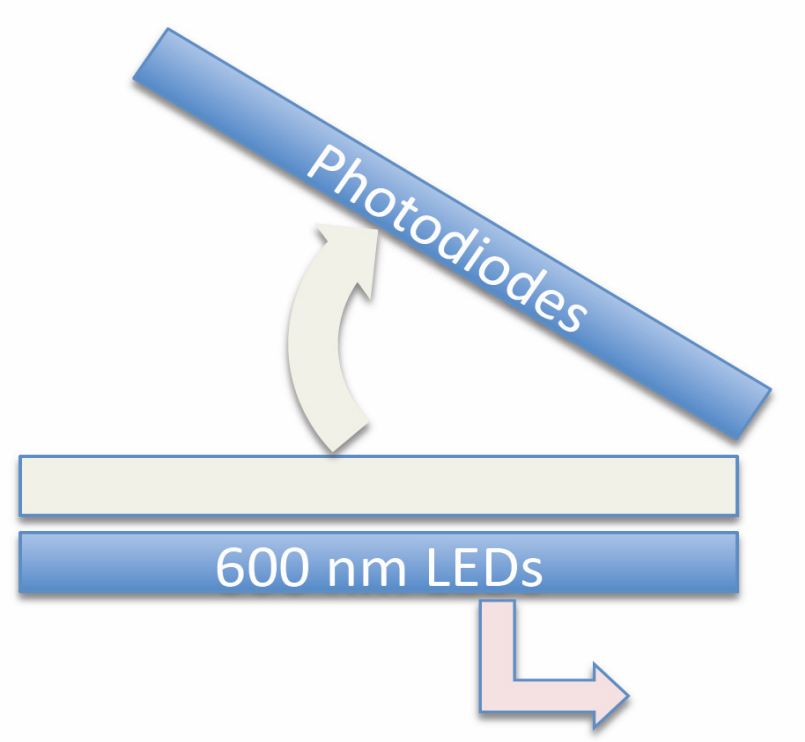
Turbidostat Final Report

MS 5



By:
Peter Harker
Maxwell Holloway
Evan Dreveskracht

Executive Summary

This project focuses around the synthetic biology idea of evolution. Ongoing research at the University of Washington is focusing on the idea of evolution over many generations to produce better synthetic enzymes. Using a turbidostat to keep a constant volume and turbidity by removing bacteria or adding nutrients over time, we can observe mutant strains, which eventually take over and grow at a rapid rate. Once this happens, other nutrients may be added to direct the evolution further. These mutant strains are the key to new research in this department, and a mini – plate reader turbidostat will help to propel this research further by streamlining these operations.

Our goal was to produce a fully automated turbidostat by producing a mini-plate reader of optical density, and interface that system with a liquid handler to exhibit desired control behavior. Although the current turbidostat performs basically the same functions, a system of light emitting diodes in a plate reader controlled by the liquid handler has three main benefits over the former system. First, the former laser tool to measure the density of particles (turbidity) tends to diffuse in the sample being measured. By using LEDs we hope to reduce or even prevent this measurement error. Second, by using a well plate of 24 instead of a single well, we can sample 23 more enzymes, increasing the chance to find the mutant strains. Finally, by using the already available liquid handler to keep a constant population size, we will reduce the laborious task of adding nutrients and removing bacteria by hand over a period of hours or days, while retaining the same data. In addition, the liquid handler could be used to transfer all 24 wells to a new plate, thus solving the reduction in measurement accuracy caused by bacterial growth on the well walls.

With help from our three other advisors and various faculties, we were able to produce a mini plate reader, which reads optical density of 24 wells in a plate and analyzes that data with our control code to give the liquid handler a volume of nutrient to add. This plate reader can also store all data for further analysis assuming a mutant strain does appear. Our budget for this project was $1000.00, and our total ending price came to be $613.58, well within our budget. Originally this project meant to combine the plate reader with the liquid handler for a fully functioning system, unfortunately due to time constraints and our ability to manufacture and assemble parts needed for testing, this was not completed. However, since the control code works in simulation and the plate reader is outputting reasonable data, all that is needed is communication and calibration with the liquid handler to perform the required functions.

We would recommend that the synchronization of the plate reader and liquid handler continue in order to complete the system and solve any unforeseen problems with data acquisition or errors.

Contents

Project Description – Customer Needs and Plan of Work

Overview

This project is directly related to current research being conducted at the University of Washington involving synthetic biology called the Directed Evolution Project. The overall goal is to help build synthetic enzymes with increased performance by using directed evolution in a turbidostat. A turbidostat is used to keep a constant turbidity, or culture density. When the density of the culture increases, the feed rate is increased, and some bacteria are removed to dilute the culture back to a set population size. When the density begins to fall, the feed rate is slowed so that growth can restore the density of the culture to its set population size.

By keeping this set population size, evolution can produce a mutant strain that capitalizes on the nutrient rich environment and adapts to take advantage by taking over the population in that sample. Once this mutant strain is found, new nutrients can be used to enable even more efficient evolution. This fine-tuning of enzyme evolution could be used in many applications for research in synthetic biology, and a mini – plate reader turbidostat is useful for its speed and efficiency in performing this research.

The mini – plate reader has three advantages over the current turbidostatic method. The first is the replacement of the light source from a laser to an LED source. According to our advisors and our customer, the ability of the current turbidostat to measure optical density is difficult due to diffusion of light from the laser source. By using LEDs instead, we hope to reduce this effect significantly. The second advantage the new system has is the number of simultaneous wells being measured. By increasing the sample size, we can increase the chance of a mutant strain being found, and take advantage of the automation. Finally, by performing autonomously the turbidostat can run with little intervention for long periods of time, reducing time spent in the lab transferring liquid and trays by hand.

Customer Needs

Our customer for this project was our EE449 professor, Eric Klavins, who is also the faculty advisor for the Directed Evolution research project at the University of Washington, using the aforementioned method of enzyme evolution. Our customer would like a turbidostat that reads in 24 samples of culture at a time, and performs all necessary functions of a turbidostat for each individual well. This includes measuring light absorbed by the bacteria, (for our function we know that bacteria absorbs light at a frequency of 600 nanometers) have the scanner program output controlled behavior, and read that output with a liquid handler to exchange fluids. This system must also perform semi – autonomously over a period of a few hours to possibly days, with data being checked periodically for anomalous growth patterns in the 24 wells of bacteria.

The budget for this project has been given as $1,000.00, which is quite reasonable given our original plan of work.

Plan of Work

First Schedule

Our original plan had changed dramatically due to numerous complications, which we assumed could happen given the unknown elements in designing each in our system, and learning each as we went along. These elements are the housing unit, circuit boards, and communication between them and the liquid handler. For the purposes of this project, we will give a brief overview of our original plan, and the reasons we changed or kept each element. The following figure gives our original plan, which was sent to our client as a tentative schedule.

Week 2	-Speak with advisors and client, order sample LEDs, photodiodes, microcontroller
Week 3	-Build test equipment to determine efficient LED wavelength. -Test functionality of microcontroller / liquid handler compatibility -Model System
Week 4	-Use test equipment to sample photodiode / LED optical density readings using blue food coloring in plate well
Week 5	-Order Printed circuit board for Photodiodes -Build / order designed housing container using Solidworks
Week 6	-Build housing container with PCB and test measurement capabilities. -Order another PCB if necessary
Week 7	-↓Continue testing equipment ↓
Week 8	-Access efficiencies with feedback control and design requirements
Week 9	-Possible upgrades (servo motor opening of housing, heat influence on LED output, etc.) -Final Report
Week 10	-Final Report (cont.) -Lab Visits
Week 11	-Turn in full Report and suggestions -Patent and make millions $$$

Table 1: Planned Schedule

Actual Schedule

The key elements of our turbidostat can be broken up into three parts. They are the structural design, circuit design, and software implementation including our PI controller. Up until week four, our schedule was holding. Sample LEDs and photodiodes were received and tested to determine efficient LED wavelength. Over the course of weeks five, six, and seven, we focused primarily on the structural design portion which includes the plastic housing in which the 24 welled plate will rest, and the circuit design with help from one of our advisors on this project, Alex Leone. After receiving the circuits, testing and assembly began immediately along with code testing and to run the system and receive data, which is given later in the report.
The following schedule shows our actual work for the 10 weeks given.

Week 2	-Speak with advisors, client, order sample LEDs and microcontroller
Week 3	-Builded test equipment and determined the 605nm LEDs would work best -Tested microcontroller input and output -Modeled system in Simulink
Week 4	-Tested sample amber 605nm LED and photodiode for output readings
Week 5	-Farmiliarized and researched programming for arduino set up. -Continued work on our system controller -Began to model our housing unit in Solidworks
Week 6	-Revised schedule -Finished modeling complete housing unit -Began modeling LED and photodiode circuits
Week 7	-Ordered plastic housing pieces to be milled -Finished arduino code
Week 8	-Finished Printed Circuit Board layout -Ordered PCB -
Week 9	-Continued to farmiliarize with Python code for easier communication with liquid handler
Week 10	-Completed milling of housing unit -Worked on final report
Week 11	-Received PCB boards for both photodiodes and LEDs -Soldered each board and interfaced with arduino -Tested arduino code with completed housing unit -Tested output in frequency of photodiodes with samples -Produced calibration data with sample empty wells over 1.5 hours -Prepared constant output of readings for Demo

Table 2: Updated Schedule

Literature Review and Related Work

SOS Lab Turbidostat – Alex Leone

Prior work was done in the SOS lab to create a working turbidostat by Alex Leone, who is an undergraduate research assistant. Figure FIX shows a schematic of what was created previously. The set up shows two photodiodes: one used to measure the input light intensity and one to measure the output light intensity. Using this method, no calibration is needed because the input intensity subtracted by the output intensity gives the theoretical amount absorbed. Due to size limitations, the turbidostat we worked on this quarter could not use this method and does require calibration.

One of the limitations of the system previously created is that only one sample can be modified at a time. A prism shaped test tube is inserted in the device and this one sample is modified to obtain a constant population. With our project, 24 different wells can be modified at one time, giving more flexibility; however, smaller amounts are used in the wells of the plates as opposed to the prism-shaped test tube.

Another issue with the previously created set up is that, even though the schematic says an LED was used, a laser was used as a light source. This laser was very noisy and created issues with reading the values. Our project tried to improve on this issue by using LEDs with current drivers to supply a constant current to each LED. Not only does this eliminate some noise, but it also gives all of the LEDs the same reference point.

Needless to say, the previously developed turbidostat gave us great stepping blocks to create our project. We knew some of the limitations that come with creating something like this and were able to expand on what was previously done by implementing LEDs instead of lasers and reading 24 wells instead of one sample at a time.

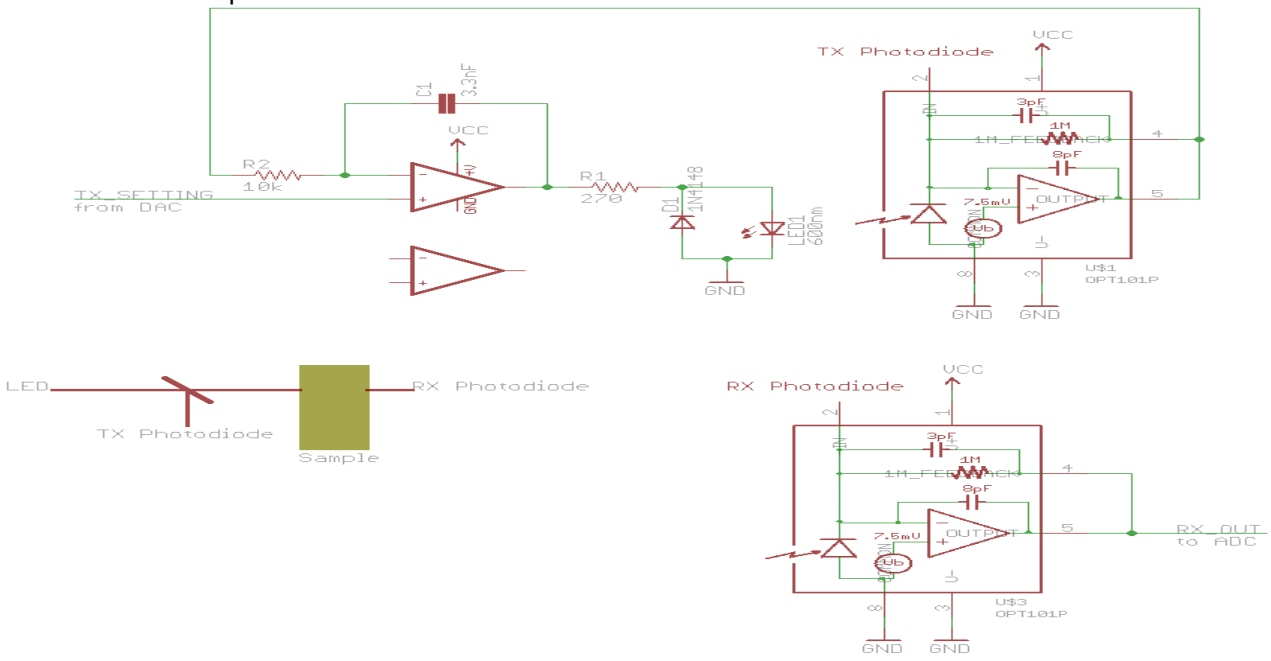


Figure 1 - Previous Turbidostat Created by Alex Leone

System Model and System Diagram

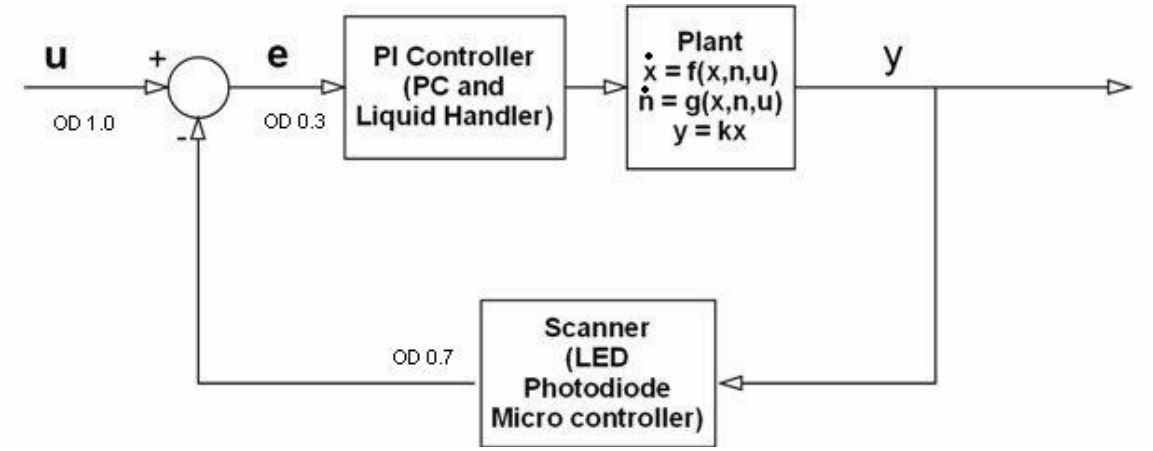


Figure 2: Block Diagram of turbidostat system

This is a total layout of what our system looks like and what we are trying to accomplish with the turbidostat. The inputs and variables are described as follows:

- x is the population of bacteria (g/L)
- n is amount of nutrients (g/L)
- u is amount of fresh nutrients being added to the system (g/L)
- Output from scanner is measured in optical density (OD) which is: $OD_{600} = Log_{10}(I_0/I)$

It is also important to understand from this diagram that there will be a reference point in OD to set the level at which we want to keep the population constant. As the scanner reads the optical density the summer will calculate the error and feed that signal to the controller. The controller will then output to a file for the liquid handler to read, then the liquid handler will execute pipeting out bacteria in order to maintain that population. The timing and execution of this pipeting will be operated on the liquid handler, shown in the following figure.

Figure 3: University of Washington SOS lab liquid handler

The very same liquid handler with its embedded software will be implementing the PI controller used for this system. This is due to the fact that all the necessary functions needed in order to control the population will be running in real time and using a micro controller will only increase the complexity and time to run the control program.

$$\dot{x} = \frac{vnx}{k+n} - ux$$

$$\dot{n} = -\gamma \frac{vnx}{k+n} + u(n_0 - n)$$

$$y = kx$$

Figure 4 - Equations for change in population (x dot) change in nutrition (n dot) and y (system output)

The equations given in the following figure were used to model the behavior of the bacteria and the nutrition given to them in a single well of a plate used in our turbidostat. The equations $\dot{x}$ and $\dot{n}$ are functions of x, n, and u ($\dot{x}$ =f(x,n,u) and $\dot{n}$ =g(x,n,u)) and are used to measure the change in the population and nutrition, respectively.

The following are all constants in these equations that are decided by the individual environment that is being worked in and the type of bacteria that is being worked with: v, k, γ, and n_o. The constants are described as following:

- v is used for the maximum growth rate and has the units of generations per hour
- k is the half saturation constant which designates half of the maximum population that can be achieved and is measured in grams per liter
- γ is the nutrient mass used per bacteria mass grown and this is unitless because it's a ratio
- n_o is the nutrient concentration in fresh media and is measured in grams per liter.

The states x and n represent the amount of bacteria and amount of nutrition in a well and are both measured in grams per liter. The variable u is used to signify the amount of nutrition that is put into the system and is measure in grams per liter. All 24 wells will behave in this same fashion and are modeled with these equations.

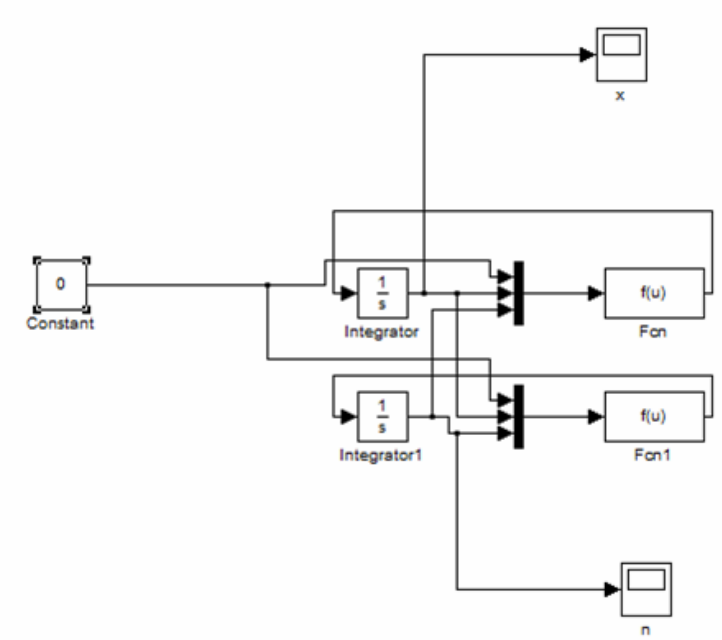


Figure 5 - Our System modeled in Simulink

The box on the left that is labeled "constant" is our u value, the box labeled "Fcn" on the right is our $\cdot\ x$, and the box labeled as "Fcn1" is our $\cdot\ n$. We used oscilloscope blocks to monitor the outputs of x and n. In order to get the values of x and n, we used an integrator block on our $\cdot\ x$ And $\cdot\ n$ blocks. The values x, n, and u were all then, in-turn, sent to our $\cdot\ x$ and $\cdot\ n$ blocks due to the fact that these blocks are functions of x, n, and u.

Performance Specifications

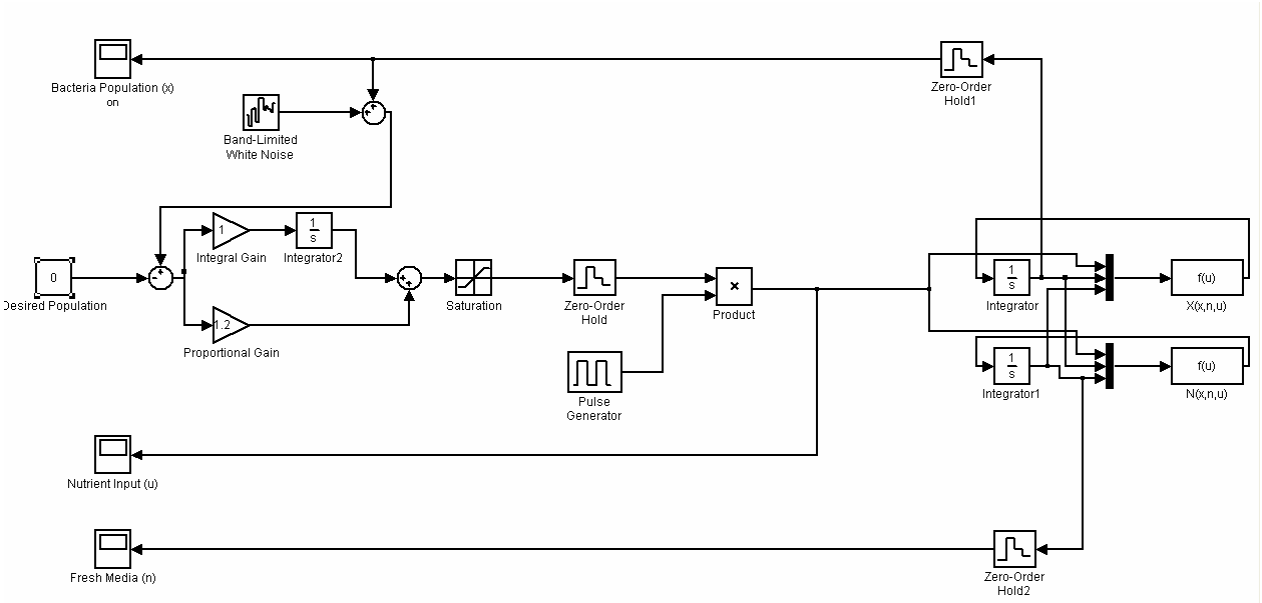


Figure 6: Simulink Model with Discrete Characteristics

Due to the fact our system is running periodically (ie liquid being dispensed every 15 minutes) we felt we needed to simulate our system model in simulink in discrete time. The following shows our simulation and desired results.

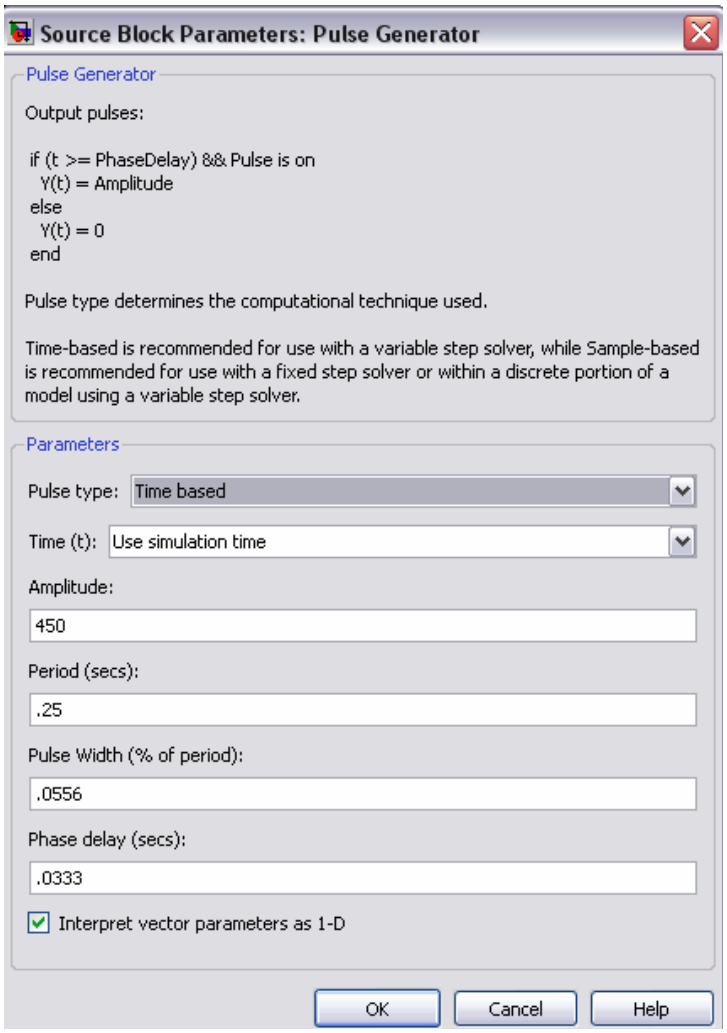


Figure 7: Pulse generator on controller output u

Amplitude = T / T_{on} = Period / Time on of period, 900 / 2 sec = 450
Period is 15 min. Units on simulation are relative to hours but simulink reads them as seconds. This was the mistake made in the presentation that I was not aware of.
Pulse Width is .0556% or about 2 seconds of period.
Phase delay is 2 min (not sure about this yet since we haven't finished the hardware)

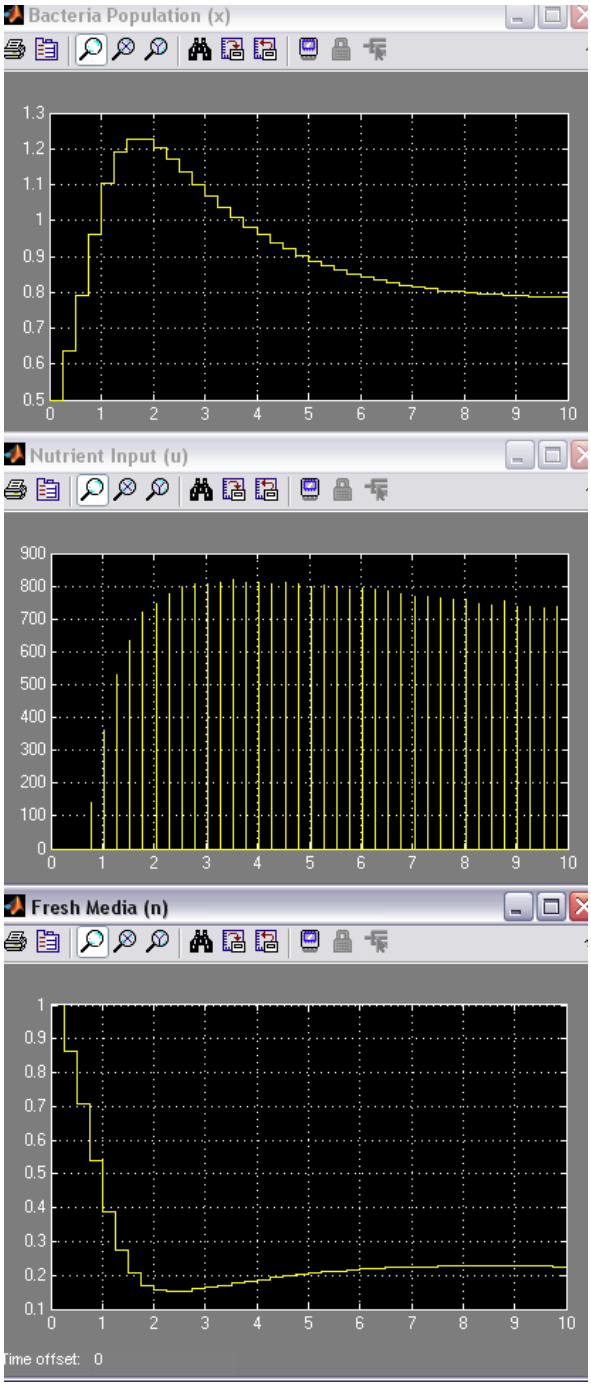


Figure 8: Simulations when desired population is 0.8

You can see the impulses which are what our system will be doing with the pipette mechanism.

Hardware and Software Design

Overview of Hardware, Electronics, and Software Design

In order to measure the optical density of the wells (which correlates with the population of bacteria in each well), a light needs be shone through each well and the light that is allowed through the solution needs to be measured using photodiodes. In order to read all of the 24 wells in a plate, two circuit boards were created. One board contains 24 LEDs and is placed under the plate pointed upwards to shine through the wells. On the other side of the plate, facing downward, are 24 photodiodes that are used to measure the light passed through the wells being measured. Both of these boards are interfaced with an Arduino breakout board which controls the LEDs that are on at any given time and the photodiode that is reading the light allowed through. Additionally, this Arduino board is connected via USB to a computer used to control a liquid handler and, using python script, the computer takes in the measurements of the optical density of the wells. These optical densities are processed through a PI controller in the python code which then writes a file of the amount of nutrients that need to be added to each well to approach the desired population. This file is then read by the liquid handler which dispenses the recorded u value to the respective wells every 15 minutes.

Hardware Specifics – LED Board

The following is a list of the components that were used to create the LED board (shown in Figures FIX and FIX):

- (2) Current Drivers - TLC5926
- (24) LEDs – HLMP-EJ10-XZ0DD-ND (Digikey Part Number)
- (5) 0.1 µF Capacitors and (1) 33 µF Capacitor (For Noise Control)
- (1) Temperature Sensor – TMP125

The light sources used were LEDs with a dominant wavelength of 600nm. The reason for choosing these LEDs is the fact that bacteria absorbs this wavelength while the nutrients allow it to pass through giving us the ability to measure the population density in each well. Current drivers were used to maintain a constant current through all of the LEDs. This minimized the issue of noise over the LEDs and made the system more stable. Additionally, the current drivers made interfacing very easy because they take in serial input which requires the use of only one pin from the Arduino for interfacing. Furthermore, the current driver interfacing was further simplified due to the fact that the serial out of the chip can be connected to the serial in of the second chip in a “daisy-chain” configuration (as shown in Figure FIX). A temperature sensor was also implemented on this board to check the effects of the LEDs on the environmental temperature of the enclosure.

The current drivers were sent commands by the Arduino using an SPI (Serial Peripheral Interface) bus. SPI normally involves four wires: Master In Slave Out, Master Out Slave In, Serial Clock, and Slave Select. For the purposes of the current drivers, data was not needed from the driver to the Arduino, making the

Master In Slave Out line irrelevant, and the slave select was not needed due to the daisy-chain configuration. The Master Out Slave In and Serial Clock lines were used, along with a latch enable line that was manually pulsed to set the output of the current driver. The LEDs are controlled by outputting data 4 times, sending 8 bits at a time (32 outputs total from the drivers, only 24 used).

It is easy to notice a kind of odd configuration of the LEDs on the second daisy-chained current driver. There is a large space in the middle of the LED outputs. The reason for wiring the LEDs to the current driver in this fashion is to make running the wires on the PCB easier. The IC has 8 different LED outputs on each side and if they were all wired to the first eight bits or the last eight bits, then the layout of the board would be kind of odd with all of the LEDs hooked up to one side of the current driver. The wiring was basically done this way to keep the board somewhat symmetrical and to make IC placement easier.

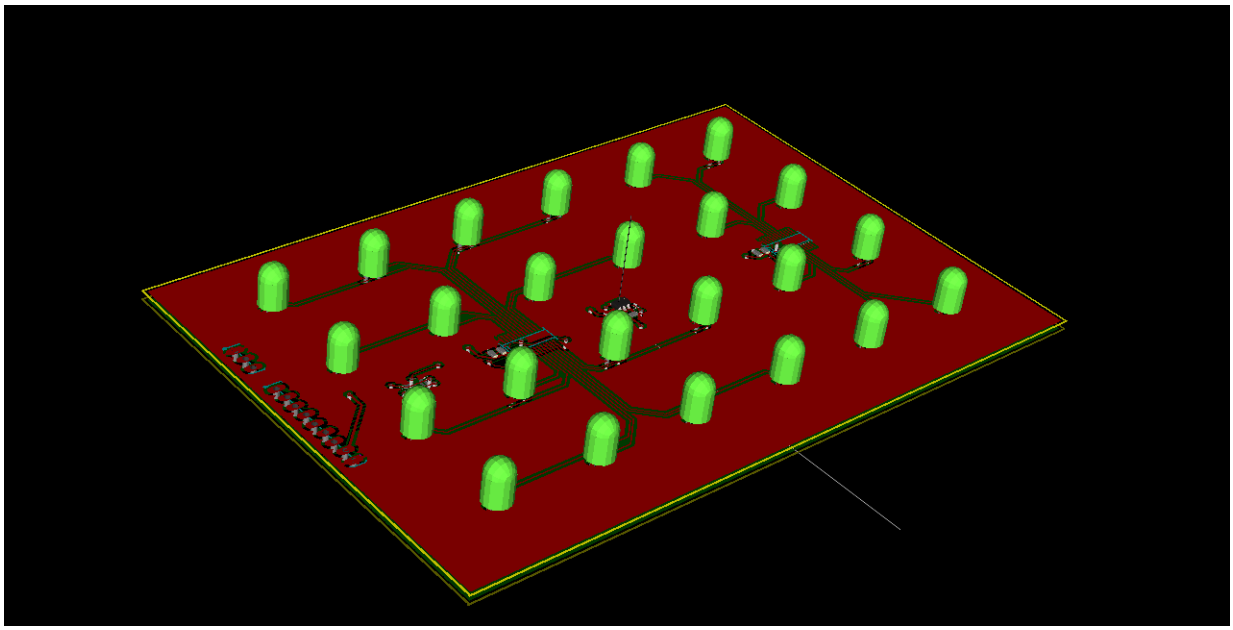


Figure 9 - 3D View of the LED Circuit Board

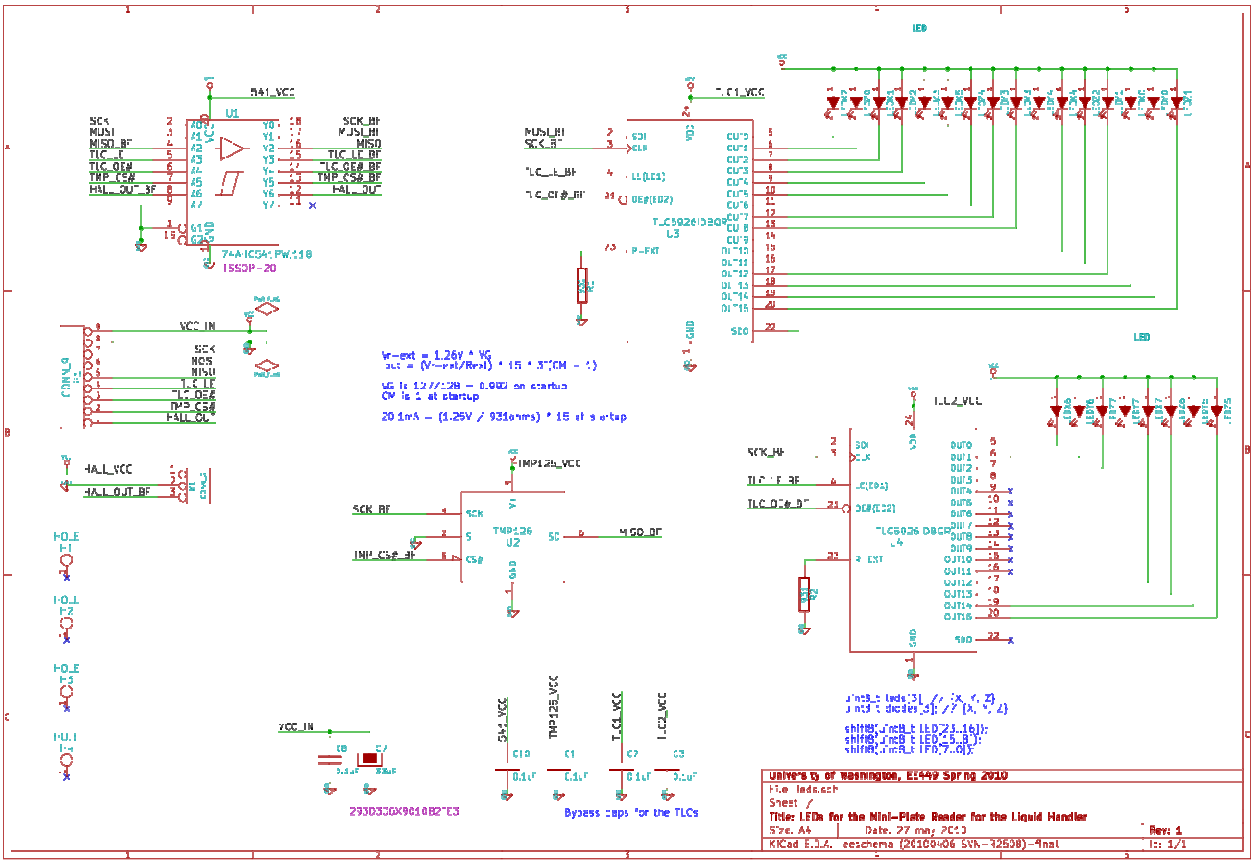


Figure 10 - Schematic of the LED Board

Hardware Specifics – Photodiode Board

The Following is a list of the components that were used to create the photodiode board (shown in figures FIX and FIX):

- (3) Shift Registers – 74LS595
- (3) 3 to 8 Decoders – 74LS138
- (24) Photodiodes (Light to Frequency Converters) – TSL230

The photodiodes that are used are light to frequency converters that give out a square wave with a frequency that's dependent on the light intensity shone upon them. These integrated circuits were used due to the small amount of analog to digital converters available on the Arduino board. Three different "blocks" of photodiodes are used in order to send the control lines to the photodiodes efficiently. To further elaborate, we wanted to use small, surface mount shift registers due to the low amount of real estate on the PCB. If we controlled all 24 of the LEDs from one shift register, we would need 5 bits for selecting the correct photodiode to measure from and would need a 16 bit output from the shift register. This would make for a bulky 24 pin IC that would be tough to situate on the board.

While interfacing with the shift registers, SPI was used once again in the same configuration. The three registers are also daisy-chained together so that data can efficiently be sent to all of them without taking up too many pins. The data that is sent out to the shift registers consists of eight bits: the select lines (A0-A2), the output enable (active low), and the sensitivity control lines (S0-S3). The select lines select which photodiode is being read in the block by sending those three bits to a decoder that has the outputs hooked up to the output enables of the photodiodes. The output enable can be used to turn off all of the photodiodes in the block if need be. The sensitivity controls are used to scale the frequency or to adjust the sensitivity of the photodiodes (the ranges of frequencies sent out) as shown in Tables FIX and FIX.

S1	S0	SENSITIVITY
L	L	Power down
L	H	1×
H	L	10×
H	H	100×

Table 3 - Sensitivity Adjustment Using S0 and S1

S3	S2	f_O SCALING (divide-by)
L	L	1
L	H	2
H	L	10
H	H	100

Table 4 - Frequency Scaling Using S2 and S3

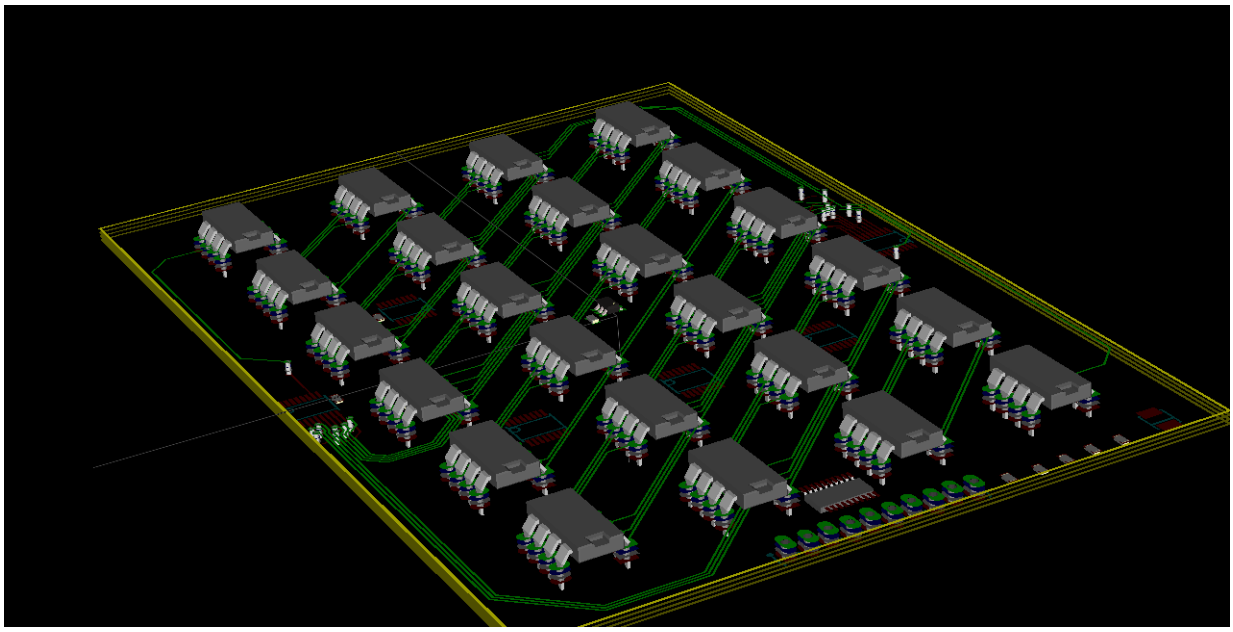


Figure 11 - 3D View of the Photodiode Board

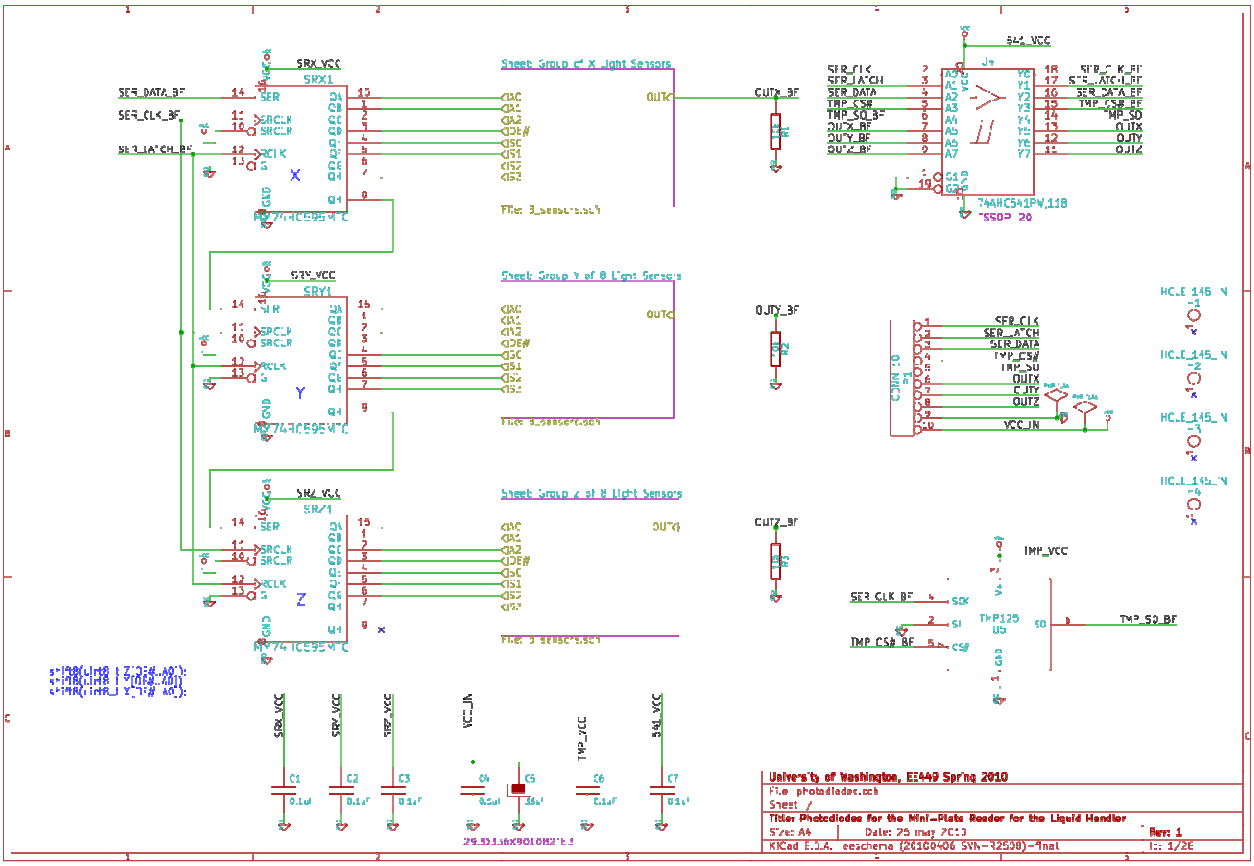


Figure 12 - The Shift Register Interface of the Photodiode Board

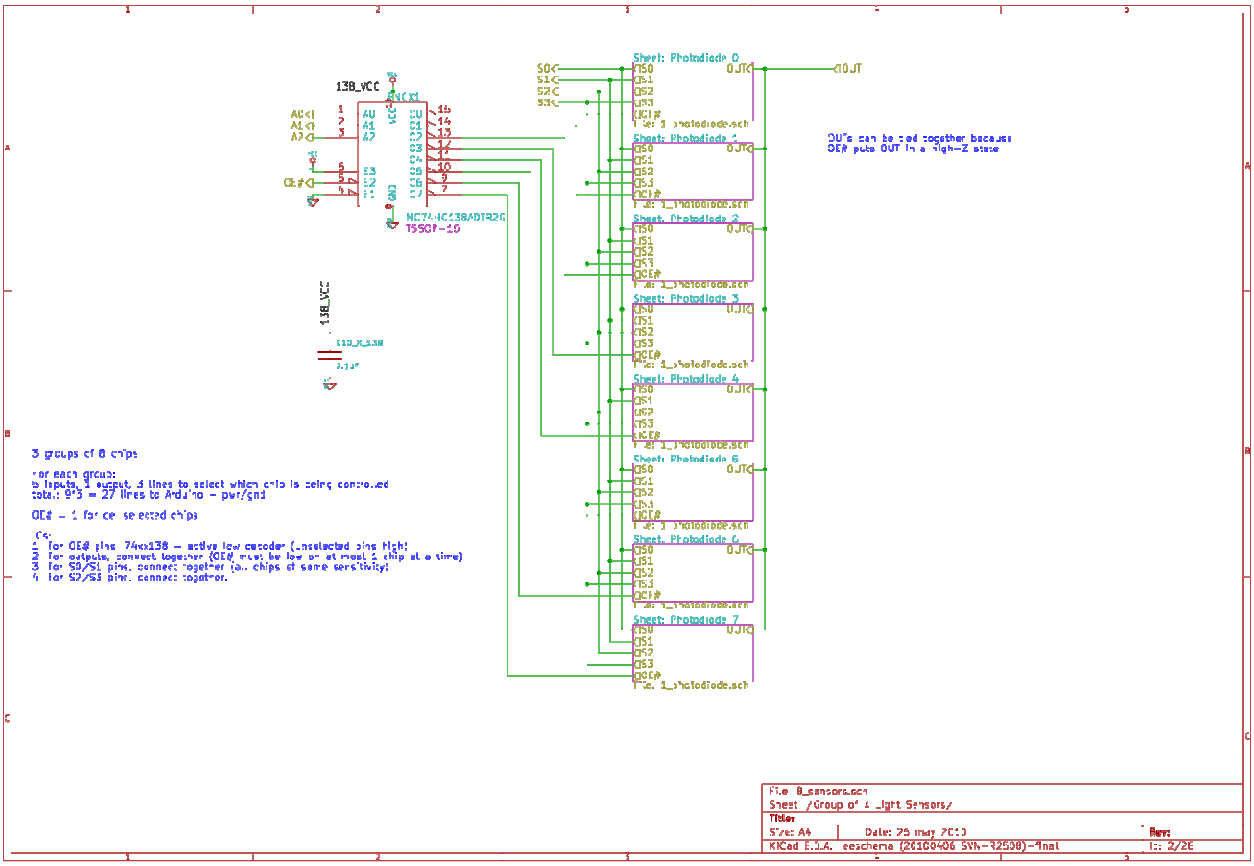


Figure 13 - The Internals of Each "Block" on Photodiode Board

Housing Unit Specifics

The turbidostat design project consists mainly of hardware design and programming the hardware to do what we want it to do. There are two main categories for which the hardware is made up of, that is the plastic housing unit, and the printed circuit boards containing the surface mounted LED's and photodiodes. To design and make these have consumed a considerable amount of time, but when finished we will have a fully functional plate scanner ready to read optical densities in a 24 big well plate.

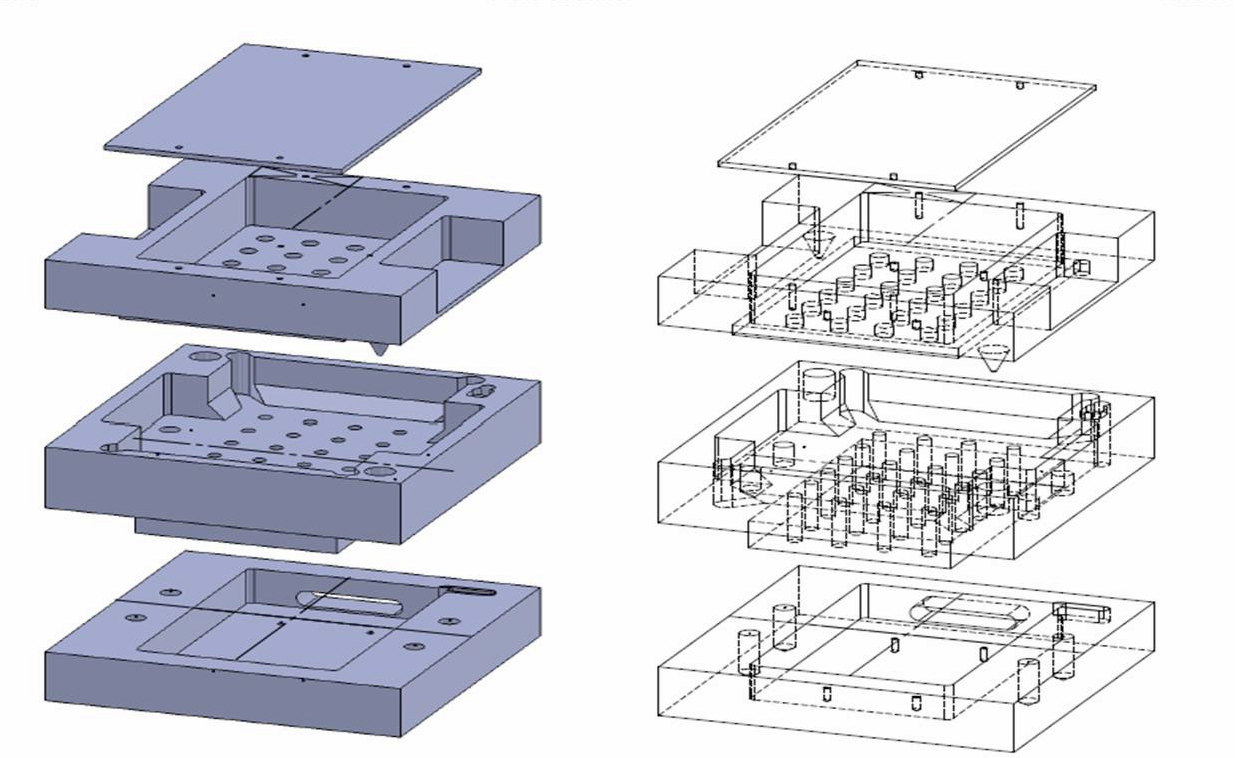


Figure 14: Scanner assembly where printed circuit boards (top and bottom) and plate (center) will sit

This is currently the design we used for our plate scanner. We ran into some issues financially in getting it made with the budget we have. In order to achieve the best quality paying someone to mill it on a CNC is far too expensive which caught us by surprise. The other option is to rapid prototype, or use a 3D printer, but our customer has concerns about the precision of the printer and there is also that same price issue in getting it made that way. The solution that we came up with was to use the CNC located in the SOS lab on the third floor of the electrical engineering building. Although learning how to work new equipment later on in our project (week 8) was time consuming, this was necessary to keep within our budget.

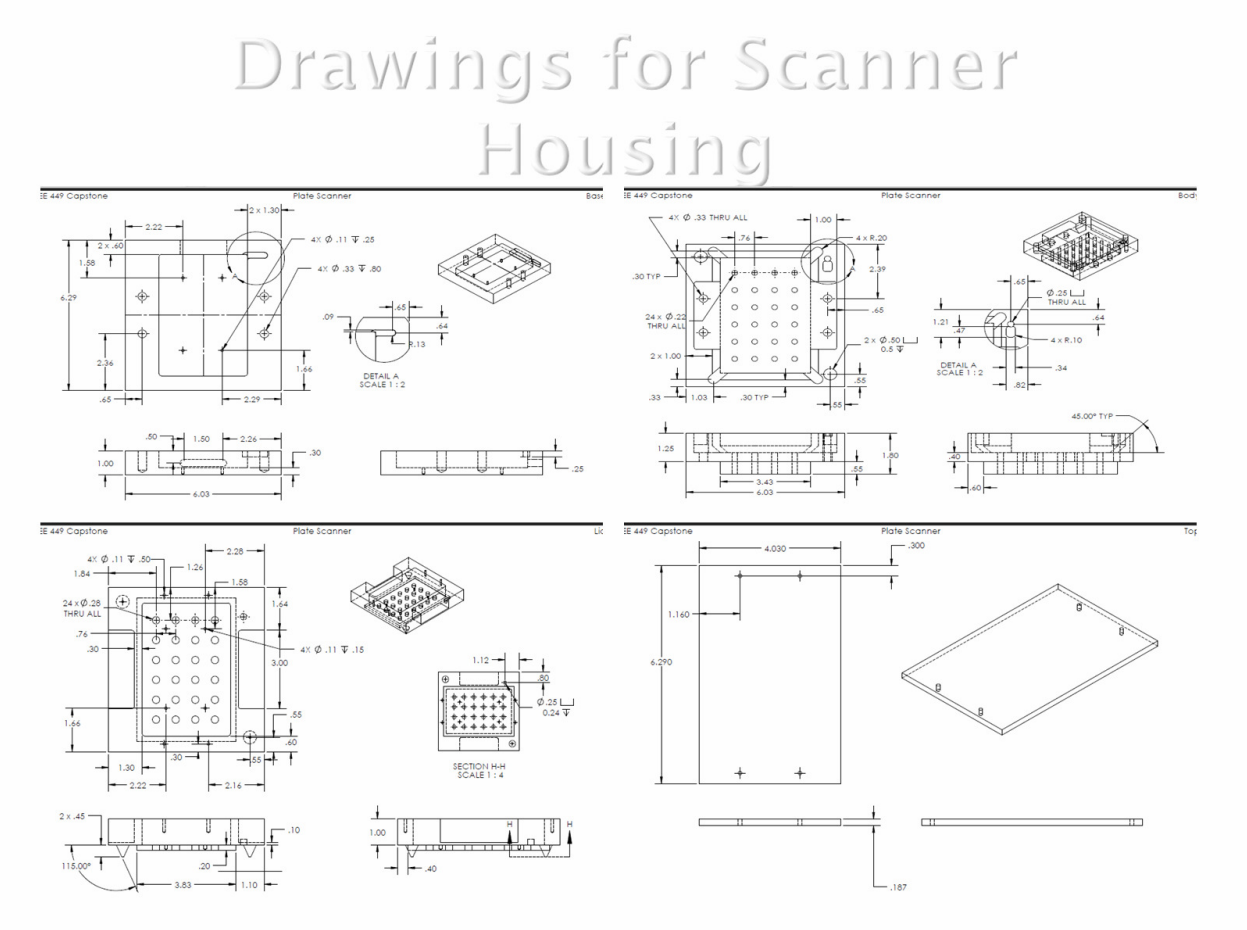


Figure 15: Drawings for scanner housing

These are the drawings that make up the assembly. It is easy to see that there are a lot of little details in the design of all three of the major parts. Some difficulty was found from converting the design on paper to a useable schematic for milling. Getting an estimate on making it seemed to be rather troublesome, due to the functionality of the CNC mill with the square corners and 45 degree chamfer on the inside wall. Some revision had to be done before getting an accurate estimate. The estimate for the above drawing was between $1,200.00 – $1,700.00, far beyond our budget. As simple as the design already stands, trying to make it simpler would exclude some of the key features that are needed in our plate scanner to get it to run efficiently.

Our completed housing unit and circuits photos can be seen in APPENDIX E.

Software Specifics

Python and Liquid Handler Communication

Python was used to communicate between the computer for the liquid handler and the board because of the high level nature of language and the ability to write and read files, communicate serially, and provide graphical data through the console. The python code can be found in Appendix A. Figure FIX shows the flowchart of the data used to control the populations of the wells. The frequencies of the photodiodes are constantly being transferred from the Arduino to the python code using half second interrupts in the Arduino code. This data is then passed into the PI controller which finds out how many nutrients need to be added to the wells in order to approach the desired value. When these values are calculated, they are stored into a temporary file that is then transferred to a "locked" file when all of the values have been recorded. This method is used to safeguard the liquid handler from reading an incomplete file. The liquid handler then requests this information every 15 minutes and reads the values saved into the "locked" file. The nutrients are then dispersed to their respective wells and the liquid handler waits another minutes until it reads the u values from the locked file and dispenses to the wells again.

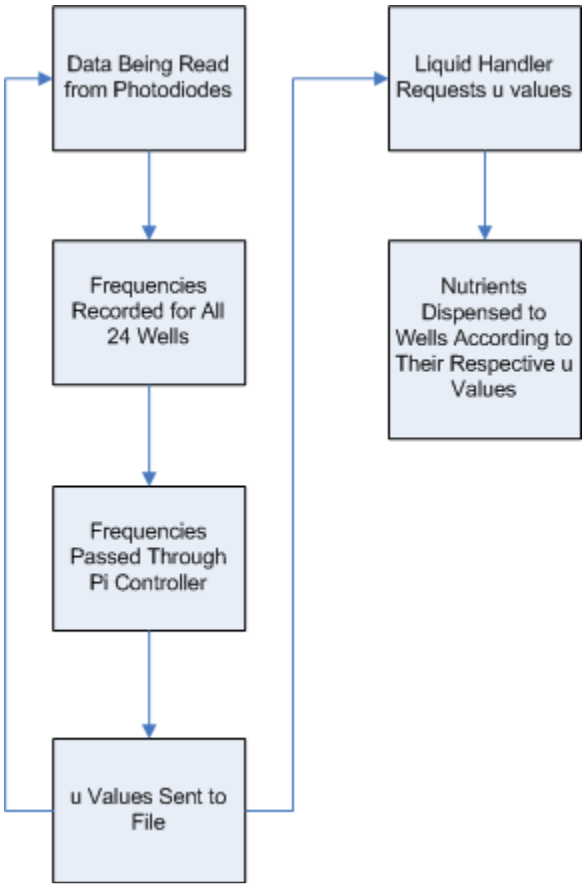


Figure 16 - Data Flowchart of Python Code and Liquid Handler

Arduino and Python Communication

The python code is used to communicate via USB interface with the Arduino board. A half second interrupt is set up in the Arduino code that reads the frequency of the current photodiode and then sends this value out to the python code. Python then sends the control signals for the next LED that should be turned on and this exchange of information continues as long as the program is running. As stated in the last section, the python code takes this information and manipulates it to write the respective u values to a file for the liquid handler.

Arduino Code

Using the control lines that are received from the python code, the Arduino sends out the serial data to the current drivers and shift registers every half a second. This data selects which LED is on and which photodiode is reading (these are obviously paired, with the LED that is on directly below the photodiode that is reading).

Data from Experiments

To verify some of the assumptions made when creating the turbidostat, some experiments were performed and data was taken. Of the experiments performed, the applicable ones were the LED spectrum reading to verify the dominant wavelength was ≈ 600nm and the photodiode frequency readings over a 90 minute time period.

LED Spectrum Reading

In order to verify the dominant wavelength of the LEDs, a spectrum analysis was run on the LED. The reading is shown in Figure FIX. As can be seen by the figure, the dominant wavelength of the LED is right around 605 nm at an intensity of 7330.73 milli candelas. Table FIX shows the specific measurements of the analysis of the LED at a wavelength of 600 nm. As can be seen, the intensity is 7,112.3803 milli candelas which is 97% of the peak intensity of the analysis and this makes the LEDs suitable for the turbidostat application.

Wavelength	Raw Data	Dark	Reference	Dark Subtracted	Transmission	Absorbance
600	7112.3803	2569.269	2555.22	4543.1113	0	6.8165

Table 5 - Spectrum Analysis Data for LEDs at 600 nm

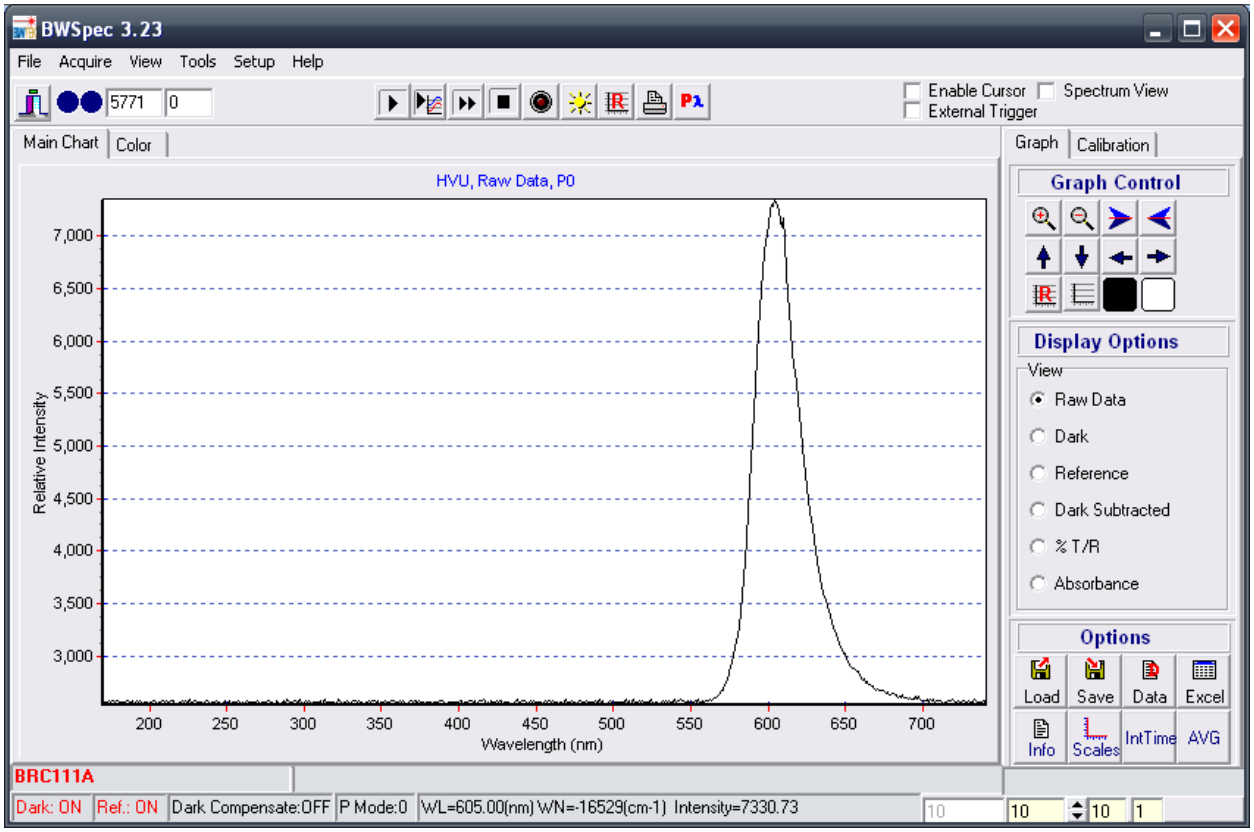


Figure 17 - Spectrum Analysis of the LEDs

Photodiode Frequency Readings Over a 90 Minute Period of Time

In order to get a good idea of the fluctuation of the photodiode readings after everything was assembled, the program was run for 90 minutes and the frequency readings were recorded as if an actual test was being done (cycling through all LEDs and taking respective frequency readings of the photodiodes). Figure FIX shows the readings from all of the photodiodes during the 90 minute time period. It's obvious that there is a lot of fluctuation between the photodiodes with the maximum frequency of 13,058 Hz and a minimum of 8,194 Hz. This is a 37.25% fluctuation and is unacceptable when measuring the wells. For this reason, the photodiodes will need to be calibrated with an initial reading with a constant concentration dispersed over all 24 wells in order for the readings to be accurate. To verify that the photodiodes themselves didn't have much fluctuation, the data taken from photodiode X_0 (location shown in Figure FIX) was graphed in Figure FIX and showed little variance over the 90 minute time period. The maximum frequency read by this photodiode over this time period was 10,332 Hz and the minimum was 10,112 Hz. This gives a 2.13% variation over this period, which is more than acceptable.

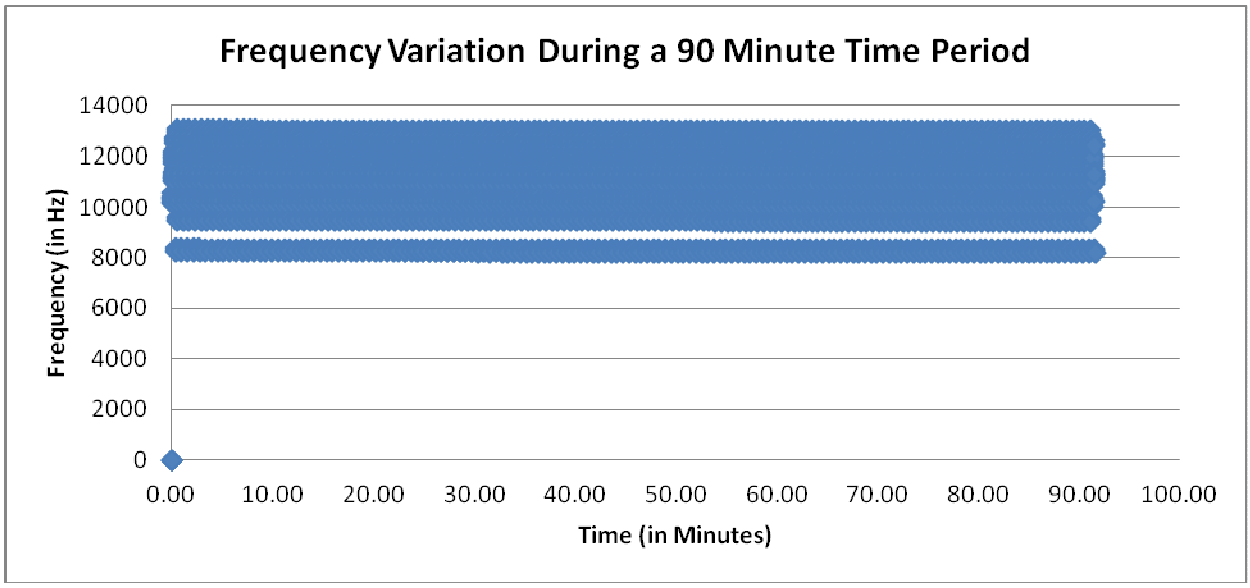


Figure 18 - Frequency Readings of All Photodiodes Over 90 Minute Period

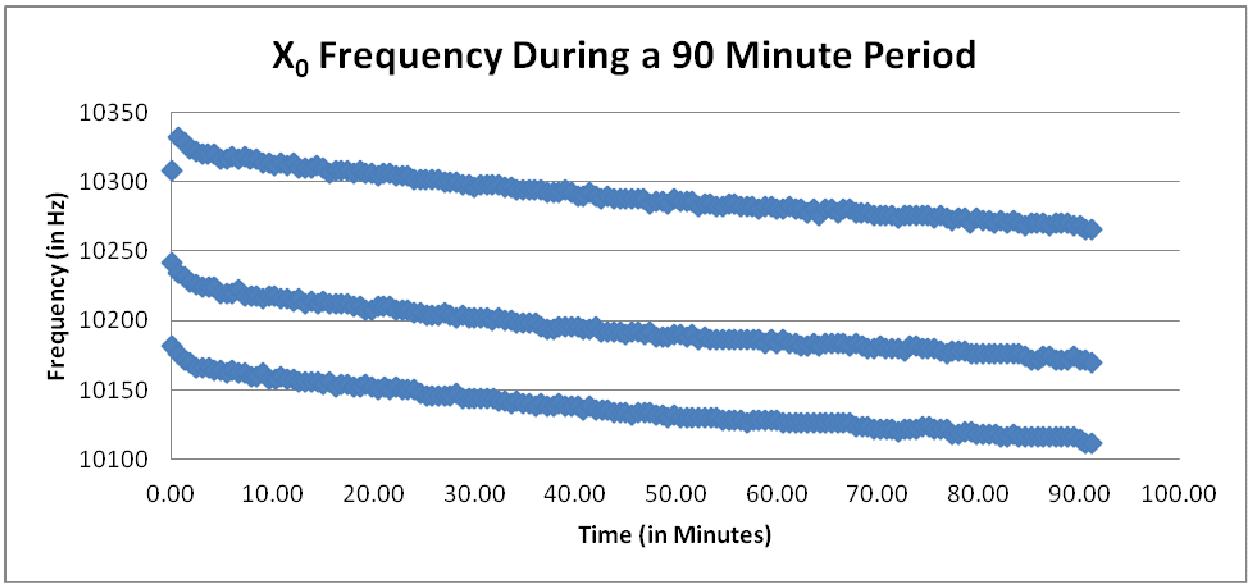


Figure 19 - Frequency Readings of Photodiode X0 During a 90 Minute Period

Conclusions and Recommendations

The initial goal of this project was to create a turbidostat by creating a plate reader that could interface with the liquid handler to add certain amounts of nutrients to maintain a designated population in each well. Though we had time constraints that affected us being able to create a completely automated system, the bulk of what is needed (plate reader) to create what we defined in our goal has been achieved. We have created an LED board and a photodiode board that can read the optical densities of all 24 wells in a plate. These boards can interface with an Arduino that can, in turn, communicate through USB with the liquid handler computer. The code on this computer can use a controller to find how much liquid needs to be dispensed to each well and write that to a file. A

chemostat has been created that takes files (like the one we created in python) and dispenses the given liquid to the wells. If we had more time to work on this project it would really only be a matter of calibrating the photodiodes, coordinating with the gripper, installing a hall effect sensor (the code is already written), and interfacing with the liquid handler using the code that's already written for the chemostat. We are confident that using what we have created over the last quarter, the research assistants in the lab will be able to set up the required interface to get a fully automated, fully functional turbidostat working within a week or two.

References

Proctor, Michael, Malene L. Urbanus, Eula L Fung, Daniel F. Jaramillo, Ronald W. Davis, Corey Nislow, and Guri Giaver. The Automated Cell, Compound and Environment Screening System (ACCESS) for Chemogenic Screening.

Klavins, Eric, Alex Leone, Safarik, Baker, Lidstrom, and Black. The Controlled Evolution Project

Nise, Norman. Control Systems Engineering, 5th Edition. Wiley, 2007.

Hedrick, J.K. and A. Girard. Control of Nonlinear Dynamic Systems: Theory and Applications: Controllability and Observability of Nonlinear Systems. 2005.

Smith, H.L. Bacterial Growth.

Petersen-Mahrt, S.K., R.S. Harris, and M.S. Neuberger. AID Mutates E. Coli Suggesting a DNA Deamination Mechanism for Antibody Diversification. Nature, 2002.

Hoskisson, P.A. and G. Hobbs. Continuous Culture – Making a Comeback?. Microbiology, 2005.

Schwaneberg, U., D. Zhurina, and T.S. Wong. The Diversity Challenge in Directed Protein Evolution, Combinatorial Chemistry & High Throughput Screening. 2006.

Lombardi, A.T. and P.J. Wangersky. Influence of Phosphorous and Silicon on Lipid Class Production by the Marine Diatom Chaetoceros gracilis Grown in Turbidostat Cage Cultures. 1991.

Arnold, F.H. Evolutionary Approaches to Protein Design. 2000.

Rinaldi, Sergio, O.D. Feo, and A. Gragnani. Food Chains in the Chemostat: Relationships Between Mean Yield and Complex Dynamics. 1998

APPENDIX A – PYTHON CODE – Communication.py

```
"""
Communication:

Arduino to Computer:

"""

class Data:
    pass

class DiscretePIController:
    def __init__(self, led='X0'):
        self.led = led
        self.ref_od = 0.5
        self.was_in_range = True


    def update(self, new_od):
        error = new_od - self.ref_od
        u = error * self.kp + error * self.ki

        if self.was_in_range:
            self.ki += error

        if u < 0 or u > 1:
            self.was_in_range = False
            u = min(1, max(0, u))

        return u

    def __str__(self):
        return self.u

class LedState:
    """
    Represents the led state.
    """
    def __init__(self, Xs=[False]*8, Ys=[False]*8, Zs=[False]*8):
        self.Xs = Xs
        self.Ys = Ys
        self.Zs = Zs

    def state_from_arduino(self, len3_u8_list):
        self.Zs[5] = True if len3_u8_list[0] & 0x80 else False
        self.Ys[5] = True if len3_u8_list[0] & 0x40 else False
        self.Zs[6] = True if len3_u8_list[0] & 0x20 else False
        self.Xs[7] = True if len3_u8_list[0] & 0x10 else False

        self.Ys[7] = True if len3_u8_list[0] & 0x08 else False
        self.Zs[7] = True if len3_u8_list[0] & 0x04 else False
        self.Ys[6] = True if len3_u8_list[0] & 0x02 else False
```

```
        self.Xs[6] = True if len3_u8_list[0] & 0x01 else False

        self.Zs[1] = True if len3_u8_list[1] & 0x80 else False
        self.Ys[0] = True if len3_u8_list[1] & 0x40 else False
        self.Xs[0] = True if len3_u8_list[1] & 0x20 else False
        self.Ys[1] = True if len3_u8_list[1] & 0x10 else False
        self.Zs[2] = True if len3_u8_list[1] & 0x08 else False
        self.Xs[4] = True if len3_u8_list[1] & 0x04 else False
        self.Ys[4] = True if len3_u8_list[1] & 0x02 else False
        self.Xs[3] = True if len3_u8_list[1] & 0x01 else False

        self.Ys[3] = True if len3_u8_list[2] & 0x80 else False
        self.Zs[4] = True if len3_u8_list[2] & 0x40 else False
        self.Xs[5] = True if len3_u8_list[2] & 0x20 else False
        self.Zs[3] = True if len3_u8_list[2] & 0x10 else False
        self.Ys[2] = True if len3_u8_list[2] & 0x08 else False
        self.Xs[1] = True if len3_u8_list[2] & 0x04 else False
        self.Zs[0] = True if len3_u8_list[2] & 0x02 else False
        self.Xs[2] = True if len3_u8_list[2] & 0x01 else False

    def state_to_arduino(self):
        """
        Returns the four bytes to be sent to the Arduino.  Send in
        the same order as the list.
        """
        return [

            (self.Zs[5] << 7) |
            (self.Ys[5] << 6) |
            (self.Zs[6] << 5) |
            (self.Xs[7] << 4) |

            (self.Ys[7] << 3) |
            (self.Zs[7] << 2) |
            (self.Ys[6] << 1) |
            (self.Xs[6] << 0),

            (self.Zs[1] << 7) |
            (self.Ys[0] << 6) |
            (self.Xs[0] << 5) |
            (self.Ys[1] << 4) |
            (self.Zs[2] << 3) |
            (self.Xs[4] << 2) |
            (self.Ys[4] << 1) |
            (self.Xs[3] << 0),

            (self.Ys[3] << 7) |
            (self.Zs[4] << 6) |
            (self.Xs[5] << 5) |
            (self.Zs[3] << 4) |
            (self.Ys[2] << 3) |
            (self.Xs[1] << 2) |
            (self.Zs[0] << 1) |
            (self.Xs[2] << 0)
        ]

    def get_leds_on(self):
        """
```

```
        Returns a list of strings "X0" , etc for which leds are on.
        """
        on = []
        for i in range(8):
            if self.Xs[i]:
                on.append("X" + str(i))
            if self.Ys[i]:
                on.append("Y" + str(i))
            if self.Zs[i]:
                on.append("Z" + str(i))
        return on

    def set_led_on(self, s):
        """
        takes 'X0' and updates the internal data structure.
        """
        if s[0] == 'X':
            self.Xs[int(s[1])] = True
        elif s[0] == 'Y':
            self.Ys[int(s[1])] = True
        elif s[0] == 'Z':
            self.Zs[int(s[1])] = True

    def clear(self):
        self.Xs = [False] * 8
        self.Ys = [False] * 8
        self.Zs = [False] * 8



class Photod8State:
    S10 = {
        "Power down": 0b00,
        "1x": 0b01,
        "10x": 0b10,
        "100x": 0b11
    }
    S32 = {
        1: 0b00,
        2: 0b01,
        10: 0b10,
        100: 0b11
    }
    def __init__(self):
        self.clear()

    def to_byte(self):
        return ((self.address & 0x07) |
                ((not self.OE) << 3) |
                (self.S10[self.sensitivity] << 4) |
                (self.S32[self.f0_scaling] << 6))

    def clear(self):
        self.f0_scaling =  1
        self.sensitivity = "100x"
        self.OE = False
        self.address = 0
```

```
    def copy_other(self, other, address=None):
        self.f0_scaling = other.f0_scaling
        self.sensitivity = other.sensitivity
        self.OE = other.sensitivity
        if address is None:
            self.address = other.address
        else:
            self.address = address


class PhotodiodeState:
    def __init__(self):
        self.X = Photod8State()
        self.Y = Photod8State()
        self.Z = Photod8State()

    def clear(self):
        self.X.clear()
        self.Y.clear()
        self.Z.clear()

    def set_photod_on(self, s, settings):
        """
        s is a 'X0', etc.
        """
        if s[0] == 'X':
            self.X.copy_other(settings, address=int(s[1]))
        elif s[0] == 'Y':
            self.Y.copy_other(settings, address=int(s[1]))
        elif s[0] == 'Z':
            self.Z.copy_other(settings, address=int(s[1]))

    def get_bytes(self):
        return [self.Z.to_byte(), self.Y.to_byte(), self.X.to_byte()]



def u16_temp_to_float(u16):
    """
    Converts the TMP125 10-bit two's-compliment to the human readable
    temperature value.
    """
    if u16 & 0x200:
        u16 = -(((~u16) & 0x3ff) + 1)
    return u16 * 0.25

def read_packet(ser, data):
    count_bytes = map(ord, ser.read(2))
    data.count = (count_bytes[0] << 8) | count_bytes[1]

    temp_bytes = map(ord, ser.read(3))
    data.just_switched = 1 if (temp_bytes[0] & 0x80) else 0
    data.hall_effect = 1 if (temp_bytes[0] & 0x40) else 0
    data.pd_temp = u16_temp_to_float(
            ((temp_bytes[0] & 0x3f) << 4) |
            ((temp_bytes[1] & 0xf0) >> 4) )
    data.led_temp = u16_temp_to_float(
```

```
            ((temp_bytes[1] & 0x0f) << 8) |
            temp_bytes[2]   )

    tlc_bytes = map(ord, ser.read(3))
    data.led_state.state_from_arduino(tlc_bytes)

    data.pd_bytes = map(ord, ser.read(3))

    period_bytes = map(ord, ser.read(2))
    data.period = (period_bytes[0] << 8) | period_bytes[1]

    data.pd_sel = ord(ser.read(1))

    newline = ser.read(1)

def u32_to_u8_list(u32):
    """
    Returns a list of 4 bytes, most-significant-byte of the u32 first.
    """
    return [(u32 >> 24) & 0xff,
            (u32 >> 16) & 0xff,
            (u32 >> 8) & 0xff,
            u32 & 0xff]

def u8_list_to_u32(u8_list):
    """
    Opposite of u32_to_u8_list.
    """
    return ( ((u8_list[0] & 0xff) << 24) |
             ((u8_list[1] & 0xff) << 16) |
             ((u8_list[2] & 0xff) << 8) |
             (u8_list[3] & 0xff) )

def period_to_u16(period):
    u16_period = int(round(period / (1024/16.0e6)))
    #actual_period =
    return u16_period

def send_packet(ser, data):
    chrs = (['n'] + map(chr, data.tlc_bytes) + map(chr, data.pd_bytes) +
            [chr((data.period >> 8) & 0xff),
             chr((data.period) & 0xff),
             chr((data.pd_sel) & 0x03)])
    ser.write(''.join(chrs))

NEXT_LEDS = ("X0 Y1 Z2 X4 Y5 Z6 Y0 Z1 X3 Y4 Z5 X7 Z0 X2 Y3 Z4 X6 Y7 X1 "
           "Y2 Z3 X5 Y6 Z7")
# NEXT_LEDS = ("X0 Z7")

def get_next_led_on(led_state):
    """
    Returns 'X0', etc.
    """
    on = led_state.get_leds_on()
    if not on or on[0] == 'Z7':
        next_led_str = "X0"
    else:
        this_led_i = NEXT_LEDS.find(on[0])
```

```
        next_led_str = NEXT_LEDS[this_led_i + 3 : this_led_i + 5]
    return next_led_str



def switch_leds(ser, received_data, data_to_send, prefs):
    next_led_str = get_next_led_on(received_data.led_state)

    data_to_send.led_state.clear()
    data_to_send.led_state.set_led_on(next_led_str)

    data_to_send.photod_state.clear()
    data_to_send.photod_state.set_photod_on(next_led_str,
                                    data_to_send.photod_settings)

    data_to_send.tlc_bytes = data_to_send.led_state.state_to_arduino()
    data_to_send.pd_bytes = data_to_send.photod_state.get_bytes()
    data_to_send.period = period_to_u16(prefs.count_period)
    data_to_send.pd_sel = 'XYZ'.find(next_led_str[0])

    send_packet(ser, data_to_send)

def freq_to_OD(freq, led_on):
    return (20000 - freq)*2.0 / 20000

def _main():
    import datetime
    import matplotlib.pyplot as plt
    import serial
    import sys
    import time
    import traceback
    import pprint
    pp = pprint.PrettyPrinter(indent=4)
    ser = serial.Serial('COM3', baudrate=115200)
    ser.setTimeout(0.5)
    last_time = time.time() + 1.77
    this_time = time.time()
    current_led = 0

    period = 0.5 # period in seconds

    prefs = Data()
    prefs.periods_per_switch = 3
    prefs.count_period = 0.5
    prefs.ki = 0.1
    prefs.kp = 0.1

    f = open('hourSamples.txt','w')


    received_data = Data()
    received_data.led_state = LedState()


    data_to_send = Data()
    data_to_send.led_state = LedState()
```

```
        data_to_send.photod_state = PhotodiodeState()

        photod_settings = Photod8State()
        photod_settings.OE = True
        photod_settings.sensitivity = '100x'
        photod_settings.f0_scaling = 1
        data_to_send.photod_settings = photod_settings

        #plt.ion()
        #plt.figure(1)

        controllers = {}
        control_output = {}
        for s in 'XYZ':
           for i in range(8):
              controller = DiscretePIController()
              controller.ki = prefs.ki
              controller.kp = prefs.kp
              controller.led = s + str(i)
              controllers[controller.led] = controller


        '''
        data.period = period_to_u16(period)
        print "period: %s seconds (%s counts)" % (period, data.period)
        '''

        periods_till_switch = prefs.periods_per_switch

        #x = []
        #y = []

        avgOD = 0
        avgOD_n = 0

        try:
           while True:
              c = ser.read(1)
              if len(c) > 0 and c == 'c':
                 read_packet(ser, received_data)

                 leds_on = received_data.led_state.get_leds_on()
                 led_on = 'None' if not leds_on else leds_on[0]
                 freq = received_data.count / prefs.count_period
                 OD = freq_to_OD(freq, led_on)
                 f.write(str(time.time()) + "," + str(led_on) + "," + str(freq) + "," + str(OD) + "," + str(received_data.led_temp) + "," +
str(received_data.pd_temp) + "\n")
                 print "[%s] photod: %s  freq: %5.2f  OD: %.3f temps(leds:%.2f photod:%.2f C)" % (
                       ' ' if received_data.just_switched else '*',
                       led_on,
                       freq,
                       OD,
                       received_data.led_temp,
                       received_data.pd_temp)
                 #x.append(datetime.date.today())
                 #y.append(received_data.led_temp)
                 #plt.plot(x, y)
                 #plt.draw()
```

```
            if not received_data.just_switched:
                avgOD += OD
                avgOD_n += 1

            if periods_till_switch == prefs.periods_per_switch:
                if avgOD_n:
                    avgOD = avgOD / avgOD_n
                    if leds_on:
                        u = controllers[led_on].update(OD)
                        control_output[led_on] = u
                        print ("NEW CONTROL OUTPUT for %s: %.3f  "
                               "(average OD: %.2f)") % (
                            led_on, u, avgOD)
                    avgOD = 0
                    avgOD_n = 0
                    print

            periods_till_switch -= 1
            if periods_till_switch == 0:
                periods_till_switch = prefs.periods_per_switch
                # switch leds
                switch_leds(ser, received_data, data_to_send, prefs)



    except KeyboardInterrupt:
        pass
    except:
        traceback.print_exc(file=sys.stdout)
    finally:
        ser.close()


def test_read_packet():
    class FakeSerial:
        def __init__(self, buffer=''):
            self.buffer = buffer

        def read(self, n):
            r = self.buffer[:n]
            self.buffer = self.buffer[n:]
            return r

        def write(self, s):
            self.buffer += s

    ser = FakeSerial()
    count = 955
    ser.write(chr(count >> 8))
    ser.write(chr(count & 0xff))

    just_switched = 1
    hall_effect = 1
    pd_temp = 534
    led_temp = 489
    ser.write(chr(
        (just_switched << 7) |
```

```
        (hall_effect << 6) |
        (pd_temp >> 4)))
    ser.write(chr(
        ((pd_temp << 4) & 0xff) |
        (led_temp >> 8)))
    ser.write(chr(
        led_temp & 0xff))

    d = Data()
    read_packet(ser, d)

    assert d.count == count
    assert d.just_switched == just_switched
    assert d.hall_effect == hall_effect
    assert d.pd_temp == pd_temp
    assert d.led_temp == led_temp


if __name__ == '__main__':
    #test_read_packet()
    _main()
```

APPENDIX B – ARDUINO CODE – whole_state.c

```
#include <avr/io.h>
#define UART_N  0
#include "fast_serial.h"

volatile uint8_t save_new_state;  // boolean set if the serial just finished writing to the register
volatile uint8_t just_switched; // boolean that's set if in the last interrupt we shifted out new values
```

```
struct State {
   uint8_t shift_to_tlcs[4]; // 2x 16 bits
   uint8_t shift_to_pds[3];  // 3x 8 bits
} current_state, new_state;

ISR(half_second_interrupt) {
   uint16_t count = TCNTn;
   TCNTn = 0;
   sei(); // we don't want to lose serial characters by blocking interrupts too long

   // get temperatures
   uint16_t pd_temp = get_pd_temp(); // only 10 bits
   uint16_t led_temp = get_led_temp(); // only 10 bits

   // report the count/state/temps for the last period
   serial_write8('c');  // 'c' for count
   serial_write8(count >> 8);
   serial_write8(count);

   // pack the just_switched/temp data:
   // first byte:  bit 7 = just_switched, bit [7..0] = bits [11-4] of pd_temp
   // second byte: bit [7-4] = bits [3-0] of pd_temp, bit [3-0] = bits [11-8] of led_temp
   // third byte:  bit [7-0] = bits [7-0] of led_temp
   // on the computer, this is:
   // bytes = map(ord, ser.read(3))
   // just_switched = bytes[0] & 0x80
   // pd_temp = ((bytes[0] & 0x7f) << 4) | ((bytes[1] & 0xf0) >> 4)
   // led_temp = ((bytes[1] & 0x0f) << 8) | bytes[2]
   serial_write8((just_switched << 7) | (pd_temp >> 4));
   serial_write8((pd_temp << 4) | (led_temp >> 8));
   serial_write8(led_temp);

   serial_write8(current_state.shift_to_tlcs[0]);
   // ... etc, (send the entire current state)
   serial_write8(current_state.shift_to_pds[2]);

   just_switched = 0;

   if (save_new_state) {

      just_switched = 1;

      led_shift8(new_state.shift_to_tlcs[0]);
      led_shift8(new_state.shift_to_tlcs[1]);
      led_shift8(new_state.shift_to_tlcs[2]);
      led_shift8(new_state.shift_to_tlcs[3]);
      led_latch();

      pd_shift8(new_state.shift_to_pds[0]);
      pd_shift8(new_state.shift_to_pds[1]);
      pd_shift8(new_state.shift_to_pds[2]);
      pd_latch();

      current_state = new_state;

      save_new_state = 0;
   }
}
```

```
void loop() {
   if (serial.available()) {
      // ...
      // 'n' + 7 bytes for new state
      save_new_state = 0; // if two set states in the same period, overwrite the old
      new_state.shift_to_tlcs[0] = serial.read();
      // ...
      new_state.shift_to_pds[2] = serial.read();
      save_new_state = 1;
```

APPENDIX C – Arduino Code – Pins.h

```
#define PIN_HIGH(pin)  pin##_PORT |=  _BV(pin)
#define PIN_LOW(pin)   pin##_PORT &= ~_BV(pin)
#define PULSE_PIN(pin) pin##_PORT |= _BV(pin); pin##_PORT &= ~_BV(pin)

#define PIN_AS_INPUT(pin)  pin##_DDR &= ~_BV(pin)
```

```
#define PIN_AS_OUTPUT(pin) pin##_DDR |= _BV(pin)

// external interrupt
#define EXT_INT      PD0   // 21
#define EXT_INT_PORT  PORTD
#define EXT_INT_DDR   DDRD

#define TLC_LE       PB0   // 53
#define TLC_LE_PORT   PORTB
#define TLC_LE_DDR    DDRB

#define TLC_OE       PL0   // 49
#define TLC_OE_PORT   PORTL
#define TLC_OE_DDR    DDRL

#define TLC_SDI      PB2 // 51
#define TLC_SDI_PORT  PORTB
#define TLC_SDI_DDR   DDRB

#define TLC_CLK      PB1  // 52
#define TLC_CLK_PORT  PORTB
#define TLC_CLK_DDR   DDRB

#define PHOTOD       PL2  // 47
#define PHOTOD_PORT   PORTL
#define PHOTOD_DDR    DDRL

// Defines for Bit Banging
#define BB_DATA0      PA0  //22
#define BB_DATA1      PA1  //23
#define BB_DATA2      PA2  //24
#define BB_LE0        PA3  //25
#define BB_LE1        PA4  //26
#define BB_LE2        PA5  //27
#define BB_CLK0       PA6  //28
#define BB_CLK1       PA7  //29

#define BB_DATA0_PORT  PORTA
#define BB_DATA0_DDR   DDRA
#define BB_DATA1_PORT  PORTA
#define BB_DATA1_DDR   DDRA
#define BB_DATA2_PORT  PORTA
#define BB_DATA2_DDR   DDRA

#define BB_LE0_PORT    PORTA
#define BB_LE0_DDR     DDRA
#define BB_LE1_PORT    PORTA
#define BB_LE1_DDR     DDRA
#define BB_LE2_PORT    PORTA
#define BB_LE2_DDR     DDRA

#define BB_CLK0_PORT   PORTA
#define BB_CLK0_DDR    DDRA
#define BB_CLK1_PORT   PORTA
#define BB_CLK1_DDR    DDRA

// Had to do CLK2 on a different port
#define BB_CLK2       PC7  //30
```

```
#define BB_CLK2_PORT   PORTC
#define BB_CLK2_DDR    DDRC

/* For MSPIM Configuration
   Decided to just bit bang
#define MSPIM_LE0     PA0  //54
#define MSPIM_LE1     PA1  //55
#define MSPIM_LE2     PA2  //56
#define MSPIM_LE_PORT PORTA
#define MSPIM_LE_DDR  DDRA
*/
```

APPENDIX D – Arduino Code – fast_serial.h

```
#if !defined(FAST_SERIAL_H)
```

```
#   define  FAST_SERIAL_H

#if !defined(BAUD)
#  error #define BAUD to the serial baud rate, eg #define BAUD 115200
#endif

// Alex Leone, 2009-11-19

#include <inttypes.h>

#define ENABLE_TX_INTERRUPT()  UCSR0B |=  _BV(UDRIE0)
#define DISABLE_TX_INTERRUPT() UCSR0B &= ~_BV(UDRIE0)


#define BLOCK_WITH_TIMEOUT_RETURN(func, bytes)    \
   do {                                  \
      uint16_t timeout = 0;                  \
      while (func() < bytes) {               \
         if (++timeout == 0) {               \
            return;                          \
         }                                   \
      }                                      \
   } while (0)

uint8_t rx_buffer[256];
volatile uint8_t rx_front;
uint8_t rx_back;

uint8_t tx_buffer[256];
volatile uint8_t tx_front;
volatile uint8_t tx_back;

ISR(USART_UDRE_vect) {
   if (tx_front == tx_back) {
      DISABLE_TX_INTERRUPT();
   } else {
      UDR0 = tx_buffer[tx_back++];
   }
}

ISR(USART_RX_vect) {
   uint8_t c = UDR0;
   rx_buffer[rx_front++] = c;
}

static inline uint8_t serial_available() {
   return rx_front - rx_back;
}

/** Undefined behavior if there's nothing in the buffer. */
static inline uint8_t serial_read() {
   return rx_buffer[rx_back++];
}

/** Undefined behavior if the buffer is full. */
static inline void serial_write(const uint8_t c) {
   tx_buffer[tx_front++] = c;
   ENABLE_TX_INTERRUPT();
```

```
}

static void serial_print(char *s) {
   char c;
   while ((c = *s++) != '\0') {
      tx_buffer[tx_front++] = c;
   }
   ENABLE_TX_INTERRUPT();
}

/** Requires BAUD to be set.  See setbaud.h. */
static inline void serial_init() {
#include <util/setbaud.h>
   UBRR0 = UBRR_VALUE;
#if USE_2X
   UCSR0A |= _BV(U2X0);
#else
   UCSR0A &= ~_BV(U2X0);
#endif
   UCSR0B = _BV(RXCIE0) // enable RX interrupt
         | _BV(RXEN0) | _BV(TXEN0); // enable rx and tx
   UCSR0C = _BV(UCSZ01) | _BV(UCSZ00); // 8-bit data, no pairity, one stop bit
   sei();
}



#endif /* !defined(FAST_SERIAL_H) */
```

APPENDIX E – Photos

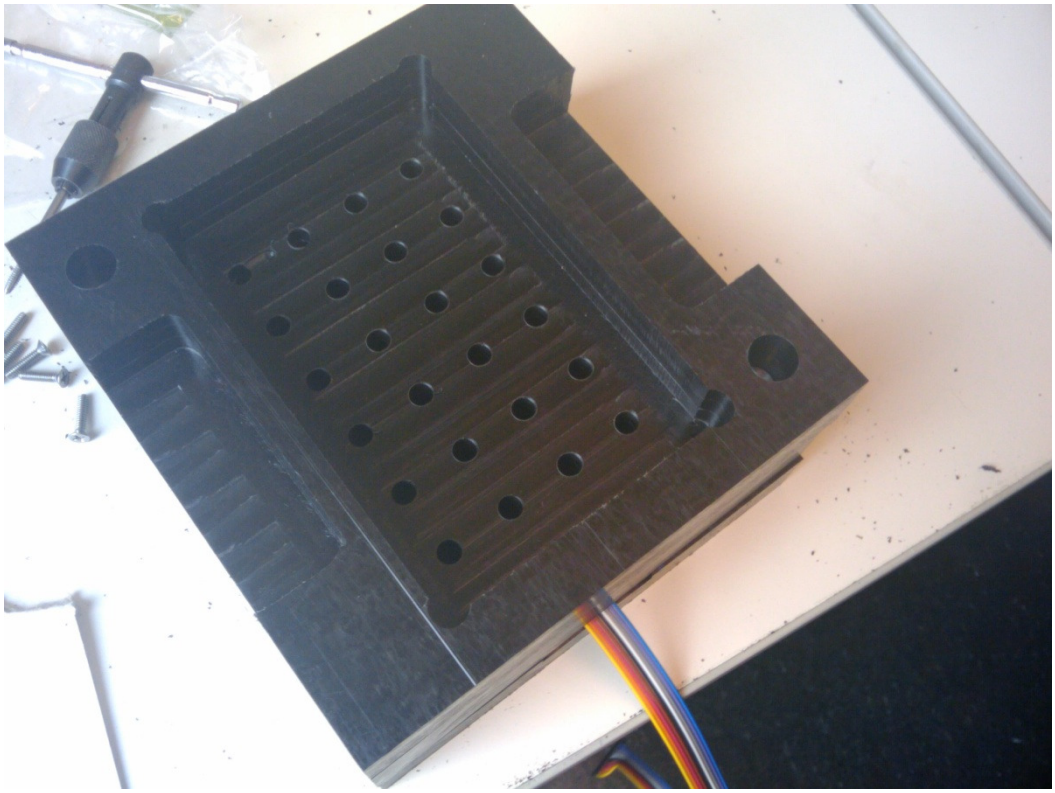


Figure 1 - Lid of Enclosure with Holes for Photodiodes

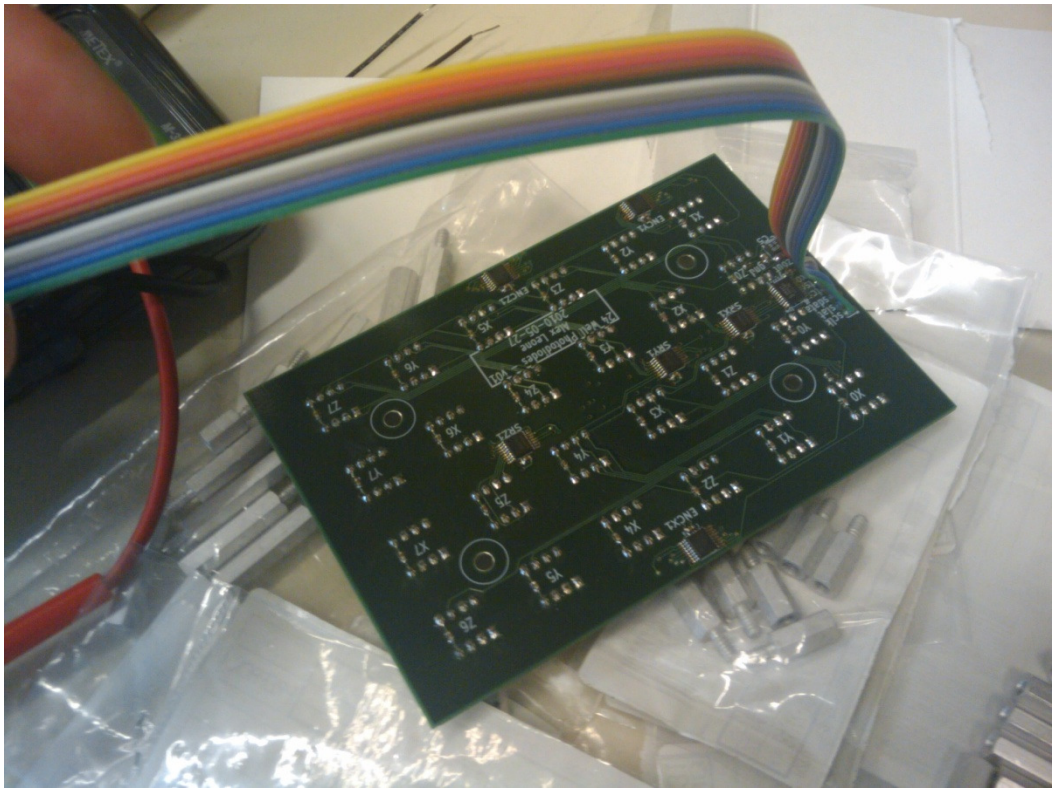


Figure 2 - Back of the Photodiode Board

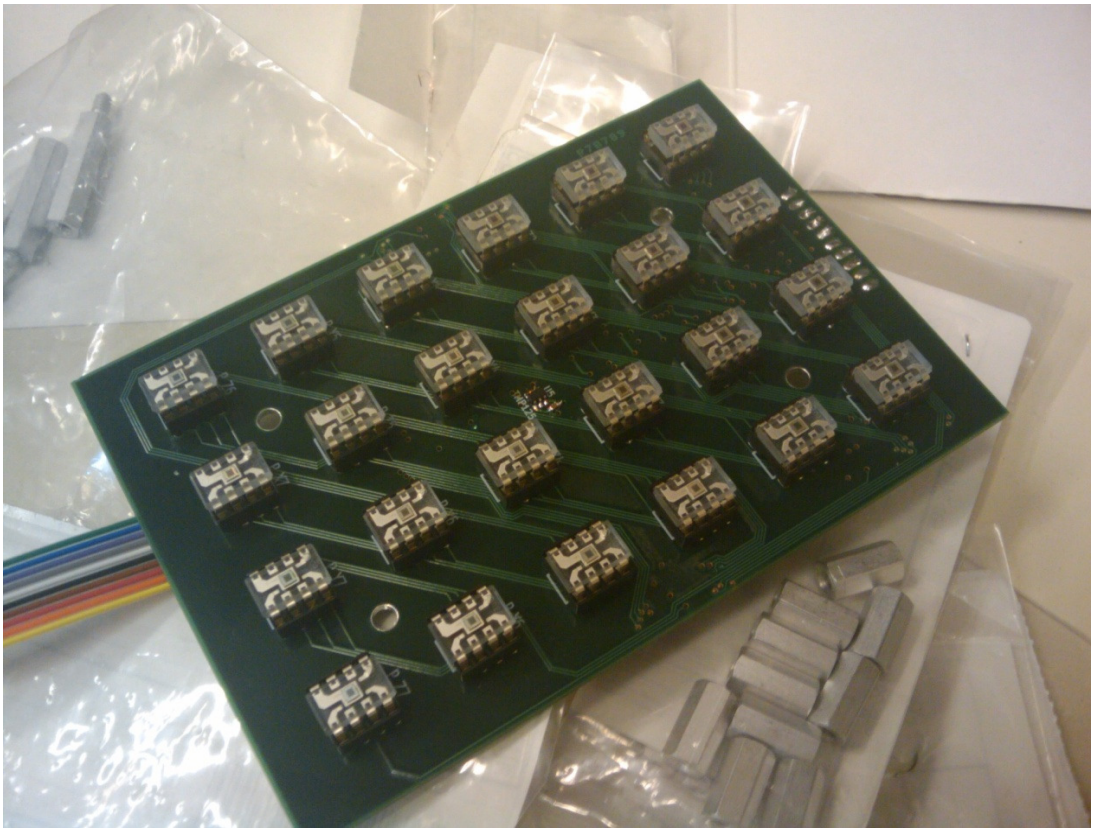
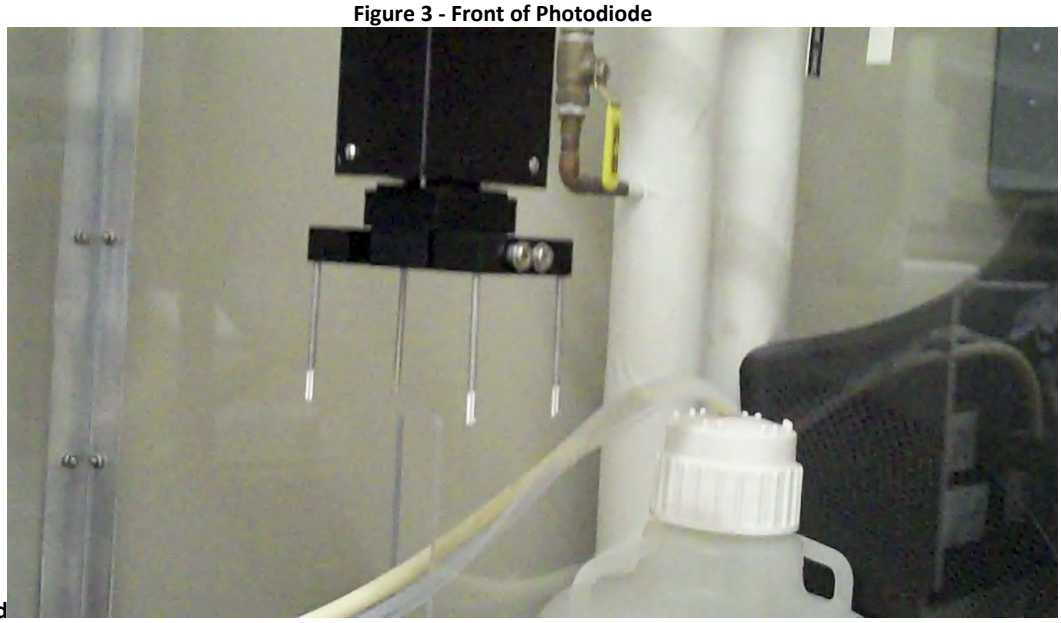


Figure 3 - Front of Photodiode



Board

Figure 4 - Gripper on Liquid Handler

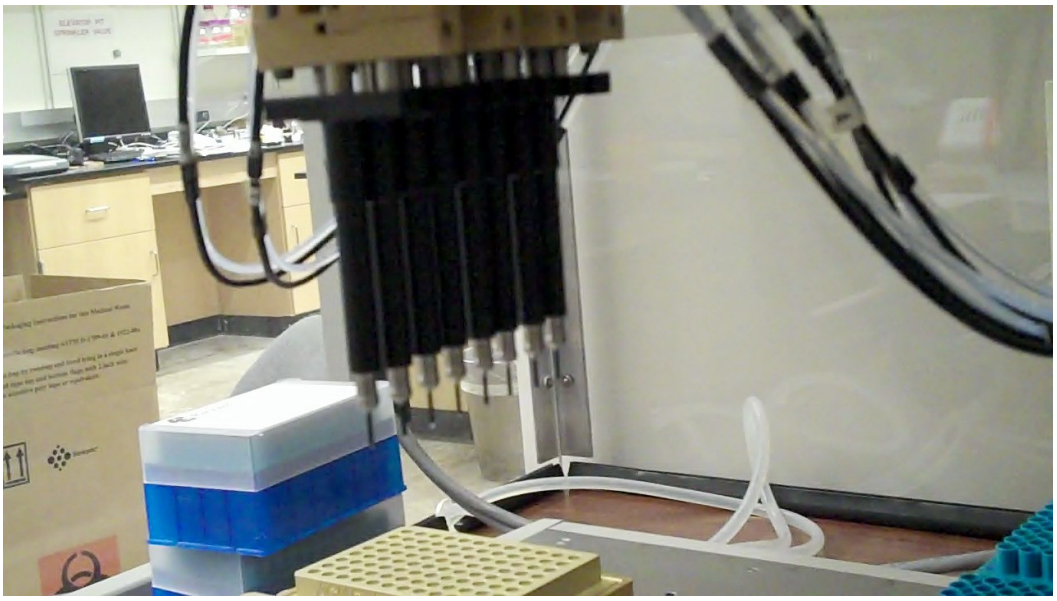


Figure 5 - Pipettes for the Liquid Handler

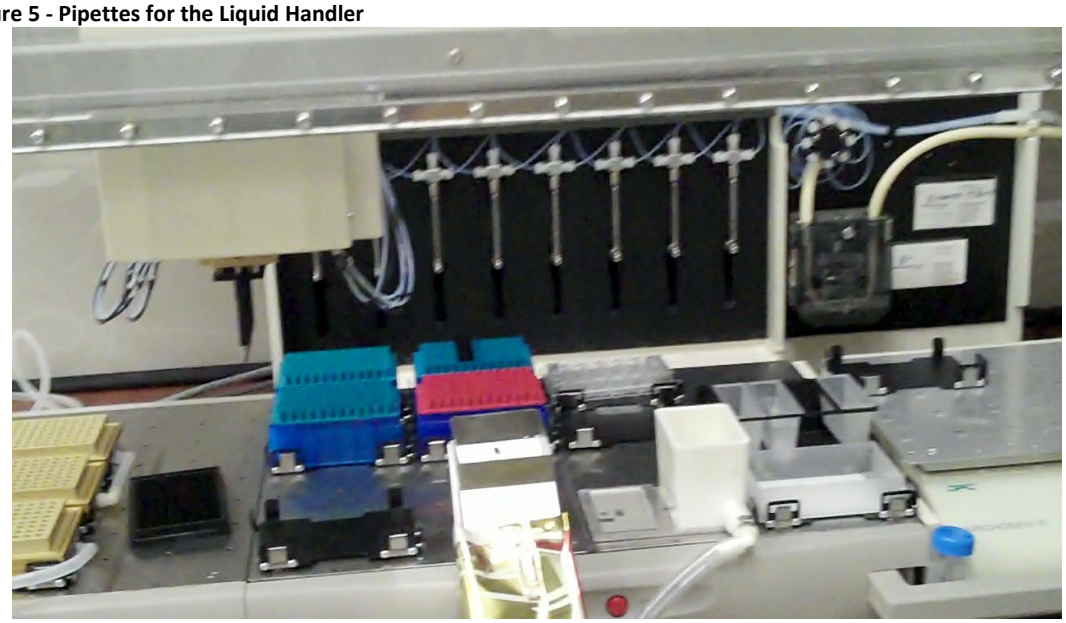


Figure 6 - Liquid Handler Arrangement

Figure 7 - LED Shining Through Plate

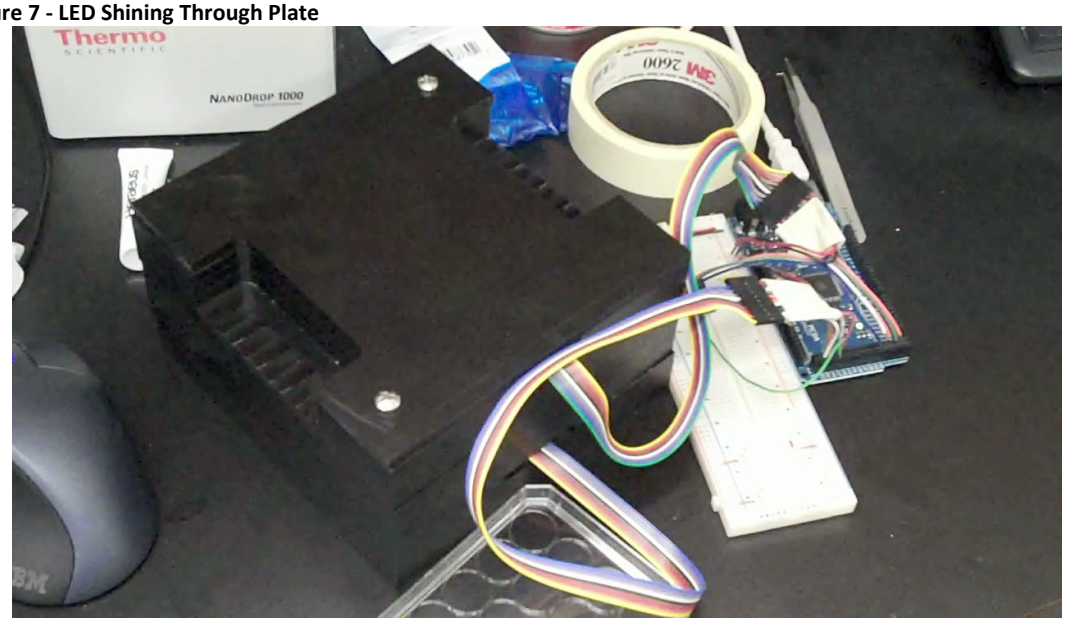


Figure 8 - View of Enclosure of Turbidostat

APPENDIX F: Parts Cost

Part	Price	Quantity	Total
Light to Frequency Converter (photodiode)	$5.36	30	$160.80
Arduino Mega	$64.95	1	$64.95
5mm 598nm Amber LEDs	$0.22	5	$1.12
5mm Orange LEDs	$0.22	5	$1.08
3mm 600nm Orange LEDs	$0.58	5	$2.90
Full-Rotation Servo	$13.95	1	$13.95
USB Cable A to B - 6 Feet	$3.95	1	$3.95
Wall Adapter Power Supply - 9VDC 650mA	$5.95	1	$5.95
Delrin Sheets - Thickness 1	$52.94	2	$105.88
Delrin Sheets - Thickness 1.75	$87.60	1	$87.60
Delrin Sheets - Thickness 0.187	$20.00	1	$20.00
LED 5mm Amber Diffused	$0.11	35	$3.86
Shipping	$44.54	1	$44.54
Photodiode Board (4-Layer)	$66.00	1	$66.00
LED Board (1-Layer)	$33.00	1	$33.00
		Total=	$615.58