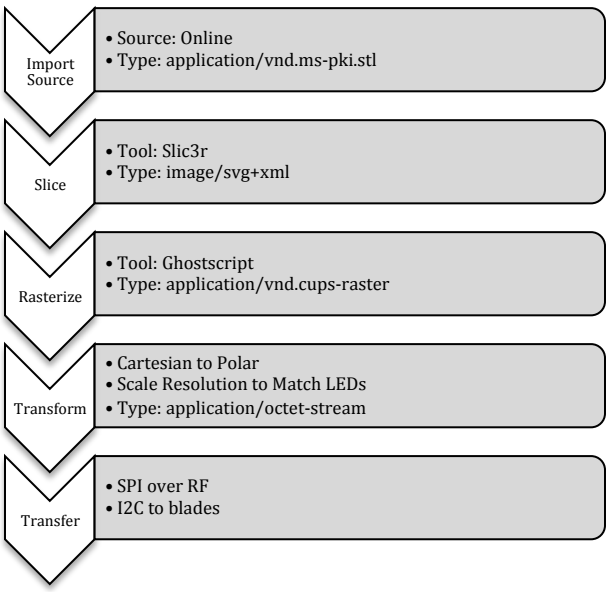


Figure 17 : Image Conversion Tool Path

Ghostscript was also a great fit because it outputs raster files. A raster file is pretty much the perfect source file for our project because it's literally a Cartesian plot of pixels. From there, all that is needed is to perform our transform and fit the pixels to the LEDs physical positions. With that map determined, we simply serialize the plot as a literal bit-per-LED representation of all the positions in a blades rotation.

The last phase of the tool chain is to transmit that pre-computed buffer to the master node on the backend. This requires a RF communications program fro the BeagleBone's side of the transfer. This last part perfectly solves the second problem addressed earlier.

Results

Mechanical Design, Fabrication, & Verification

Throughout our mechanical build process, we took measures to ensure our cuts, drills, etc.… were made to specification. The frequent use of dial calipers and other measurement tools allowed us to maintain high precision throughout. Additionally, the machine shop can machine down to a .001" precision.

Our end product, the synthesis of our mechanical build process, was a refined, tuned machine. The rubber grips allowed the display to not slide on the display table. Additionally, we used stage weights as a back-up measure to ensure the folding table didn't crash.

Our meticulous balancing of the blades proved to be successful and we were able to rotate at a consist 1800 RPM. Speed measurements were verified throughout with a tachometer. Additionally, the clear acrylic safety shield added two layers of refinement. For one, it gave people piece of mind that our project wouldn't kill them and additionally, it worked excellently as a sound buffer.

The most important aspect of our design was the upper bearing that we installed. It kept the shaft from coming out of balance and worked exactly as expected. Even though we still had vibrations, the counter-clockwise rotation ensured that the nuts only ever tightened.

Hardware Design and Verification

For our hardware synthesis, all systems worked very well together. Our carefully designed system was able to intricately inter-communicate and display crisp clean images.

Our concerns with power draw proved not to be a huge issue, as the 18-gauge wire was able to supply enough current up to the blades without melting or breaking. One thing we did observe is that bearing that start to cut away at the rubber sheathing on the wire.

The power supply we used was able to run the control panel, the motor/speed controller, and all the boards and the LEDs comfortably. Using a 270W PC power supply proved to be very convenient because it gave us high power supplies of all the common voltages we could ever need (3.3V, 5V, 12V).

The 3mm LEDs gave us a great fine resolution display and were very bright, even in the harsh lighting of the atrium. Additionally, the ATmega 328p microcontrollers did a great job of controlling all the onboard systems.

PCB Design and Fabrication

Our PCBs that were printed through ExpressPCB looked excellent and very professional. Outside of 3 minor issues (that were easily fixable), production went off without a hitch for the second revision.

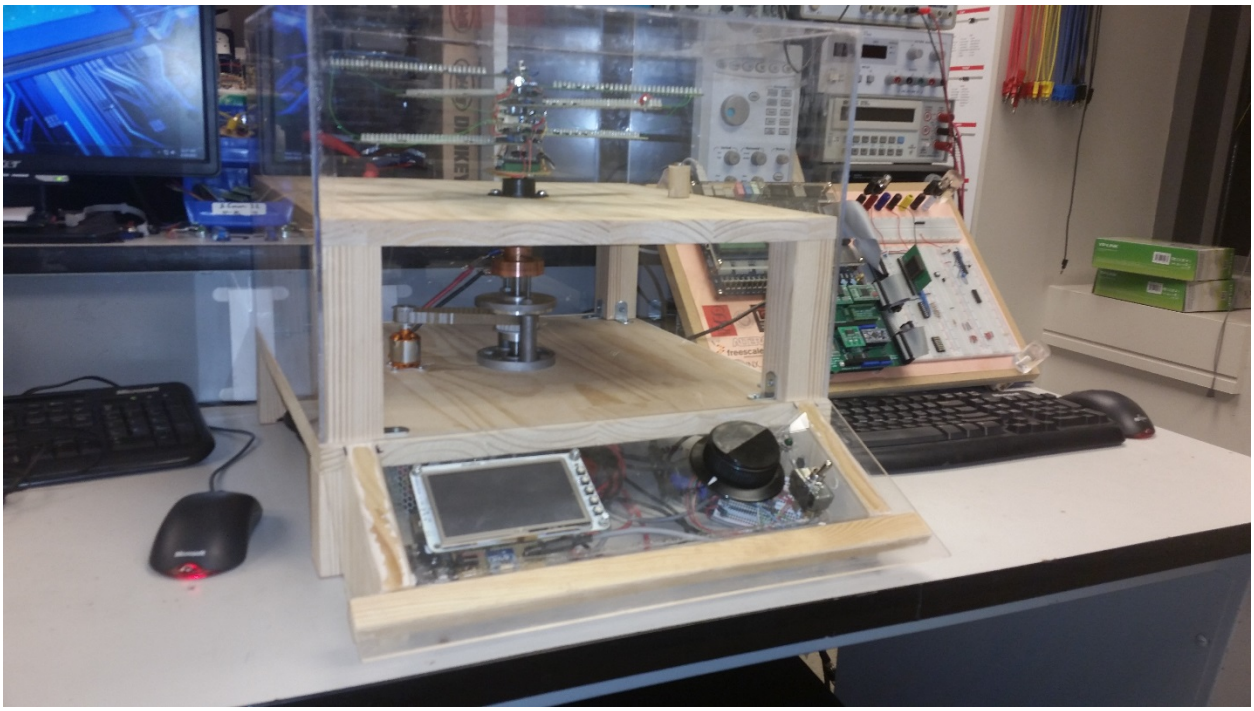


Figure 18 : Front view of the completed project

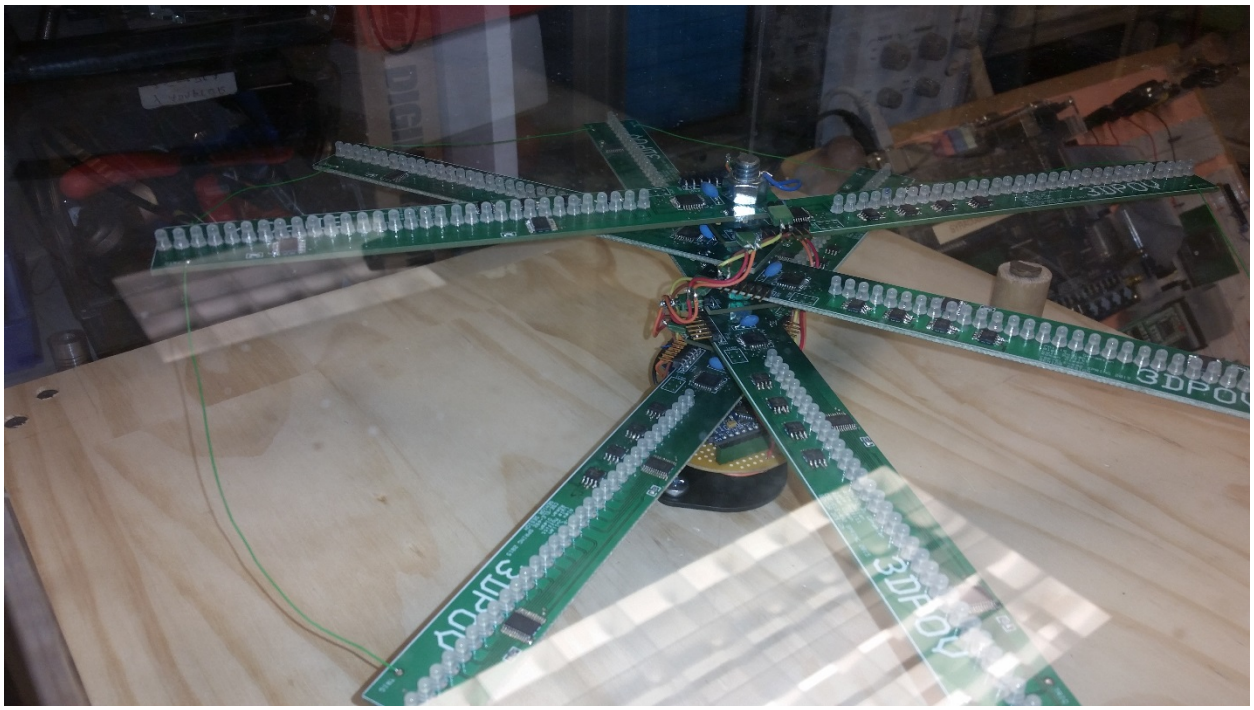


Figure 19 : Close up of the LED blades, fully assembled and connected.

Software Design

No results to document.

Software Verification

The following is a table summarizing our timing measurements for various devices.

Device	Test Size (Bytes)	Read Time (µs)	Write Time (µs)	Avg. Throughput (Bps)	(kBps)	Notes
EEPROM	256	522.87	450.89	528,685.71	528.69	*Using Arduino's digitalWrite() function for pin control.*
EEPROM	32	68.35	64.8	481,002.83	481.00	*Using direct manipulation of MMIO*
LED Driver (Theory)	4	10	10	400,000.00	400.00	
Soft SPI (Theory)	32	128	128	250,000.00	250.00	*Software defined SPI operating at 2MHz*
SD Card	1024	7040	9080	129,114.94	129.11	*Block device with custom SD library*
SD Card	512	17300	3160	95,810.35	95.81	*Block device with custom SD library*
I2C (Theory)	33	700	700	47,142.86	47.14	*Based off of datasheet timings, operating at theoretical max of 400 kHz*
RF	32	700	700	45,714.29	45.71	*Includes a 260µs PLL lag*
SD Card	512	72848	72848	7,028.33	7.03	*SD FAT using sdfatlib*

Software Integration

It did not make it through testing in time for Open House. The Arduino libraries have too many (undocumented) dependencies that conflict with interrupts. During debugging, it would run fine and then freeze indefinitely.

Software Interface

This tool chain never made it to the Open House demo. There was not enough time to complete it.

Discussion

Mechanical Design, Fabrication, & Verification

Over the course of building the 3DPOV display, we ran into a vast variety of issues and were exposed to issues we had previously never thought of.

Throughout the build process, getting a precise mechanical build was a huge struggle. We found that any small errors in our build process could cause error that would be too significant to compensate for. We knew from the beginning that we wanted to use a dual-helix formation because the pattern balances well (each opposing blade is 180 degree offset and the boards are all relatively identical). We found that even small errors in the alignment of the blades could cause issues.

To address this, we took a piece of foam board and cut a 45, 45, 90 triangle that we could use to ensure perfect alignment. In a fit of Murphy's Law, we discovered that it is detrimental to rotate the system clockwise. What ends up happening when you spin clockwise is that any vibrations in the shaft will cause the nuts holding the blades in place to loosen. In a test run, the nuts loosened to the point that all blades came loose and snapped into place. Luckily, we had our safety shield at the time and additionally, the motor has a built in safety shut-off that turns off the motor when too much torque is applied to it.

Since the speed controller sends out a 3-phase signal to the motor, we discovered that just swapping two of the wires would change the direction that the motor travels.

Another place of great tribulations was the shaft coupling. Mr. Timothy Breen from the workshop helped us machine probably 4 or 5 shaft couplings before we got it right because we found that we needed much more precision in our coupling than initially planned. However, even with our final coupling we were able to measure .006" of play in the lower shaft. What must've happened is that the lower shaft slowly bent out of shape as we continued to test the system. The issue with having this amount of play in the shaft is that when we coupled the 8-inch carriage bolt to the lower shaft, that 0.006 inches of play translated into over .060 inches of play. This caused the system to be extremely unstable.

We were able to address this, however, by incorporating the bearing on the upper layer. We initially tested this by drilling a whole in a rectangular piece of plastic and fitting it to the carriage bolt. We then spun the shaft while holding the stock piece firmly to the top board. We observed that our play in the shaft disappeared. We then knew we needed to get a bearing to place on the top layer.

We discovered after planning the bearing system out that we would need to machine a coupling to sit on the carriage bolt that would have room to feed up the power cable without causing any interference. Upon installing the bearing and the coupling, the play wasn't completely eliminated. The initial bearing coupling was made out of plastic and we had concerns that the bearing would melt the plastic and cause a bad situation to happen, so we meticulously crafted an aluminum coupling to work for us.

Even with all these improvements, the system still needed fine-tuning. We found that we could use small nuts to counterbalance the system properly. The clear acrylic safety shield also acted as a nice sound buffer. All of the mechanical precision work that was done gave us a very good appreciation for mechanical systems we never thought of. For instance, the huge hard drivers you would find in banks were precision instruments that had a similar rotational system, but those had no more than a couple of micrometers of play.

Hardware Design and Verification

From a hardware perspective, we also ran into various issues. In our shift from the old Melexis MLX92241 Hall Effect sensor that was a current output device, we needed to find a device that would give us a digital voltage output that would eliminate the external Schmitt trigger.

Initially, we had chosen the Toshiba TCS20DPR Hall Effect sensor. Upon doing testing, we discovered it had one singular issue that would cripple our system from working. An inspection of the datasheet revealed that the sensor operated at 20 Hz, a rate much slower than our rotational speed of 30 Hz. We then had to go back to the drawing board and find a new sensor.

We ordered 5 or 6 different Hall-Effect sensors and the one we chose out of the bunch was the Melexis MLX92212LSE-ABA-000 unipolar switching digital output Hall-Effect sensor. This sensor has a switching frequency of 10 kHz, so it was more than capable of handling our work capacity.

We also spent a good deal of time choosing the right Memory device. As we described earlier, we had to do extensive testing and benchmarking to determine which device would suit our needs the best. The benchmarking on the SD card always proved to be mixed. On one hand, we could get an SD card with enormous amounts of storage, however, the 8-bit Arduino proved to be too weak to take advantage of the speed available with SD cards (additionally, we didn't have the funds to license the SD card specification for our project).

The EEPROM memory chips were very promising initially, but they had the write cycle issue that was a bit of a bummer. However, if we never had to write to them after an initial start-up, they would be an ideal solution because they can read indefinitely.

The SRAM chips were also very promising. They offered the same performance as the SRAM, but without the write cycle dependencies. As a down side, the SRAM chips are volatile memory, meaning that any loss in power erases the data on the system almost immediately. This trade-off is manageable because the ATmega chips can be programmed while the board is powered.

Our decision on the SRAM chips came down to read and write performance. The lack of having a write cycle was critical for us in implementing our ideas.

Another issue we ran into, one that we don't fully understand, was again related to the Hall Effect sensor. Whenever we ran the system before moving to the 8 layer 3D system, we didn't have to worry about any form of interference on the interrupt line. However we found that once we connected all the layers together, the boards would start randomly triggering at spots. The distance between the layers wasn't great enough to have to consider a distributed model (nor was the signal at a high enough frequency).

One theory we had to the random triggering had to do with the Hall-Effect sensor picking up spurious electromagnetic emissions. Since we are sending a large amount of current up the wire and said wires are spinning very quickly, we know from Maxwell's equations that a time-varying or space-varying current will induce Magnetic or Electric fields. We think the sensor was possibly sensitive enough to pick this up.

We solved our issue by doing an extra check in software when our interrupt is called. Basically, the first thing the interrupt service routine does is it checks if the signal is still low. If this conditional is false, we exit from the ISR and move on.

PCB Design and Fabrication

We also ran into issues in our PCB layout phase. Our initial board layout was far too aggressive and our spot-checking proved to be very poor. Our boards were so bad that we decided it would be easier to just redo them. The second revisions only had two minor errors in them, so we were able to use them.

We also ran into a minor road bump in trying to cut the PCBs. ExpressPCB gave us 4 copies of our board, each of which had 3 copies of the individual boards on them. We first tried to use the guillotine cutter in lab to chop up the boards, but it proved futile to attempt to use it. Dr. Brandon Choi was gracious enough to let us use his PCB cutter to slice up our boards.

Another interesting error we had in the boards was that in order to burn the bootloader, we needed to supply power to the boards through the external power system even though the sparkfun AVR programmer has a built in power supply system. We were very concerned that the reason we couldn't program the boards was because the SPI lines were being loaded by the LED drivers and the memory chips. Fortunately, this wasn't the case.

A final hardware error we had related to the SRAMS. It turned out that there was no guarantee that the memory chips would start in SPI mode and in order to send the reset command you had to do so in SDI or SQI. We had to use software SPI to solve this issue and send the reset command if necessary.

Software Design

Starting with a massively underpowered microprocessor was a mistake from the beginning. Having reduced computing power and speed under such tight timing constraints necessitated the creation of optimized libraries. If we had started with a more powerful processor, we could have spent that time more effectively.

Software Verification

The original design was bloated and unnecessarily redundant. You could look at the parallelized simplicity as a feature, but you can also look at it as a crutch. If we hadn't verified that all the components worked early in project, it would have meant disaster later on. Moreover, unit testing each component so early in the development process, allowed for us to have a tight feedback loop during the design process. A direct byproduct of this is that we got to iterate through and improve our design multiple times. As a result, our final product was refined as good as it could be.

Software Integration

Sometimes last minute bugs can hold everything back. Part of the beauty of compile-time programming techniques is that it helps move problems from run-time to compile-time. Unfortunately, this cannot be done for all problems that one might encounter, which leaves open the possibility for exactly this kind of situation. It just serves to drive home the lesson that when your production team is halved, the project goals must be adjusted accordingly.

Software Interface

This part of the project failed to come together because fewer team members were available to work on the project as a whole than had originally be promised. Part of the beauty in the software design of this project is that it was massively scalable. Depending on time and resources, this part of the project could have been anywhere between fully manipulating 3D objects in memory, down to just a simple batch process executed via script. If more people had been working on this project, this could have easily been done.