Lego Sorter

Alicia Bird, Electrical Engineering

Project Advisor: Dr. Mark Mitchell

April 23, 2019

Evansville, Indiana

Acknowledgements

I would like to thank the University of Evansville's EECS Department for funding this project. I would also like to thank and acknowledge Dr. Mark Mitchell (project advisor) for support and technical assistance in the project. Additionally, I want to thank Jeff Cron for technical support and 3D printing access.

Table of Contents

List of Figures

List of Tables

Introduction

For this project, the problem presented is the undesirable amount time and manual labor required to sort Lego's. To provide an alternative option to manual sorting, a machine was created that included an organization system to retrieve, detect, and place Lego bricks.

While the lofty goals of this project were not entirely achieved, the sorter is able separate Lego pieces individually and detect color before moving a single piece to an appropriately determined bin.

Background

Lego's are a popular manufactured toy consisting of interlocking plastic bricks. These bricks come various shapes, sizes, and colors. Since 1958, there have been 400 billion Lego bricks produced. Approximately 19 billion Lego elements are produced per year, 2.16 million are molded every hour, 36,000 every minute [1].

Now, with this many Lego bricks floating around, organizing quickly becomes a hassle. A possible solution to such a problem is a Lego Sorter. This solution receives many unsorted Lego bricks and processes them into "bins" of like color. This device is a considerable time saver, taking on the lengthy process of sorting. Additionally, the pieces are well organized upon completion into easily accessible bins, convenient for next project use.

To achieve such a solution, certain steps are taken in the design process to overcome the various challenges. The objective for the sorter is to differentiate Lego's of different types. To start, the process involves separating singular pieces from a large pile using mechanical aspects such as gear motors and an actuator. The next task is to obtain piece information. To do so, a color sensor scans the brick and the RGB values are evaluated. Finally, based on the information received, a brick is moved and deposited to a determined bin.

The minimum design requirements outlined from previous planning were to complete a sorter that can differentiate between either color or size, depending on user input. It should receive a large pile of LEGO's to sort and outputs sorted LEGO's to separate, organized by likeness containers. For this iteration of the project, sorting by size remains unaccomplished and can be improved upon in the future.

Design Approach

Hardware Design

The designed solution to producing a Lego Sorter uses three major phases. These are: separate singular pieces from a large pile, obtain brick information, and deposit brick into the best category bin. Each phase will be broken down and the implementation explained. An overall photo of the completed Lego Sorter is provided in Figure 1.

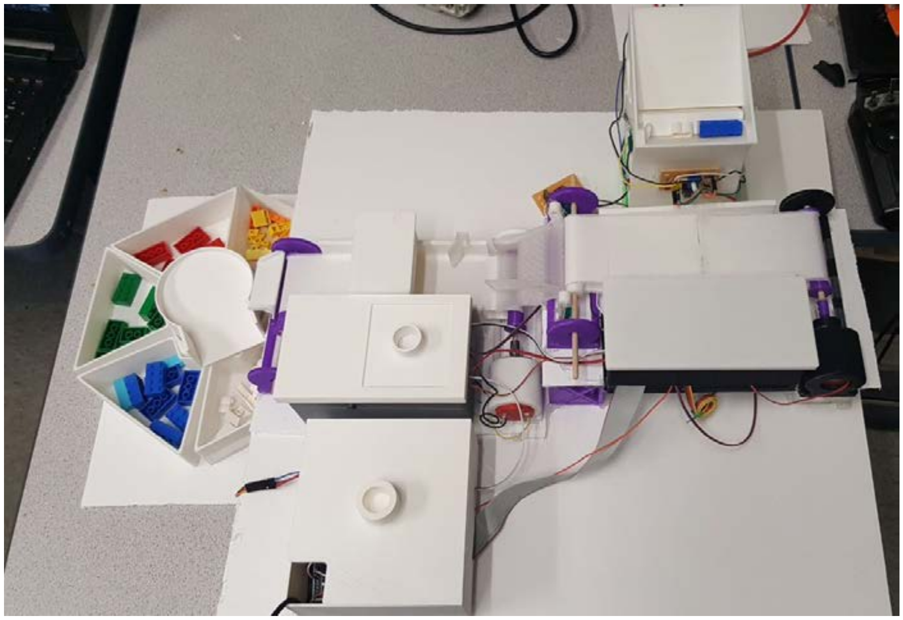


Figure 1: System Photo

Separate Pieces

To separate bricks, a mechanical system is used. This mechanical system consists of three parts. The first is a container which houses unsorted Lego bricks. Here, Lego bricks await sorting. It is crucial few bricks be taken out of the container at a time. This process is carried out by an “elevator” that moves up and down along a slope of the container.

The following explanation works concurrently with Figure 2, an early design photo. The elevator, labeled as 2, consists of a thin piece of 3D printed material that acts as a “shelf”. This shelf can carry just a few Lego bricks to the top of the bin, depositing them on to a waiting area below. Because of the smallness of the shelf, the elevator can only move a few pieces at a time.

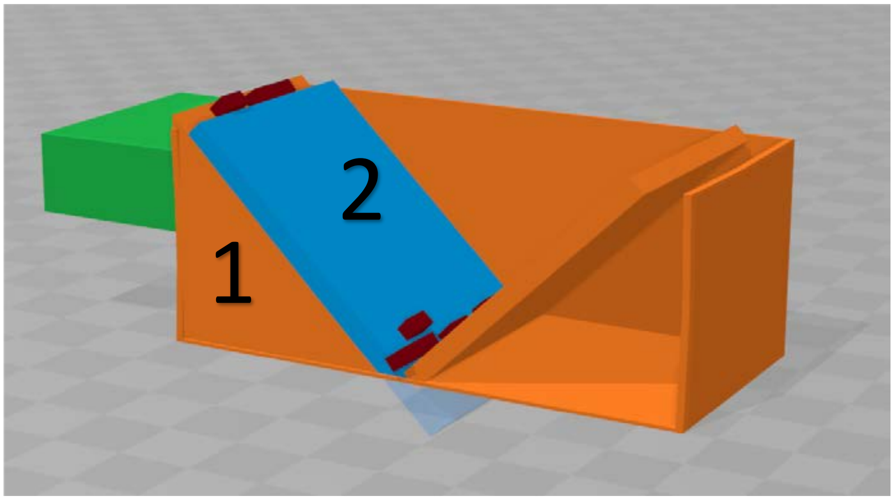


Figure 2: Bin Housing Unsorted Lego's and Elevator.

The shelf, as modeled in Figure 2 (labeled with 1) and in Figure 3, is a few millimeters larger than the standard Lego. Additionally, the container that holds unsorted Lego's is shown in Figure 4. The box has been designed in a way that allows the elevator to move smoothly up and down to transport Lego's.

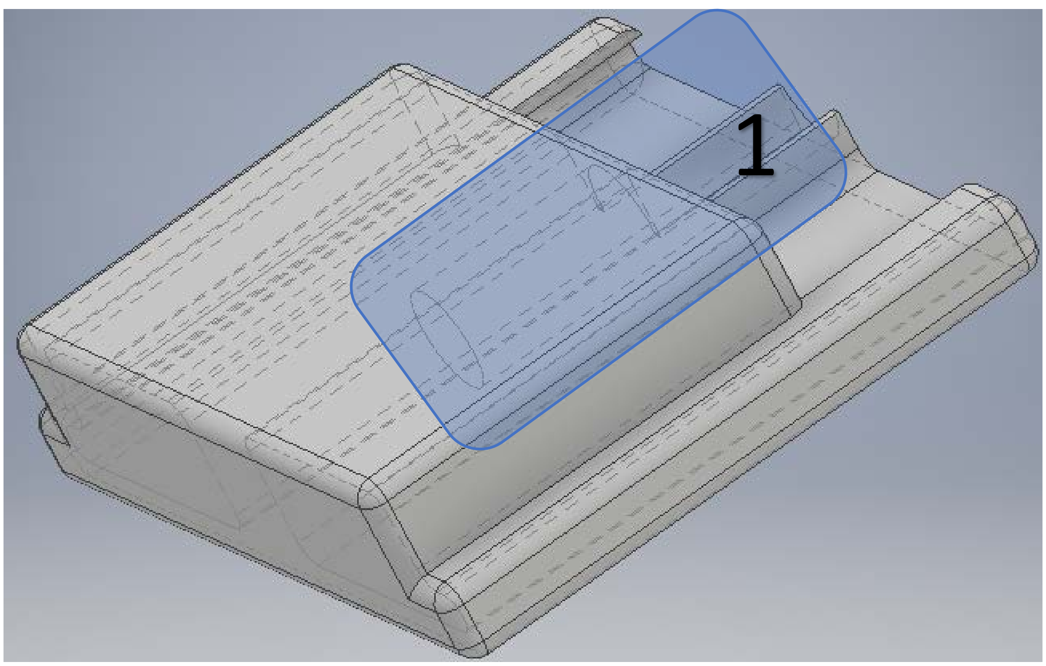
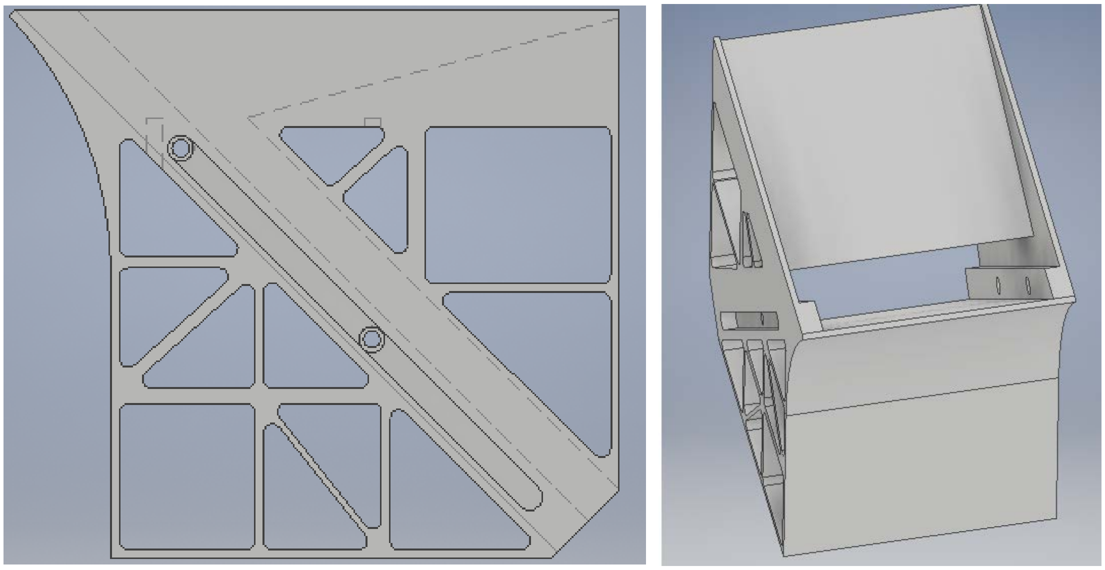


Figure 3: Elevator



(a) (b)

Figure 4: Holding Box Side View and Top View

The box in Figure 4 has a sloped bottom to allow more pieces to slide on to the shelf and be pushed up, when the elevator returns to the bottom.

Moving to mechanical elements, the elevator is controlled by an actuator. The bottom of the elevator allows the moving pin of the actuator to sit inside it. See label 1 in Figure 3 for a model of this. The actuator can then slide the elevator up and down the sloped surface seen in Figure 4's Holding Box.

The actuator is electrically controlled by two pins. In the software section, it is elaborated on how the pins change. Electrically, the actuator it simply connected as in Figure 5. The actuator itself was received from Jeff Cron and has seen a fair bit of use. As a result, the model number has worn off. A best guess determines this model as an RC Linear Actuator L-12, which is shown in Figure 5b.

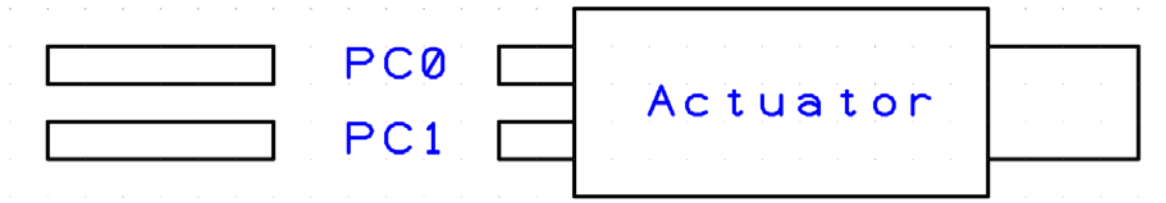


Figure 5: (a) Actuator Circuit Design

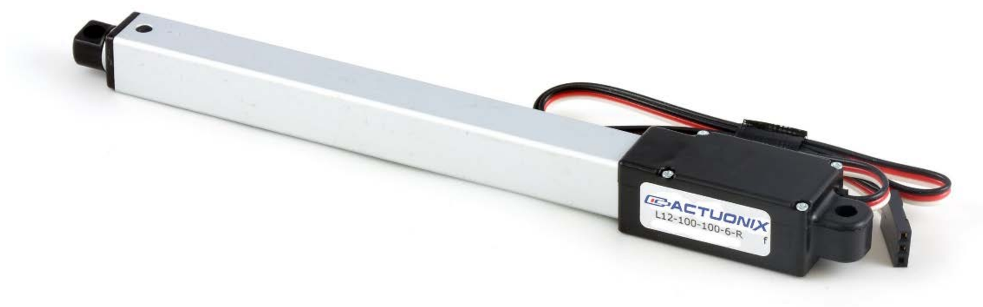


Figure 5: (b) Linear Actuator L12

On the box itself, there is another section of interest. This section is the inclusion of circular holes modeled into the box. These are spaces specifically made to hold two sets of infrared (IR) sensors and LED's, as can be seen in Figure 6. These sensors are connected to GPIO pins on the ARM board. Based on the position of the elevator, these sensors are triggered. More is said on this in Software Design.

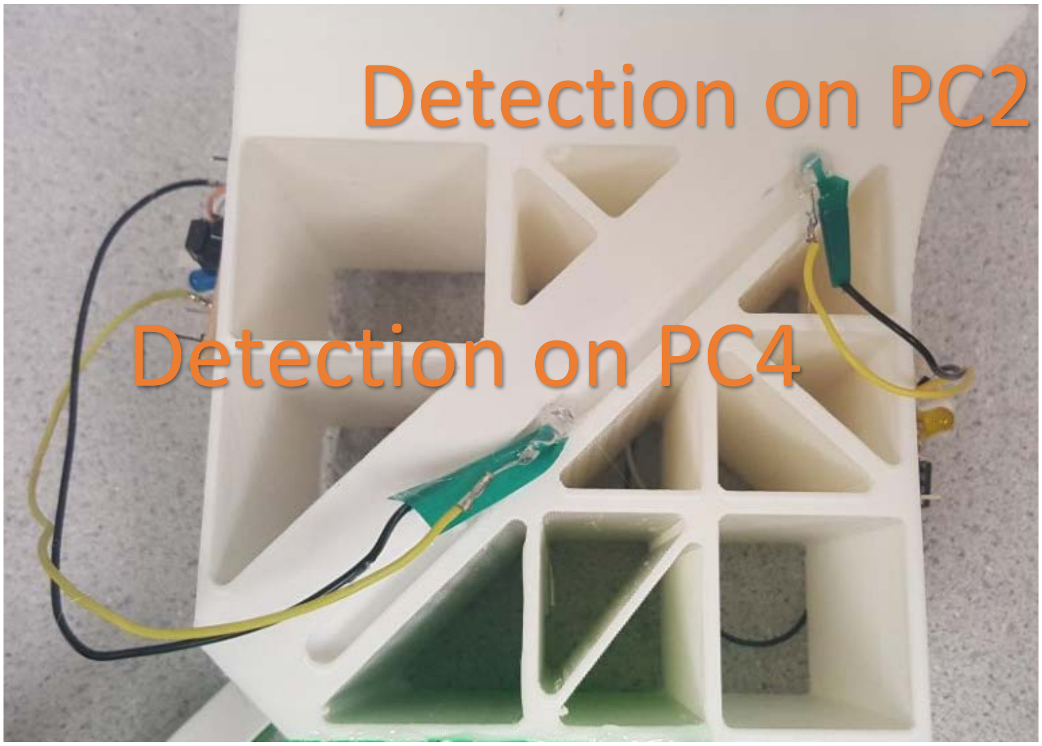


Figure 6: IR Sensors Fitted in to Modeled Slots

When an object is blocking the IR sensor, voltage at the collector drops to ground. Unfortunately, the IR sensor does not have the drive to force a GPIO pin on the ARM board to ground. As a result, to properly sense voltage, a voltage comparator, checks the collector voltage against a threshold created by a voltage divider [2]. If the collector voltage increases above the threshold, the sensor is therefore blocked. As a result, the voltage comparator pulls the GPIO pin to ground. A general diagram of these connections can be seen in Figure 8. Because this IR-voltage comparator circuit is used several times for this project, a general diagram is present. The pins connected to the elevator are PC2 and PC4. PC2 is configured to signal whether the elevator has surpassed the dumping height. If this is true, the actuator retracts and the elevator reverses. PC4 signals if the elevator is at “ground level.” Once here, bricks slide onto the elevator and the actuator extends, sliding it up the ramp. The placement of IR sensors can be viewed in Figure 6 with pins labeled appropriately on the figure.

Additionally, the electrical detection system is shown in Figure 7(a). The pinout for the voltage comparator used, LM311, is provided in Figure 7(b) [2].

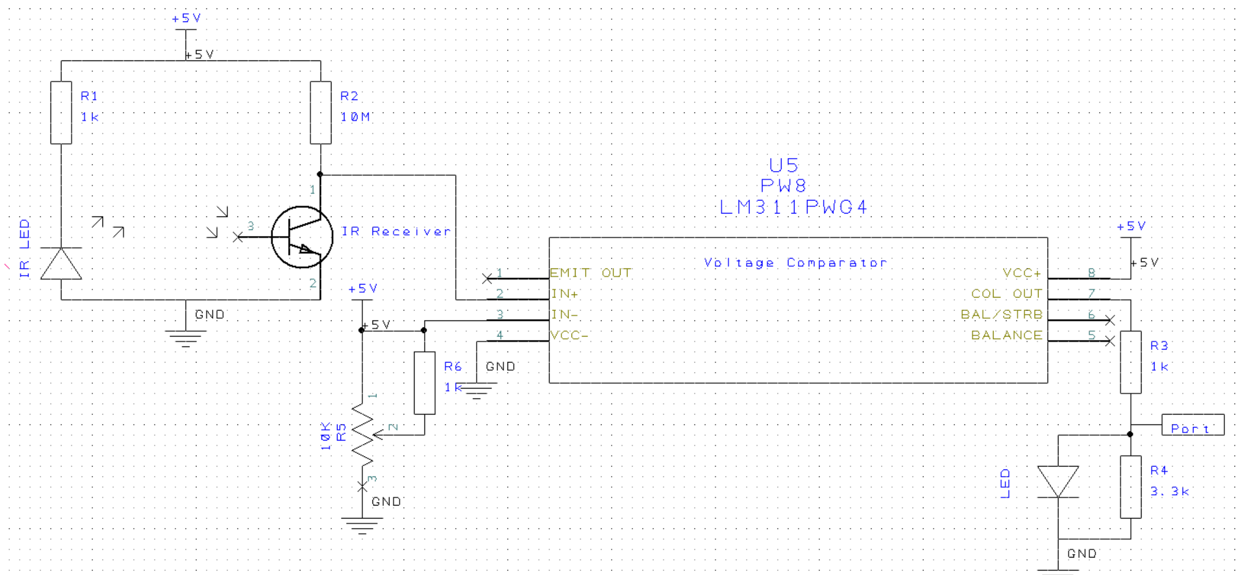


Figure 7: (a) Electrical Setup of IR LED, Receiver, and Voltage Comparator

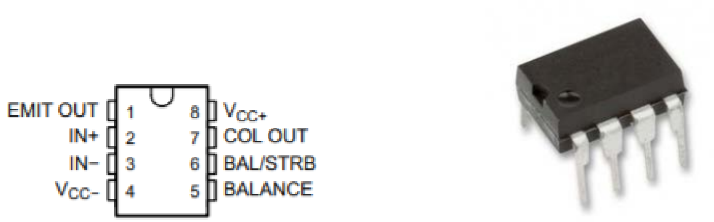


Figure 7: (b) LM311 Pinout and (c) Chip

Next in this system is the conveyor belts. To review, at this point Lego's have moved from the elevator element on to a conveyor belt system.

The system is made up of two conveyors, one moving at a very slow rate, while the other is considerably faster in comparison. The slower conveyor belt is the first of the two - this is where pieces land when falling from the elevator, as can be seen in the Figure 8 mock up as well as implemented in Figure 9.

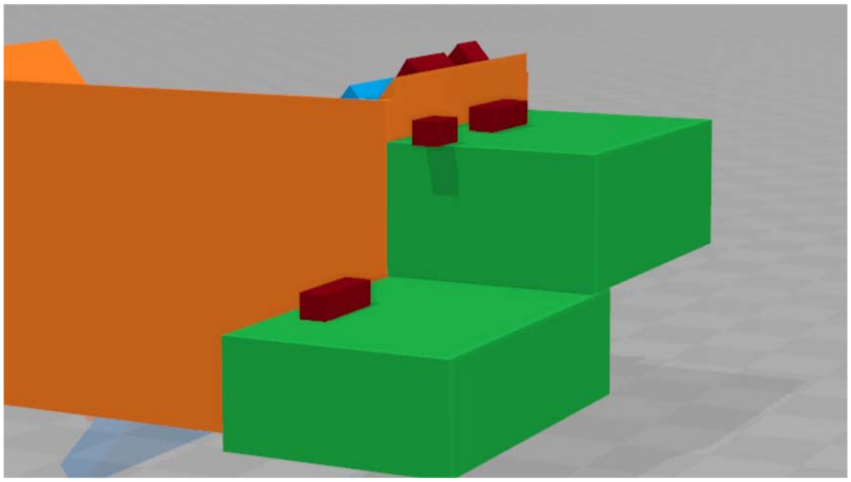


Figure 8: Conveyor belt system after the elevator

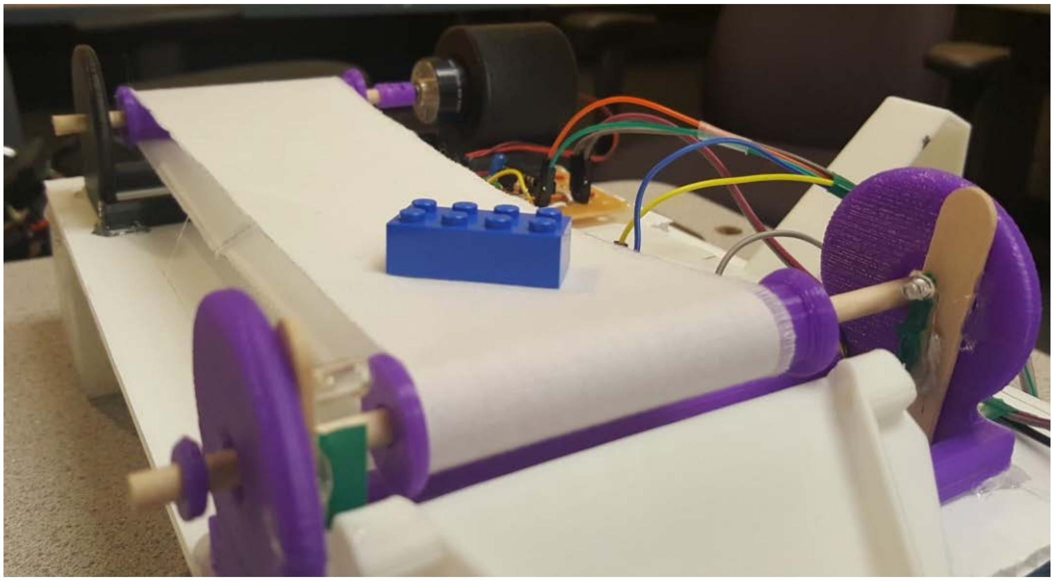


Figure 9: Slow conveyor belt

Following that, the pieces move forward to fall onto the second, faster conveyor belt, positioned under the second. During its fall, the piece triggers an IR sensor. This sensor signals 1) stop the slow conveyor 2) stop the elevator and 3) start the fast conveyor. This fast conveyor is present to help elongate the distance between pieces for better separation. This conveyor belt can be seen in Figure 10.

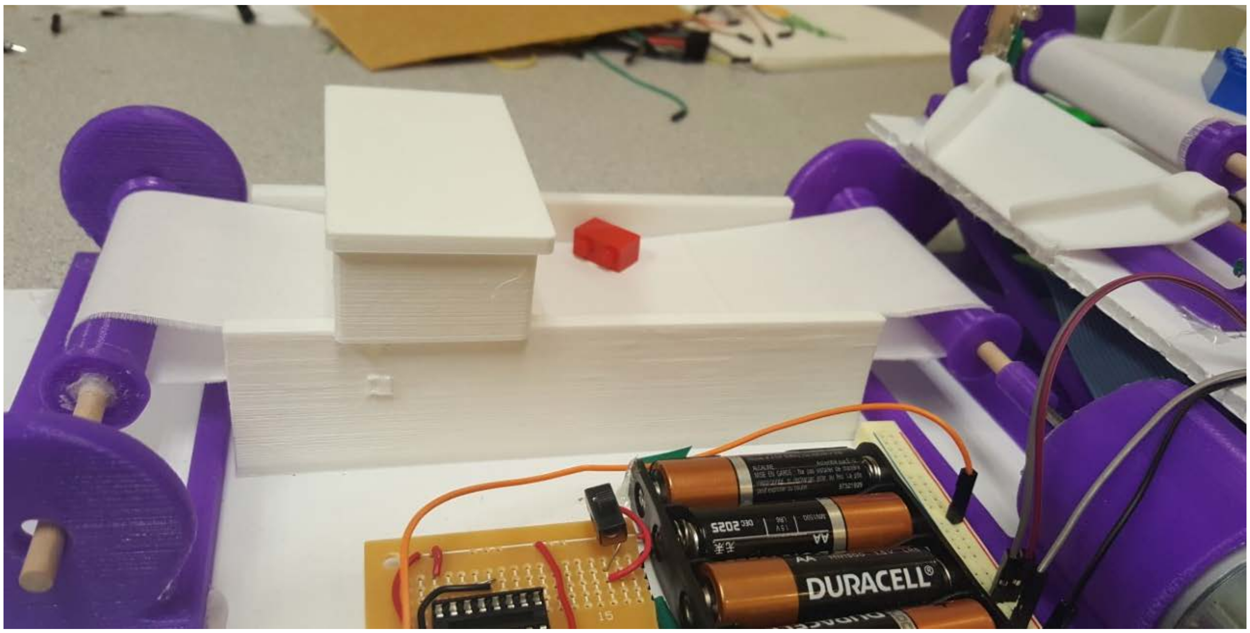


Figure 10: Fast conveyor belt

Both belts are controlled by the microcontroller and driven by an H-Bridge. An electrical schematic for each conveyor can be seen in Figure 11 and 12 respectively. The IC used is an SN754410NE and a pinout of the package is included in Figure 13 [3].

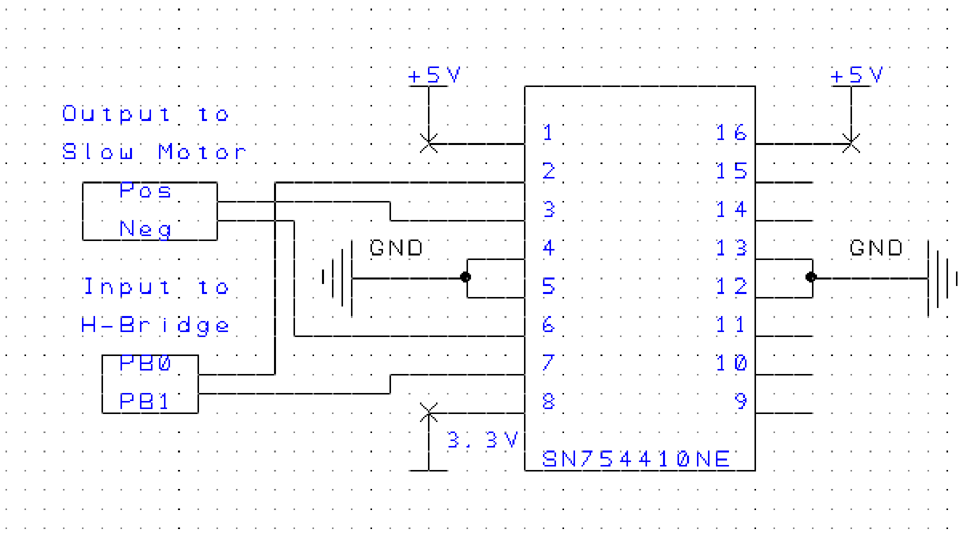


Figure 11: H-Bridge for Slow Conveyor

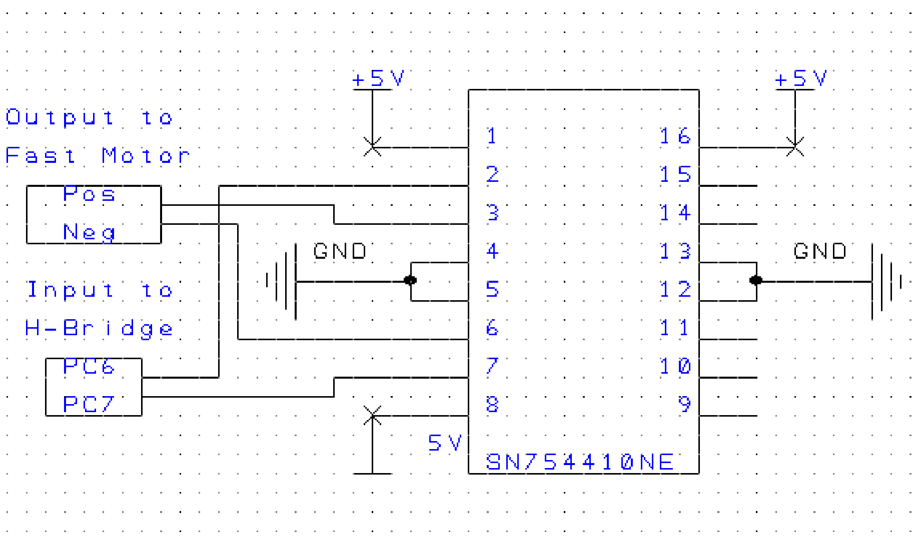
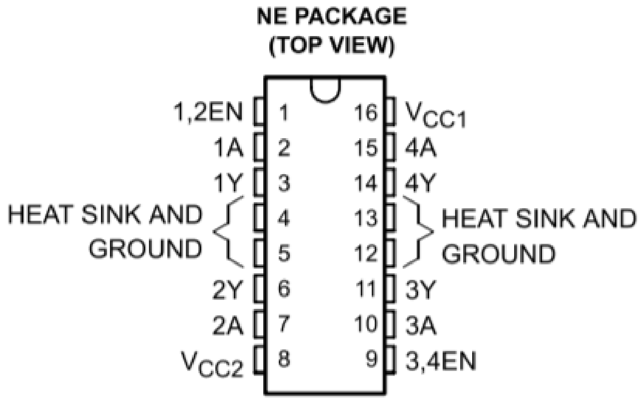


Figure 12: H-Bridge for Fast Conveyor



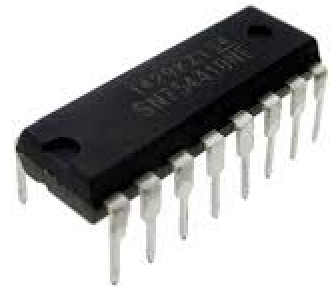


Figure 13: (a) SN754410NE Pinout and (b) Chip

Gather Information

The next design aspect is to gather information from the brick. The sorter observes color to decide how to differentiate between bricks. Finding the color is accomplished with RGB color sensor TCS3472, a color sensor sold by Adafruit [4]. It is shown in Figure 14. This data is sent to the microcontroller through I2C [4]. Setting color ranges from testing determines what category the piece sits in. This is elaborated upon in the Software Design section.

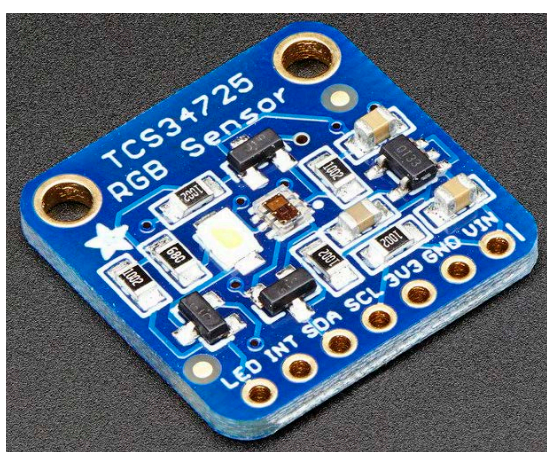


Figure 14: TCS3472 Color Sensor

The placement of the color sensor is extremely important. The TCS sensor requires closeness to an object. To accomplish this, the setup for this color sensor utilizes the second conveyor belt as well as an IR trigger. The color sensor is placed in a box as modeled in Figure 14. A hole in the bottom of the box allows sight onto the conveyor belt underneath. When the IR sensors under the box have been triggered, indicating a piece is below the box, the conveyor belt stops and allows time for the color sensor to read the brick. Once finished, the conveyor belt continues and ejects the brick from the system. This setup is pictured in Figure 15.

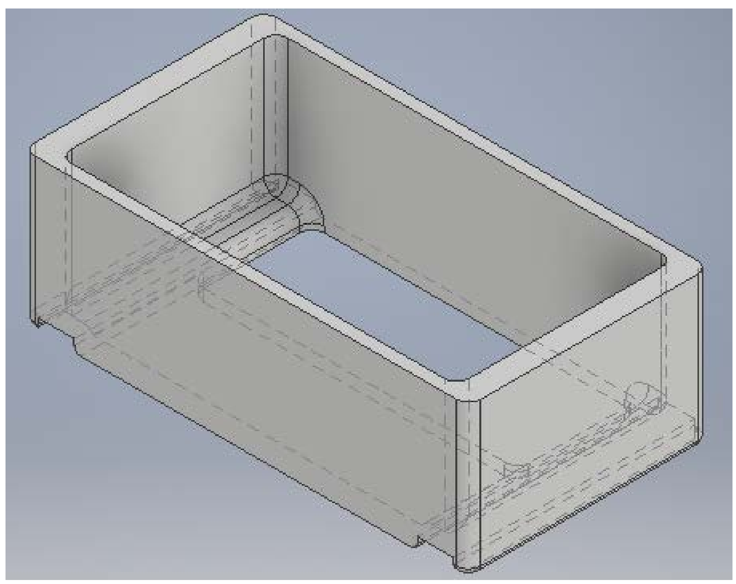


Figure 15: (a) Color Sensor Box

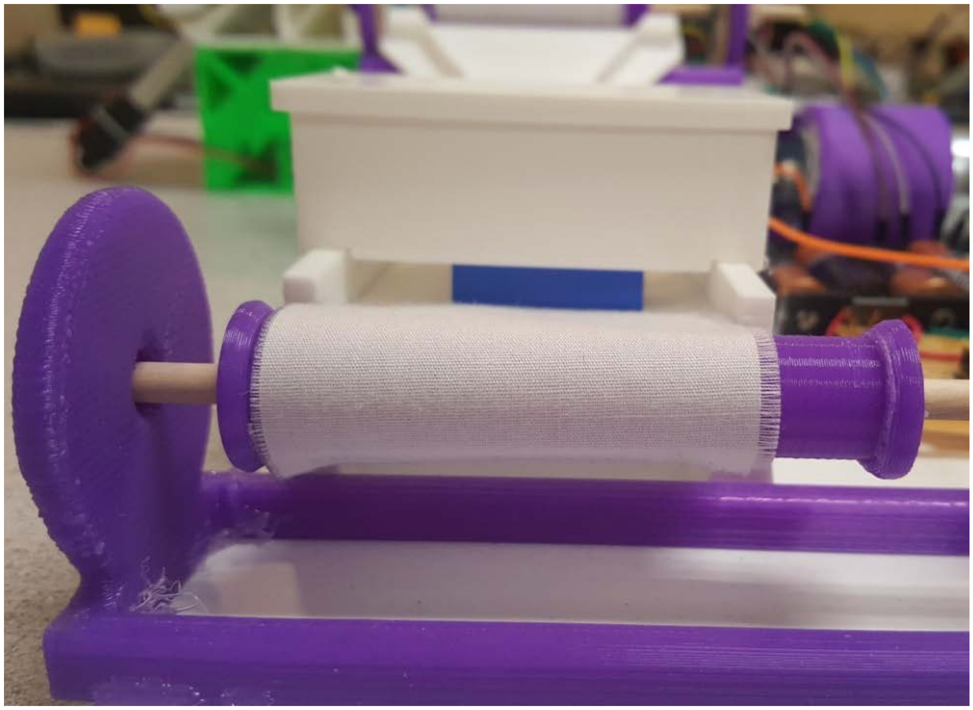


Figure 15: (b) Color Sensor Box and Conveyor Implemented

Deposit

The final design aspect is moving the piece into a selected bin, based on input from the information system. This starts by moving a “slide.” Figure 16 shows an early mockup of the system. As shown, the slide, labeled as 1, lines up with a bin, labeled 2. This slide is controlled by a MG996R servo. Activation and use are described in Software Design. An image and electrical hookup are provided in Figures 17 and 18 respectively. Once the servo’s slide is in place, the fast conveyor belt clicks on and ejects the brick onto the slide. The brick now has the momentum to continue down to sloped slide and into a designated bin

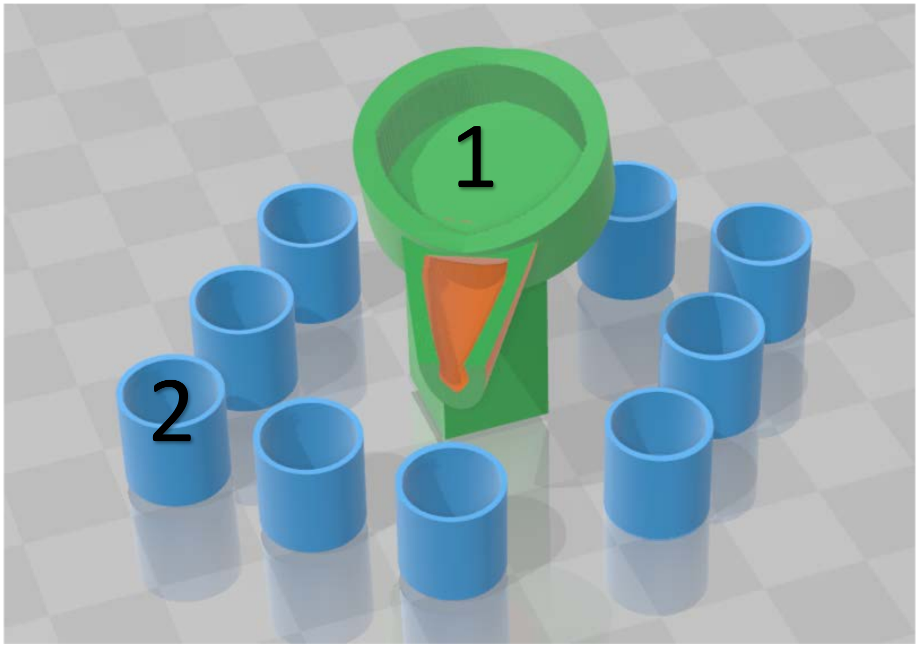


Figure 16: Tunnel and Bins

Figure 17: MG996R Servo

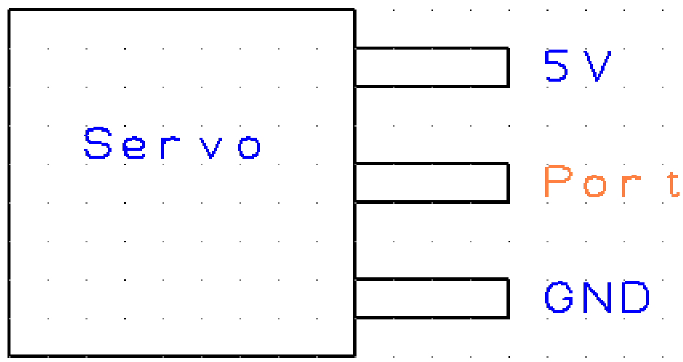


Figure 18: Schematic of Servo

The three main design phases have been covered. The remaining electrical design will now be covered. All operations are run through the ATMEL Microcontroller [5]. For the supplying power, the microcontroller uses the 5V power supplied by USB [5]. The fast gear motor and H-Bridge IC's are supplied by a 6V battery connection.

Software Design

These aspects will be covered in much of the same order as in Hardware Design.

Separate Pieces

The actuator has three states for movement and halting. It uses PC6 and PC7 to adjust between reverse, forward, and stop [5]. Code for the three states is in Figure 19.

```
void actuator_reverse(void)
{
  GPIOC_ODR &= ~(1);       //  switch PC0 to 0
  GPIOC_ODR |= 1 << 1;     //  switch PC1 to 1
}

void actuator_forward(void)
{
  GPIOC_ODR |= 1;            //  switch PC0 to 1
  GPIOC_ODR &= ~(1 << 1);    //  switch PC1 to 0
}

void actuator_stop(void)
{
    GPIOC_ODR |= 1 << 1;  //PC0 to 1
    GPIOC_ODR |= 1;       //PC1 to 1
}
```

Figure 19: Actuator States

To know what state the actuator should transition to, the IR sensors on PC2 and PC4 are used. These ports are setup as interrupts [5].

These ports are not identical, however. As shown in Figure 20, PC2 looks across a higher side of the ramp. As a result, it is almost always blocked by the elevator. The only time PC2 pushes high is when the sensor is unblocked. Therefore, the interrupt connected to the pin is configured as a rising edge [5]. When the interrupt is triggered, the elevator has been pushed above the sensor. At this point, the elevator has pushed Lego's out of the box and may now reverse.

PC4 works in the opposite way and thus is configured as a falling edge [5]. It is almost entirely unblocked by the elevator. The only time the sensor pulls low is when the elevator has been pulled down far enough to let more Lego's on to the shelf and block the IR sensor. The elevator then switches states and moves forward.

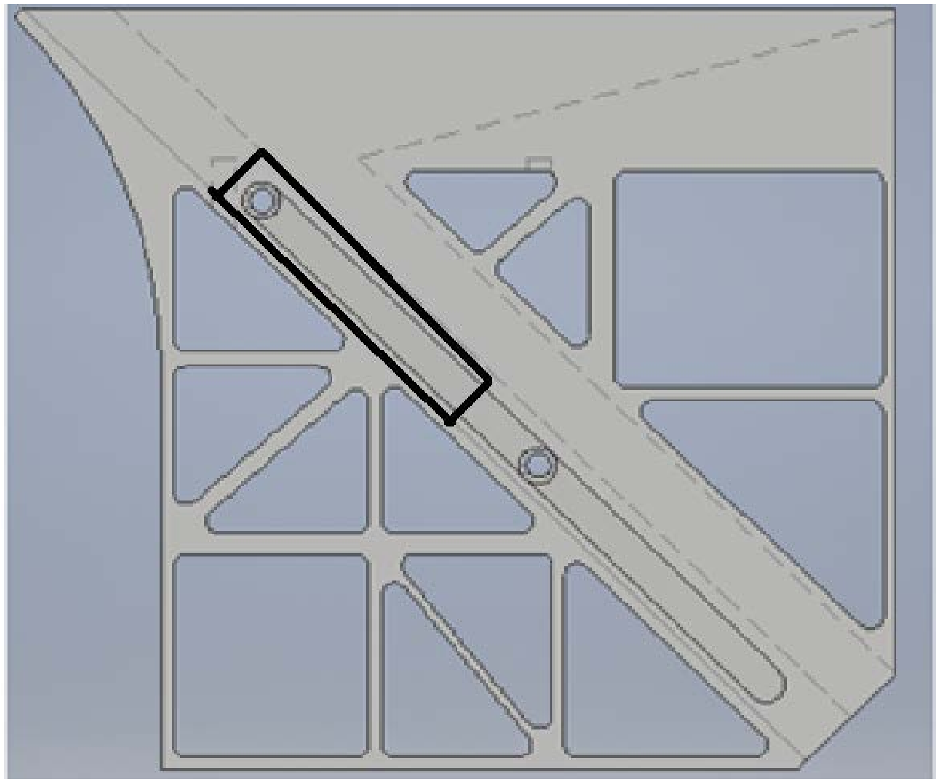


Figure 20: Elevator versus IR Placement

The next software design to break down is that controlling the motors of the conveyors. Figure 21 outlines the functions used to start and stop both motors. The pins are connected to an H-Bridge for each motor, which enables this toggling. As for implementation, the motors start and stop to help separate bricks from each other, as well as allow for good placement under the color sensor. The second conveyor belt motor begins in a stopped state. Meanwhile, the slow motor runs while it awaits bricks from the elevator system. As bricks fall on to the conveyor system, they move along until the first brick triggers the IR system on the end. The IR system can be seen in Figure 7(a). This sub-system triggers on a calling edge [5]. IR placement is seen in Figure 22.

Once triggered, the IR system interrupt signals for the elevator and slow motor to stop. The interrupt also signals for the fast conveyor belt to begin movement.

```
void slow_motor_stop(void)
{
  //STOP: Both PB0 and PB1 are 1
  GPIOB_ODR |= 1;
  GPIOB_ODR |= 1 << 1;
}

void slow_motor_start(void)
{
  //START: Set PB0 to 1 and PB1 to 0
  GPIOB_ODR &= ~(1<<1);
}

void fast_motor_stop(void)
{
  //STOP: Both PB2 and PB3 are 1
  GPIOC_ODR |= 1 << 6;
  GPIOC_ODR |= 1 << 7;
}

void fast_motor_start(void)
{
  //START: Set PC7 to 1 and PC6 to 0
  GPIOC_ODR &= ~(1<<6);
}
```

Figure 21: Functions for Motor Movement

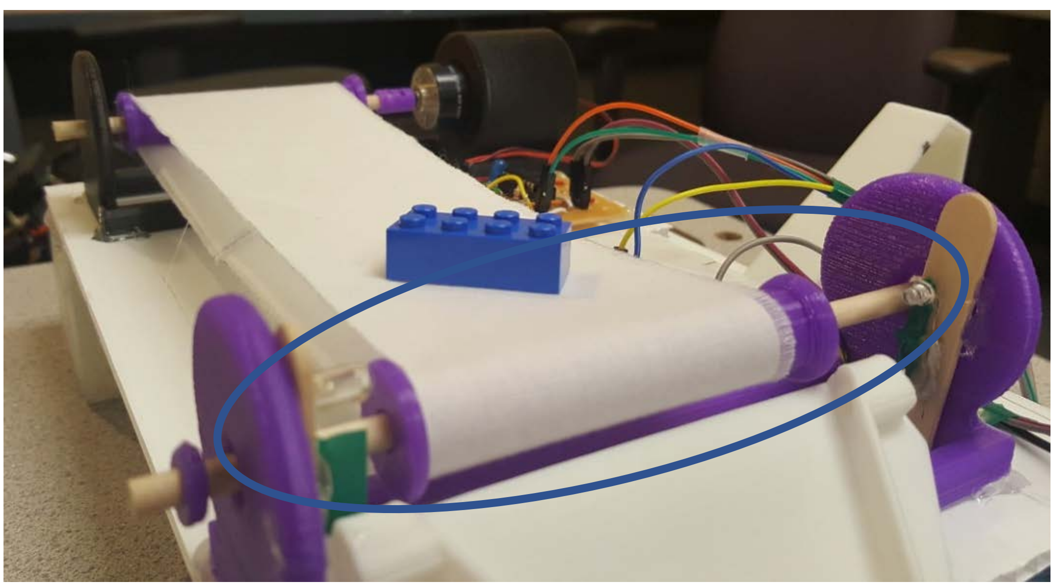


Figure 22: IR Placement on Conveyor Belt

The fast conveyor begins to move the singular piece. Once more, an IR sensor is placed here to signal the conveyor to stop directly under the color sensor. From here, the color sensor may scan the piece.

Gather Information

Two discussions are required for explanation of the color sensor. First the setup of to render the sensor usable and the other is interpreting that data. To initiate and read from the sensor, I2C protocol is required [4]. Figure 23 includes some pseudo-code involving setup and use of sensor.

```
I2C_Setup()
{
        Activate Pins
        Start Bit
        Acknowledge
        Send Slave Address
        Request Data
        Receive Data
}
```

Figure 23: I2C Protocol pseudo-code

Once the protocol has finished, is received in terms of high and low registers pertaining to RGB values [4]. RBG values usually are numbers ranging from 0 to 255 tucked into a triplet. The range from 0 to 255 indicates how saturated each color and through additive color mixing, the result is achieved. For example, the triplet (255, 0, 0) has 255 red, 0 green, and 0 blue. This color obviously red. However, a color such as (72, 66, 134) is blue.

It is difficult to account for every value of between 0 and 255 for 3 different variables. As a result, this machine uses approximations for each color by determining the percentage of each RGB variable in the result. The pseudo-code for this math can be seen in Figure 24.

```
{
        Determine maximum value from the three variables
        Divide each variable by the max
        Normalize to a 0 to 2 scale
}
```

Figure 24: Pseudo-Code for Approximating Color

The last important aspect of the color sensor is using these new values to determine the color. Table 1 details the results of extensive testing determining where values will lie once translated into the 0 to 2 scale. 0 indicates none of that color is present, 1 indicates some of that color is present, and 2 indicates that color is fully present.

Table 1: Mapping Normalized Values to Colors

Colors

<table>
<tr><th></th><th colspan="2">Black - Bin 0</th><th colspan="4">Red - Bin 1</th><th colspan="3">Yellow - Bin 2</th><th></th></tr>
<tr><td>R</td><td>None</td><td>Mid</td><td>Mid</td><td>Full</td><td>Full</td><td>Full</td><td>Full</td><td>Full</td><td>Mid</td><td></td></tr>
<tr><td>G</td><td>None</td><td>Mid</td><td>None</td><td>Mid</td><td>None</td><td>Mid</td><td>Full</td><td>Full</td><td>Mid</td><td></td></tr>
<tr><td>B</td><td>None</td><td>Mid</td><td>None</td><td>Mid</td><td>None</td><td>None</td><td>Mid</td><td>None</td><td>None</td><td></td></tr>
<tr><th></th><th colspan="7">Blue - Bin 3</th><th>White - Bin 4</th><th></th><th></th></tr>
<tr><td>R</td><td>None</td><td>Mid</td><td>None</td><td>None</td><td>Mid</td><td>None</td><td>None</td><td>Full</td><td></td><td></td></tr>
<tr><td>G</td><td>None</td><td>Full</td><td>Full</td><td>Mid</td><td>Mid</td><td>None</td><td>Mid</td><td>Full</td><td></td><td></td></tr>
<tr><td>B</td><td>Mid</td><td>Full</td><td>Full</td><td>Mid</td><td>Full</td><td>Full</td><td>Full</td><td>Full</td><td></td><td></td></tr>
<tr><th></th><th colspan="5">Purple - Bin 5</th><th colspan="5">Green - Bin 6</th></tr>
<tr><td>R</td><td>Full</td><td>Full</td><td>Mid</td><td>Full</td><td>Mid</td><td>None</td><td>Mid</td><td>None</td><td>Mid</td><td>None</td></tr>
<tr><td>G</td><td>Mid</td><td>None</td><td>None</td><td>None</td><td>None</td><td>Mid</td><td>Full</td><td>Full</td><td>Full</td><td>Full</td></tr>
<tr><td>B</td><td>Full</td><td>Full</td><td>Mid</td><td>Mid</td><td>Full</td><td>None</td><td>Mid</td><td>None</td><td>None</td><td>Mid</td></tr>
</table>

Deposit

After the process of scanning finishes, the machine receives the best match of color of the brick. The next system, a servo with a slide attached, shown in Figure 25, moves to position directly over the desired bin. A servo operates on a 20 ms period and has a duty cycle of around 1 to 2 ms [6]. Using this information, the duty cycle can be adjusted to change the rotation angle of the servo. The positions the servo most move to do not increase in a linear pattern so if-statements are used to change the duty cycle and load the CCR1. For example, if the servo is to move to the first bin or “Bin 0,” the registers are loaded with 400. If the position desired is the last bin, “Bin 4” the CCR1 is loaded with 2600. These two numbers equate to .4 ms and 2.6 ms.

Figure 25: Servo with Slide

Standards and Constraints

The system was held to certain standards. One typical standard in mind is safety. Electrical components were mostly properly sealed and inaccessible to avoid electric shock or hazard. The electrical system itself is encased as much as possible to avoid harm or interference with project.

In practice, the Lego Sorter has a positive environmental and economic impact. A bin of organized Lego’s is more like to be used than an unorganized one. Thus, rather than the toys sitting around and inevitably ending up trashed, an organized pile can help promote longer usage of the toy.

Furthermore, neatly storying the sorted Lego's for later use or gifting would also facilitate this, as the Lego Company sees passing on or giving Lego's to charity as their version of recycling, as the actual recycling process is a complicated procedure [7]. Lastly, longer usage of the Lego toys translates to savings on buying future toys for a lower economic impact on the purchaser.

In terms of manufacturability, this Lego Sorter is uneasily reproduced. The project mostly consists of customized 3D Printed parts which are time consuming to produce. However, that has the opposite effect on sustainability. Because most of the project is made from plastic, the majority can be recycled.

As far as social, political, and ethical constraints, this project has none.

An IEEE Standard Code that could pertain to this project is IEEE 730-2014. It outlines requirements for initiating, planning, controlling, and executing the Software Quality Assurance processes of a software development or maintenance [7].

However, there are more physical constraints on the system that lower its effectiveness. First, there are over two thousand different Lego sizes. Some of which are oddly shaped or perhaps exceptionally small. Pieces that are oddly shaped or too big will be unable to be operated on by the system. This includes items like baseplates (Figure 26a), large "Technic" pieces (Figure 26b), and tiny studs (Figure 26c).

Figure 26: (a) Baseplate, (b) Technic, and (c) Stud

Additionally, although the bin will be designed to hold many Lego's, there will still be a maximum capacity of Lego's that can be held. There is some constraint as to the types of colors the Lego Sorter can determine. With only a set amount of output bins, colors such as pink and orange are put into a closely associated bin rather than their own.

Results

Results on the Lego Sorter functionally fully are still pending. However, the box and elevator, the conveyor belt system, and the color sensor all work independently. These will soon be implemented together.

Conclusions and Recommendations

With 19 billion Lego's created each year, about 2350 unique sizes, and 52 different colors, the diversity of Lego bricks allow for astounding amounts of creativity. However, with the enormous amount of bricks, the task of keeping some semblance of organization is an ordeal. Manually sorting Legos quickly can be a difficult, time consuming endeavor that only compounds as more Lego bricks are accumulated. The Lego Sorter created can help to mitigate that time-consuming process.

In the future, recommendations would be made to add weight to the sorter. Some elements were worked out but the reality of sorting by size or weight was never fully realized. Table 2 and 3 shows that there are easy differentiators for sorting by weight. The math and system are already complete for sorting a unique Lego brick. What is left is finding a solution for gathering weight data. This project tried to break into circuit boards on sensitive kitchen scales. This method proved ineffective and due to time constraints, no other method was able to be acted upon.

Bricks		Plates		Tile		Special			Minifig			
Size	Weight	Size	Weight	Size	Weight	Type	Size	Weight	Type	Weight		
1 x 1 x 1	0.44	1 x 1	0.2	1 x 1	0.15	Window	1 x 2 x 2	0.66	Torso	1.25		Color Code
1 x 2 x 1	0.8	1 x 2	0.36	1 x 2	0.23	Bracket	1.2 x 2 x 1.	0.8	Head	0.57		x <= .2
1 x 3 x 1	1.24	1 x 3	0.53	1 x 3	0.39	Slope	2 x 2 x 1	0.69	Legs	1.16		.2 < x <= .4
1 x 4 x 1	1.64	1 x 4	0.71	1 x 4	0.54	Corner	Tile 2 x 2	0.43				.4 < x <= .6
1 x 6 x 1	2.42	1 x 6	1.06	1 x 6	0.83	Corner	2 x 2 x .1	0.54				.6 < x <=.7
1 x 8 x 1	3.21	1 x 8	1.36	1 x 8	1.06	Corner	2 x 2 x 1	1.25				.7 < x <= 1
												1 < x <= 1.2
2 x 2 x 1	1.35	2 x 2	0.64	2 x 2	0.5	Cross	3 x 3	0.85				1.2 < x <= 2
2 x 3 x 1	1.92	2 x 3	0.93	2 x 3	0.68	Wheel	Small	0.5				2 < x <= 3
2 x 4 x 1	2.32	2 x 4	1.2	2 x 4	0.9	Wheel	Medium	1.03				3 < x <= 4
2 x 6 x 1	3.74	2 x 6	1.74			Wheel	Large	1.98				x > 4
2 x 8 x 1	4.75	2 x 8	2.27			Axel	Small	0.22				
						Axel	Medium	0.57				
		3 x 3	1.25			Axel	Large	1.02				
		4 x 4	2.22									
4 x 6	6.3	4 x 6	3.3									
		4 x 8	4.7									

Table 2: Survey of Common Brick Weights

Reorganized by weight										
1 x 1		0.2		1 x 4		0.71		2 x 2 x 1		1.35
1 x 2		0.36		1 x 6		0.83		2 x 3 x 1		1.92
1 x 1		0.15		2 x 3		0.93		1 x 3 x 1		1.24
1 x 2		0.23		2 x 4		0.9		1 x 4 x 1		1.64
1 x 3		0.39		1 x 2 x 1		0.8		1 x 8		1.36
Axel	Small	0.22		Cross	3 x 3	0.85		2 x 6		1.74
				Bracket	1.2 x 2 x 1.67	0.8		3 x 3		1.25
								Torso		1.25
2 x 2		0.64						Corner	2 x 2 x 1	1.25
1 x 3		0.53		1 x 6		1.06		Wheel	Large	1.98
2 x 2		0.5		2 x 4		1.2				
1 x 4		0.54		Legs		1.16				
2 x 3		0.68		1 x 8		1.06		2 x 4 x 1		2.32
1 x 1 x 1		0.44		Wheel	Medium	1.03		1 x 6 x 1		2.42
Head		0.57		Axel	Large	1.02		2 x 8		2.27
Window	1 x 2 x 2	0.66						4 x 4		2.22
Slope	2 x 2 x 1	0.69								
Corner	Tile 2 x 2	0.43		4 x 6		3.3				
Corner	2 x 2 x .1	0.54		2 x 6 x 1		3.74		2 x 8 x 1		4.75
Axel	Medium	0.57						4 x 6		6.3
Wheel	Small	0.5						1 x 8 x 1		3.21
								4 x 8		4.7

Table 3: Organized Brick Weights

Lego Weights in Grams						
Bin 1	**Bin 2**	**Bin 3**	**Bin 4**	**Bin 5**	**Bin 6**	**Bin 7**
0 -> .2	.2 -> .4	.4 -> .6	.6 -> 1	1 -> 1.2	1.2 -> 2	3 +

Table 4: Distribution of Weights to Bins

References

[1] J. Diaz, “Everything You Always Wanted to Know About Lego,” Gizmodo, 23-Sep-2016. [Online]. Available: https://gizmodo.com/everything-you-always-wanted-to-know-about-lego-5019797. [Accessed: 23-Mar-2019].

[2] Texas Instruments, “LM111, LM211, LM311 Differential Comparators,” LM311 datasheet, Sep. 1973 [Revised March 2017].

[3] Texas Instruments, “Quadruple Half-H Driver,” SN754410 datasheet, Nov. 1966 [Revised Jan 2015].

[4] Adafruit Industries, “Adafruit TCS3472 Color Sensor,” TCS3472 datasheet, Aug. 2012.

[5] "STM32F4DISCOVERY STMicroelectronics", Mouser Electronics, 2017. [Online]. Available: https://www.mouser.com/ProductDetail/STMicroelectronics/STM32F4DISCOVERY/?qs=J2qbEwLrpCGdWLY96ibNeQ%3D%3D. [Accessed: 01- Nov- 2017].

[6] Tower Pro, “High-Torque MG996R Digital Servo,” MG996R datasheet.

[7] “Customer Service,” *Help Topics - service LEGO.com*. [Online]. Available: https://www.Lego.com/en-AU/service/help-topics/fun-for-fans/behind-the-scenes/our-company/recycling-bricks. [Accessed: 23-Mar-2019].

[8] “730-2014 - IEEE Standard for Software Quality Assurance Processes,” IEEE-SA - The IEEE Standards Association - Home, 13-Jun-2014. [Online]. Available: https://standards.ieee.org/standard/730-2014.html. [Accessed: 12-Nov-2018].

Appendices

Full Inclusion of Code

```
#include "stm32f446.h"
//Alicia Bird
#include <stdbool.h>
#include <math.h>
```

```
//Global varaible for actuator
bool act_dir;
//0 = down; 1 = up;

void motor_setups(void);
void slow_motor_stop(void);
void slow_motor_start(void);
void fast_motor_stop(void);
void fast_motor_start(void);

void actuator_setup(void);
void actuator_stop(void);
void actuator_forward(void);
void actuator_reverse(void);

void servo_setup(void);
void slide_shift(int position);

void TIM3_setup(void);
void TIM2_setup(void);

void intr4_setup(void);
void intr2_setup(void);
void intr5_setup(void);
void intr10_setup(void);

void I2C_setup(void);
bool master(int slave_addr, int R_W, bool stop, bool ack, int number_of_data_items, int * data);
int get_color_data(void);
int determine_bin_color(float Red, float Green, float Blue, bool bw, float red_raw, float green_raw, float
blue_raw);

void SETUP(void);

int main()
{
	//Setup System
	SETUP();

	//Phase 1: Start Seperating a Piece
	actuator_forward();
	act_dir=true;
	slow_motor_start();
	fast_motor_stop();


	//INTERRUPT TRIGGERED
```

```
	//Phase 2: A Piece has fallen from First to Second Belt
	//slow_motor_stop();
	//actuator_stop();
	//fast_motor_start();

	//INTERRUPT TRIGGERED
	//Phase 3: A Piece is below the Color Sensor
	//bin = get_color_data;

	//Phase 4: Bin determinded
	//fast_motor_start();
	while(1)
	{
		//get_color_data();
	}

	return 0;
}

void EXTI2_IRQHandler()
{
	int i;
	//Actuator top trigger. Triggers when open thus the elevator has dumped and may reverse
	for(i=0;i<10000;i++);
	if(act_dir == true && (GPIOC_IDR & 1<<2) == 1<<2 && (GPIOC_IDR & 1<<4) == 1<<4)//open,
elevator past 2, may reverse
	{
		actuator_stop();
		actuator_reverse();
		act_dir = false;
		NVICISER0 &= ~(1 << 8);
	}

	if(EXTI_PR & (1<<2))//Check the specific pin, PC2
		EXTI_PR |= 1<<2;

	//turn off until bottom is triggered

}

void EXTI4_IRQHandler()
{
	int i,j;
	for(i=0;i<30000;i++);

	if((GPIOC_IDR & 1<<4) == 0)
	{
		//Actuator bottom trigger. Triggers when closed thus elevator is at bottom and may lift
```

```
			actuator_stop();
			actuator_forward();
			for(i=0;i<1000;i++)
				for(j=0;j<100;j++);
			act_dir = true;
		}
		if(EXTI_PR & (1<<4))//Check the specific pin, PC4
			EXTI_PR |= 1<<4;


		//start top int again
		NVICISER0 |= (1 << 8);

}

void EXTI9_5_IRQHandler()
{
		int i,j;
		if((GPIOC_IDR & 1 << 5) == 0)
		{
			//PHASE 2: A Piece has fallen from First to Second Belt
			slow_motor_stop();
			actuator_stop();


			for(i=0;i<500;i++)
				for(j=0;j<10000;j++);
			fast_motor_start();

			if(EXTI_PR & (1<<5))//Check the specific pin, PC5
				EXTI_PR |= 1<<5;
		}
}

void EXTI15_10_IRQHandler()
{
		int bin, avg=0, temp, i, j;
		if((GPIOC_IDR & 1 << 10) == 0)
		{
			//Phase 3: A Piece is below the Color Sensor
			fast_motor_stop();
			//toggle LED on
			GPIOC_ODR |= 1<<8;


			//take an average of color data
//			for(i=0;i<5;i++)
//			{
```

```
//                      temp = get_color_data();
//                      avg = avg + temp;
//              }
//              bin = round(avg/i);
                bin = get_color_data();
                GPIOC_ODR &= ~(1<<8);

                //Phase 4: Bin determinded
                slide_shift(bin);

                for(i=0;i<500;i++)
                        for(j=0;j<10000;j++);
                fast_motor_start();

                //while((GPIOC_IDR & 1<<10) == 0);//Wait until Lego clears
                for(i=0;i<500;i++)
                        for(j=0;j<10000;j++)
                {
                        fast_motor_start();
                }
                fast_motor_stop();
                slide_shift(3);
                slow_motor_start();
                act_dir = true;
                actuator_forward();

        }
        if(EXTI_PR & (1<<10))//Check the specific pin, PC10
                EXTI_PR |= 1<<10;
}

void intr4_setup()
{
        //RCC for SYSCFG
        RCC_APB2ENR |= 1 << 14;

                //EXTI steps
        //Configure the mask bits of EXTI_IMR: unmask bit 4 for PC4
        EXTI_IMR |= 1 << 4;

        //Configure trigger slection bits in RTSR and FTSR
        //when IR is trigger, it is a falling edge? so use FTSR
        EXTI_FTSR |= 1 <<4;

        //Configure enable and mask bits that control NVIC IRQ channel mapped to EXTI4
        NVICISER0 |= 1 << 10;

        //Set the SYSFG-> PC set in bit 2
```

```
	SYSCFG_EXTICR2 |= 2;
}

void intr2_setup()
{
	//RCC for SYSCFG
	RCC_APB2ENR |= 1 << 14;

	//EXTI steps
	//Configure the mask bits of EXTI_IMR: unmask bit 2 for PC2?
	EXTI_IMR |= 1 << 2;

	//when IR is trigger, it is a
	EXTI_RTSR |= 1 << 2;

	//Configure enable and mask bits that control NVIC IRQ channel mapped to EXTI3
	NVICISER0 |= 1 << 8;

	//Set the SYSFG-> PC set in bit 2
	SYSCFG_EXTICR1 |= 2<<8;
}

void intr5_setup()
{
	//RCC for SYSCFG
	RCC_APB2ENR |= 1 << 14;

	//EXTI steps
	//Configure the mask bits of EXTI_IMR: unmask bit 5 for PC5?
	EXTI_IMR |= 1 << 5;

	//when IR is trigger, it is a falling edge so use FTSR
	EXTI_FTSR |= 1 << 5;

	//Configure enable and mask bits that control NVIC IRQ channel mapped to EXTI9_5
	NVICISER0 |= 1 << 23;

	//Set the SYSFG-> PC set in bit 2
	SYSCFG_EXTICR2 |= 2<<4;
}

void intr10_setup()
{
	//RCC for SYSCFG
	RCC_APB2ENR |= 1 << 14;

	//EXTI steps
	//Configure the mask bits of EXTI_IMR: unmask bit 10 for PC10?
```

```
	EXTI_IMR |= 1 << 10;

	//when IR is trigger, it is a falling edge so use FTSR
	EXTI_FTSR |= 1 << 10;

	//Configure enable and mask bits that control NVIC IRQ channel mapped to EXTI3
	NVICISER1 |= 1 << 8;//position 42. 31 per register. shift to register 1. 40-31=9

	//Set the SYSFG-> PC set in bit 2
	SYSCFG_EXTICR3 |= 2<<8;
}

void TIM3_setup()
{
	//Clocks
		RCC_AHB1ENR |= 1; //GPIOA Enable
		RCC_APB1ENR |= 2; //Timer3 Enable Bit

	//Output Mode: CC1S in CCMR1 defaults to 0 which is output mode
		//CR1
		//ARPE: Auto-reload preload enable
		TIM3_CR1|=1<<7;//Register is buffered

		//EGR
		//UG:Update Generation--might have to be set right before starting timer?
		TIM3_EGR|=1;

	//CCMR1: Turn on channel 1 and 2
		//OC1PE: Output Compare 1 Preload Enable
		TIM3_CCMR1|=1<<3;//Preload enabled. R/W can access register
		TIM3_CCMR1 |= 1<<2; //OC1FE output compare 1 fast enable

			//OC3PE: Output Compare 1 Preload Enable
		TIM3_CCMR2|= 1<<3;//Preload enabled. R/W can access register
		TIM3_CCMR2 |= 1<<2; //OC3FE output compare 1 fast enable

		//OC1M: Output Compare 1 Mode
		TIM3_CCMR1|=6<<4;//PWM Mode 1

		//OC3M: Output Compare 1 Mode
		TIM3_CCMR2|=6<<4;

		//CCER:
		//CC1E:Capture/Compare 1 output enable
		TIM3_CCER|=1;//OC1 signal is output on output pin
		//CC1P:Output Polarity
		TIM3_CCER|=1; //Active Low
```

```
		//CC2E: Capture/Compare 3 output enable
		TIM3_CCER|=1<<8;//OC3 to output

		//Prescalar
		TIM3_PSC=15;

		//Outputs and ARR/CCR
		//PA6 IN ALT FOR PWM
		GPIOA_MODER|=2<<12;
		GPIOA_AFRL|=2<<24;

		//LOAD
		TIM3_ARR=20000;
		//ARR:set to 20ms
		//f=1/20m=50
		//counts=1M/50=20k
}

void slide_shift(int position)
{
	int i,j;
	TIM3_CR1|=1;

	if(position == 0)
		TIM3_CCR1 = 400;

	if(position == 1)
		TIM3_CCR1 = 800;

	if(position == 2)
		TIM3_CCR1 = 1400;

	if(position == 3)
		TIM3_CCR1 = 2025;

	if(position == 4)
		TIM3_CCR1 = 2600;
	TIM3_CR1|=1;
	for(i=0;i<1000;i++)
			for(j=0;j<10000;j++);
	TIM3_CR1 &= ~(1);
}
void motor_setups(void)
{
	//Enable Port B for use
	RCC_AHB1ENR |= 2;	//	GPIOB Enable

	//SLOW MOTOR
```

```
	//Set modes of operation
	GPIOB_MODER |= 1;				//	PB0 set to output, controls one side of
hbridge
	GPIOB_MODER |= 1 << 2;		//	PB1 set to output, controls other side of hbridge

	//motor comes up in stop mode
	//STOP: Both PB0 and PB1 are 1
	slow_motor_stop();

	//FAST MOTOR
	//PC6 and PC7
	GPIOC_MODER |= 1<<12;
	GPIOC_MODER |= 1<<14;

	//motor comes up in stop mode
	//STOP: Both PC6 and PC7 are 1
	fast_motor_stop();
}

void slow_motor_stop(void)
{
	//STOP: Both PB0 and PB1 are 1
	GPIOB_ODR |= 1;
	GPIOB_ODR |= 1 << 1;
}

void slow_motor_start(void)
{
	//START: Set PB0 to 1 and PB1 to 0
	GPIOB_ODR &= ~(1<<1);
}

void fast_motor_stop(void)
{
	//STOP: Both PB2 and PB3 are 1
	//GPIOC_ODR |= 1 << 6;
	//GPIOC_ODR |= 1 << 7;
	GPIOC_ODR &= ~(1<<6);
	GPIOC_ODR &= ~(1<<7);
}

void fast_motor_start(void)
{
	//START: Set PC7 to 1 and PC6 to 0
	//GPIOC_ODR &= ~(1<<6);
	GPIOC_ODR |= 1 << 6;
	GPIOC_ODR |= 1 << 7;
}
```

```
void actuator_setup(void)
{
	//Enable Port C for use
	RCC_AHB1ENR |= 1<<2;

	//Set modes of operation
	GPIOC_MODER |= 1;			//	PC0 set to output, controls one side of hbridge
	GPIOC_MODER |= 1<<2;		//	PC1 set to output, controls otherside of hbridge

	//start in stop mode
	actuator_stop();
}

void actuator_reverse(void)
{
	GPIOC_ODR &= ~(1);			//	switch PC0 to 0
	GPIOC_ODR |= 1 << 1;		//	switch PC1 to 1
}

void actuator_forward(void)
{
	GPIOC_ODR |= 1;						//	switch PC0 to 1
	GPIOC_ODR &= ~(1 << 1);		//	switch PC1 to 0
}

void actuator_stop(void)
{
		GPIOC_ODR |= 1 << 1;	//PC0 to 1
		GPIOC_ODR |= 1;				//PC1 to 1
}

void servo_setup(void)
{
	//set up
	TIM3_setup();
	//start from 0 (9 or 10 total positions depending on need)
	TIM3_CR1|=1;
	slide_shift(3);
	//start timer
	//TIM3_CR1|=1;

}

void I2C_setup()
{
	//The max frequency for the RGB color sensor is 400kHz
	RCC_APB1ENR |= 1<<21;
	//Set up
```

```
	I2C1_CR2 |= 10; //Set peripheral clock frequency to be 16MHz
	I2C1_CCR |= (3 << 14); //Set the master clock to be in Fast mode I2C
	I2C1_CCR |= 5; //Set CCR to be 16 MHz (master mode)
	I2C1_TRISE |= 9; //Make the rise time 10ns instead of 2
	I2C1_CR1 |= 1; //Peripherial enable

	//I2C SCL: PB6		SDA: PB7
	GPIOB_MODER |= 2<<12;
	GPIOB_MODER |= 2<<14;
	GPIOB_OTYPER |= 3<<7;
	GPIOB_AFRL  |= 4<<24;
	GPIOB_AFRL  |= 4<<28;

	//LED Toggle
	GPIOC_MODER |= 1<<16;
	GPIOC_ODR &= ~(1<<8);

}
bool master(int slave_addr, int R_W, bool stop, bool ack, int number_of_data_items, int * data)
{
	unsigned int i, j, num;
	bool success = false;

	//Set the start bit
	I2C1_CR1 |= (1 << 8);
	for(i = 0; i < 254; i++); //Allow the start to happen
	I2C1_CR1 &= ~(1 << 8);

	//Add padding time
	for(i = 0; i < 3; i++)
		for(j = 0; j < 254; j++);

	if(R_W == 1)
		//Enable ACKS to be sent
		I2C1_CR1 |= (1 << 10);

	//If the start condition was successfully send, send the received slave address plus the R/W bit
	if((I2C1_SR1 & 1) == 1)
		I2C1_DR = (0x29 << 1) + R_W;

	//Add padding time
	for(i = 0; i < 3; i++)
		for(j = 0; j < 254; j++);

	//If the slave addr was matched, send the two received data bits and stop the transmission
	if(((I2C1_SR1 & 2) == 2) && ((I2C1_SR2 & 2) == 2))
	{
		success = true;
```

```
		for(num = 0; num < number_of_data_items; num++)
		{
			//If writing data to the slave
			if(R_W == 0)
			{
				//Add padding time
				for(i = 0; i < 3; i++)
					for(j = 0; j < 254; j++);

				//Send data
				I2C1_DR = data[num];

				//If it's the last time the loop runs - send the stop condition
				if(num == (number_of_data_items - 1) && stop == true)
				{
					for(i = 0; i < 3; i++)

	for(j = 0; j < 254; j++);
					I2C1_CR1 |= (1 << 9); //STOP
				}
			}

			//If reading data from the slave
			if(R_W == 1)
			{
				while((I2C1_SR1 & (1 << 6)) == 0); //The RxNE bit is send whenever data
is transmitted
				data[num] = I2C1_DR; //RDataL

				//After the second to last data grab, ACK needs to be turned off
				if(num == (number_of_data_items - 2) && ack == false)

	I2C1_CR1 &= ~(1 << 10);

				//If it's the last time the loop runs - send the stop condition
				if(num == (number_of_data_items - 1) && stop == true)
				{
					for(i = 0; i < 3; i++)
						for(j = 0; j < 254; j++);

					I2C1_CR1 |= (1 << 9); //STOP
					if(ack == true)
						I2C1_CR1 &= ~(1 << 10);
				}
  }
			}
		}
```

```
		//Add padding time
		for(i = 0; i < 3; i++)
			for(j = 0; j < 254; j++);

		//Reset Stop Flag
		I2C1_CR1 &= ~(1 << 9);
		//If there was an acknowledge failure
		//An AF happens when the system recieves a NACK, so another start or a stop must be
generated

		if((I2C1_SR1 & (1 << 10)) == (1 << 10))
		{
			I2C1_SR1 &= ~(1 << 10);
			I2C1_DR &= 0;
			success = false;
		}

		//If there is arbitration loss (this can do an interupt, so maybe mess with that later
		if((I2C1_SR1 & (1 << 9)) == (1 << 9))
		{
									//Turn off the ARLO flag
									I2C1_SR1 &= ~(1 << 9);
									//Reset the peripherial
									I2C1_CR1 &= ~1;
									I2C1_CR1 |= 1;
									success = false;
		}

	//Add padding time
	for(i = 0; i < 3; i++)
		for(j = 0; j < 254; j++);

	return success;
}

//The address of the RGB peripherial is 0x12
int get_color_data()
{
	//MASTER TRANSMISSION
	unsigned int i = 0, j = 0, k = 0;
	bool success;
	int data[2], color_data[6];
	//Communicate to the slave to start the RGB sensing sequence
	data[0] = (1 << 7);
	data[1] = 3;
	success = master(0x29, 0, true, true, 2, data);

	//Wait for the RGB value to be taken (MESS WITH THIS - the wait time is 2.4 ms)
```

```
	for(i = 0; i < 254; i++)
		for(j = 0; j < 254; j++)
			for(k = 0; k < 254; k++);

	//If previous was successful, ask for data
	if(success == true)
	{
		//Intiate a combined protocol interaction (see slave documentation for details
		data[0] = (1 << 7) + (1 << 5) + 0x16; //Send the command code (command + auto-
increment + RDataL addr)
		success = master(0x29, 0, false, true, 1, data);
	}

	if(success == true)
	{
		success = master(0x29, 1, true, false, 6, color_data);
	}

  //Add padding time
	for(i = 0; i < 2; i++)
		for(j = 0; j < 254; j++);


	//Turn the device off after data collection
	if(success == true)
	{
			data[0] = (1 << 7);
			data[1] = 0;
			success = master(0x29, 0, true, true, 2, data);
	}

	//Do things with the collected data (which should be located in color_data
	float red, red_raw, green, green_raw, blue, blue_raw, max;

	red_raw = (color_data[0]) | (color_data[1] << 8);
	green_raw = (color_data[2]) | (color_data[3] << 8);
	blue_raw = (color_data[4]) | (color_data[5] << 8);

	if (red_raw > green_raw)
  {
    if (red_raw > blue_raw)
					max = red_raw;
    else
					max = blue_raw;
  }
  else if (green_raw > blue_raw)
				max = green_raw;
  else
```

```
        max = blue_raw;

                //Normalization
                red = floor((red_raw/max)*2);
                green = floor((green_raw/max)*2);
                blue = floor((blue_raw/max)*2);



        //would it be better to average a few and then get the bin?
        //global var and avg?
        int bin_number;
        bool bw = false;
        if(red_raw < 2500 && green_raw < 2500 && blue_raw < 2500)
                bw = true;

        bin_number = determine_bin_color(red, green, blue, bw, red_raw, green_raw, blue_raw);
        return bin_number;
}

int determine_bin_color(float Red, float Green, float Blue, bool bw, float red_raw, float green_raw, float
blue_raw)
{
        //RGB values are rounded to a selection of 3 numbers: no color(None), some color(Mid), and full
color(Full)
        //Round values:

        int bin;
        //excel has detailed list of where colors lie. editable

        //Black - Bin 0
        if(Red == 0 && Green == 0 && Blue == 0)
                bin = 0;
        if(Red == 1 && Green == 1 && Blue == 1)
                bin = 0;
        if(Red == 2 && Green == 2 && Blue == 2 && bw == true)
                bin = 0;


        //Red - Bin 1
        if(Red == 1 && Green == 0 && Blue == 0)
                bin = 1;
        if(Red == 2 && Green == 1 && Blue == 1 && red_raw < 3000 && green_raw < 2300)
                bin = 1;
        if(Red == 2 && Green == 0 && Blue == 0)
                bin = 1;
        if(Red == 2 && Green == 1 && Blue == 0)
                bin = 0;
```

```
//Yellow - Bin 0 - new, replacing black
if(Red == 2 && Green == 2 && Blue == 1)
	bin = 0;
if(Red == 2 && Green == 2 && Blue == 0)
	bin = 0;
if(Red == 1 && Green == 1 && Blue == 0)
	bin = 0;
if(Red == 2 && Green == 1 && Blue == 1 && red_raw > 2800 && green_raw > 2300)
	bin = 0;

//Green - Bin 2
if(Red == 0 && Green == 1 && Blue == 0)
	bin = 2;
if(Red == 1 && Green == 2 && Blue == 1)
	bin = 2;
if(Red == 0 && Green == 2 && Blue == 0)
	bin = 2;
if(Red == 1 && Green == 2 && Blue == 0)
	bin = 2;
if(Red == 0 && Green == 2 && Blue == 1)
	bin = 2;
if(Red == 1 && Green == 2 && Blue == 2 && blue_raw < green_raw)
	bin = 2;
if(Red == 2 && Green == 1 && Blue == 1 && red_raw < 2600 && green_raw > 2000)
	bin = 2;

//Blue - Bin 3
if(Red == 0 && Green == 0 && Blue == 1)
	bin = 3;
if(Red == 1 && Green == 2 && Blue == 2 && blue_raw > 2200)
	bin = 3;
if(Red == 0 && Green == 2 && Blue == 2)
	bin = 3;
if(Red == 0 && Green == 1 && Blue == 1)
	bin = 3;
if(Red == 1 && Green == 1 && Blue == 2)
	bin = 3;
if(Red == 0 && Green == 0 && Blue == 2 && blue_raw > green_raw)
	bin = 3;
if(Red == 0 && Green == 1 && Blue == 2)
	bin = 3;

if(Red == 1 && Green == 2 && Blue == 1 && blue_raw > 2400)
	bin = 3;


//Purple/Pink - Bin 3 - same as blue
```

```
	if(Red == 2 && Green == 1 && Blue == 2)
		bin = 3;
	if(Red == 2 && Green == 0 && Blue == 2)
		bin = 3;
	if(Red == 1 && Green == 1 && Blue == 1)
		bin = 3;
	if(Red == 2 && Green == 0 && Blue == 2)
		bin = 3;
	if(Red == 1 && Green == 0 && Blue == 2)
		bin = 3;

	//White - Bin 4
	if(Red == 2 && Green == 2 && Blue == 2 && bw == false)
		bin = 4;
	if(red_raw>  2500 && green_raw > 2500 && blue_raw > 2500)
		bin = 4;

	//if(red_raw > 1800 && red_raw < 2300 && green_raw > 1800 && green_raw < 2300 &&
blue_raw > 1800 && blue_raw < 2300)
		//bin = 0;
	//if(red_raw > 2100 && red_raw < 2400 && green_raw > 2100 && green_raw < 2400 &&
blue_raw > 2100 && blue_raw < 2400)
		//bin = 0;

	//Reject

	return bin;

}

void SETUP()
{
	//setup the system
	actuator_setup();
	motor_setups();
	servo_setup();
	I2C_setup();

	intr2_setup();
	intr4_setup();
	intr5_setup();
	intr10_setup();
}
```