Robotic Swarm

Julia Kalmer, Electrical Engineering

Project Advisor: Dr. Marc D. Mitchell

April 17, 2019

Evansville, Indiana

Table of Contents

List of Figures

I. Introduction

Swarms of robots are their own beasts to conquer; they require not only building the individual robots but figuring out how to get them to interact with one another and the world around them in an autonomous manner. A key component of a swarm is communication that is not continuous; that is, each robot should be able to talk to one another without constantly querying each other for information on location, task status, etc. The robots need to have some manor of querying each other without having a direct line of communication or be told how to move by a controller. These requirements make a swarm have another level of complexity than other autonomous robots; each robot must be very aware of where it and other robots are in space without getting direct information from its compatriots. However, these feature also benefit the technology by making the swarms able to be extremely flexible in space and in number of robots. Theoretically, once you have a robust enough system it should be able to operate just about anywhere.

Robotic swarms have a multitude of applications based upon their ability to map themselves in space, one of which is construction. The goal of the presented system is to present a proof of concept robot that will lead to a swarm that will build simple structures out of wooden blocks. The algorithm that is generated from this project will have a range of applications, from a children's toy to building simple, cheap housing.

II. Problem Statement

The design of each robot will allow it to pick up small playing blocks and place them into patterns that are specified by a programmer, the block patterns will be at least 10 x 10 blocks

wide and 3 stories high. Each robot will be able to communicate with other identically built robots; this will allow transmission of a general error signal, an individual robot's status (i.e. headed towards building area, headed towards materials area, etc.), and if the structure has been completed. A general error signal will also be indicated in a way that will let onlookers know that the robot is in peril. Each robot will be able to recognize the difference between the area where the structure is to be built and the area where the materials are; both of these areas will be marked with colored tape.

Summing it up, the minimum specifications for a singular robot will be as follows:

- Communicate with other identical robots
- Lift a small block and use it to form user-defined structures
- Move autonomously to build structures

After the initial robot has been completed and tested, more identical robots can be completed and implemented into the system. This would allow the algorithm and hardware to be more dynamic and robust with each iteration, as adding additional robots will allow underlying issues to become evident and addressed. Laying the groundwork for building a more robust system is the main goal of the project, as it would quite possibly assist further research into the solutions for several large problems.

III. Design Approach

A. Stage Design

One of the most important design aspects of this system is the stage on which the robot is going to preform, as it literally sets the stage for how the robot needs to behave and be built. As

detailed in the minimum requirements, the robot needs to transverse an area that is taped down and be able to tell where it is in space. The space in which the robot needs to move needs to be intelligently designed and simple, so three areas were determined as necessary for the robots to be able to complete their task: Start, Materials, and Building. The robots will begin in the start area, as its name suggests, and once turned on, they will either begin their task or wait for another robot to come wake them up. From the start area, the flow of movement becomes simple; the robots will move into the materials area, where they will grab a block, then go to the building area where the structure will be built. Figure 1 shows the conceptual mapping the stage and figure 2 shows the physical implementation of this design.

With staging in mind for the robots, the specifics of how each robot will know where it is in space could be determined; i.e. the robots will know what color of tape each area corresponds to. Having this the tape simplifies the building grid, however it requires the robots to have the colors hard coded into them.

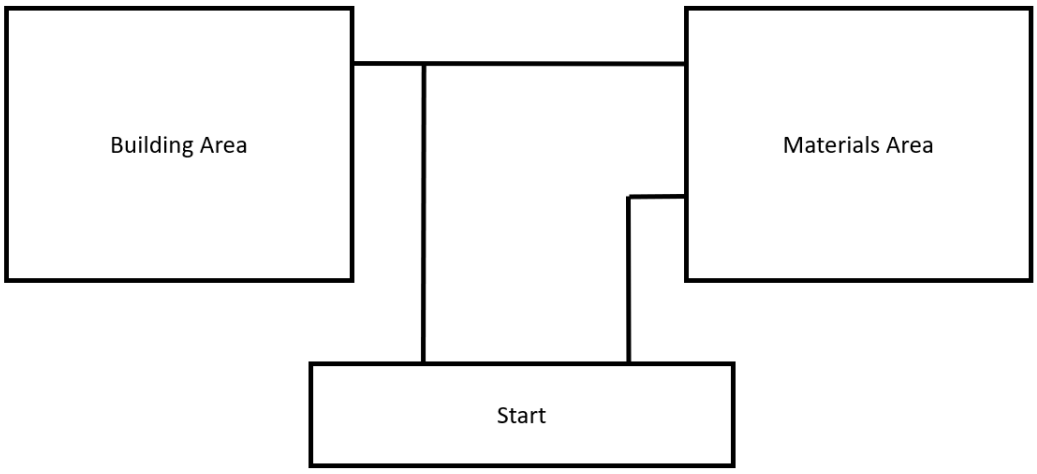

Figure 1: Conceptual design of stage

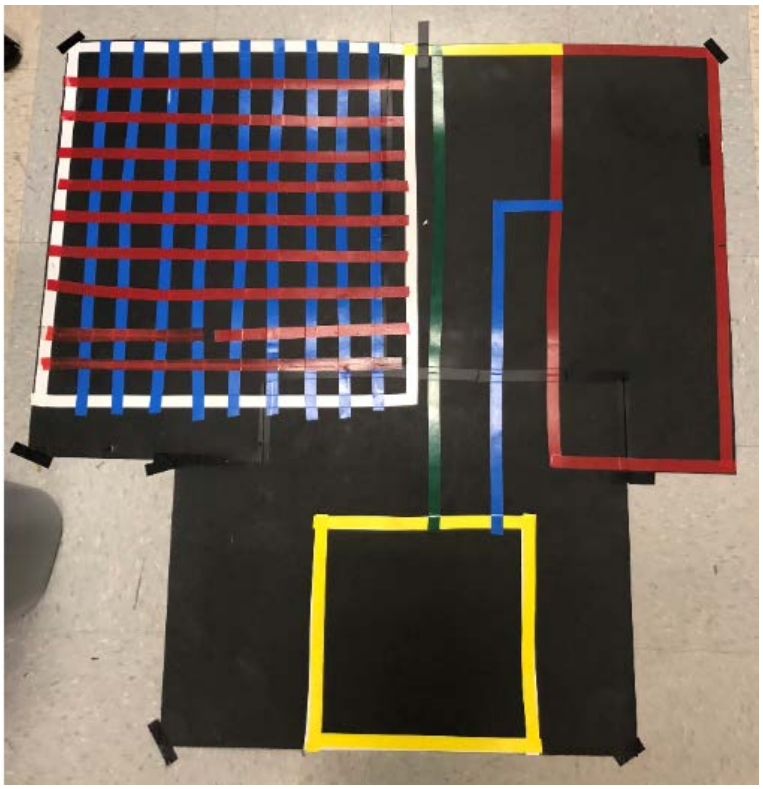

Figure 2: The physical implementation of the stage

B. Hardware Design

The robot is designed to resemble a forklift, that is, it has a base which holds all the hardware components such as the microcontroller and battery pack, and in front of the base is a lifting system. However, instead of using only a vertical lifting mechanism, the system is designed to have a vertical and a horizontal lifting ability. This allows for more versatile movement and accounts for some error in the movement of the base. The wheels of the base are not very precise, since they are just generic hobby wheels. With the proposed lifting system the robot can pick up a block and correctly place it in the field even if the base is off center.

The base of the robot is very simple in design, as can be seen by the 3D models in figures 3 and 4. The connection between the two pieces needed to be strong, however, to ensure that the lifting pieces had a stable base. It was determined that it would be a good fit to have ridges that allowed the two pieces to tightly nestle inside each other. The implementation of the connections can be seen in the 3D models of the top and bottom of the base. Originally, the design accounted for modeled holes that would increase stability, which is why they appear in figure 4, however the idea was later scrapped because it was difficult to get the holes to line up and they became unneeded. The design shown in figure 4 was not redone, however, because only one version of the lid was 3D printed and the poles were simply cut off.

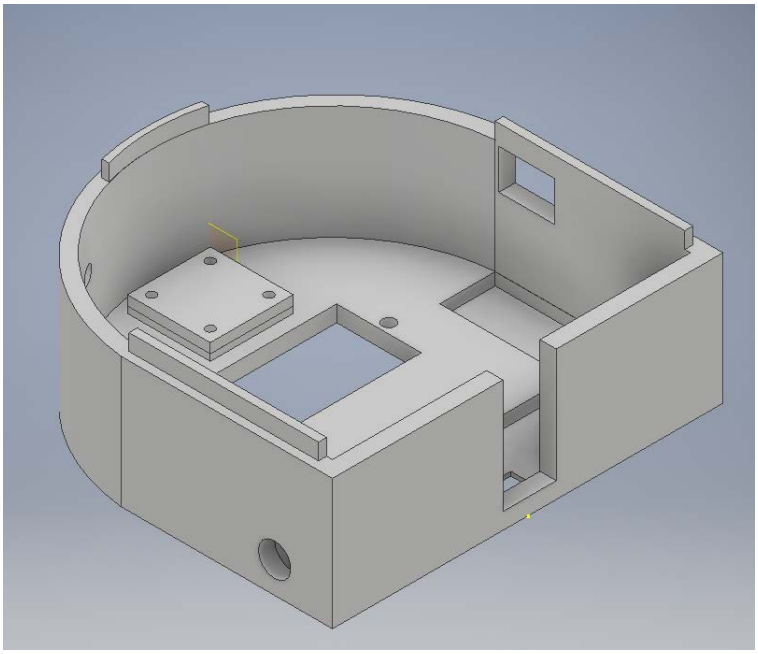

Figure 3: 3D model of the bottom of the base of the robot

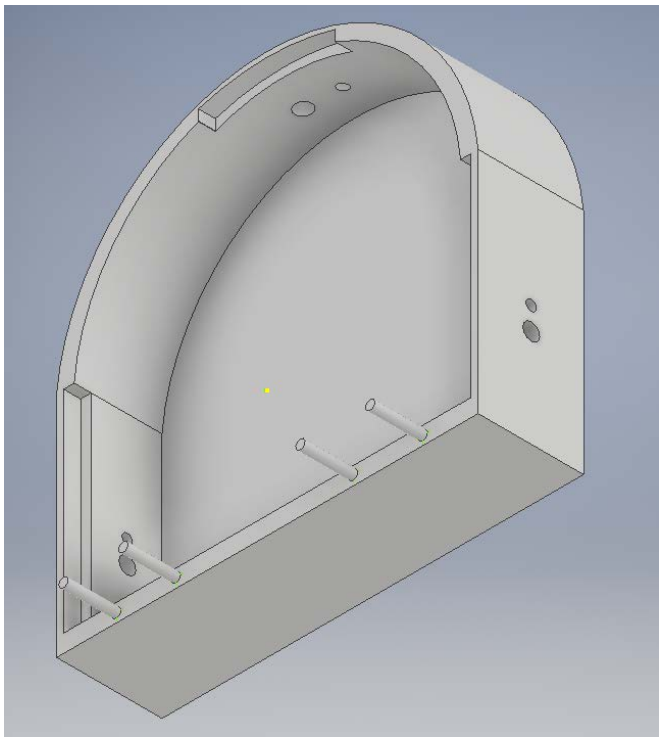

Figure 4: 3D model of the top of the base of the robot

In addition to being a strong base for the lifting, the base needs to house all of the physical components, so extra space was given to ensure that everything could fit; especially on the first iteration. The top piece is much smaller in comparison to the bottom piece because all it needs to house is holes for the IR sensors. The bottom piece is where most of the space for physical components is stored; it has an opening for the batteries to be accessed and an area to securely attach the caster ball on the back. The front wheels were designed to be external so that less space would be needed on the inside, with a roller in the back for stability. Custom fitted pieces hold the two external wheels to their respective motors. The 3D model of these pieces can be seen in figure 5. All of the base pieces fitted together can be seen in figure 6.

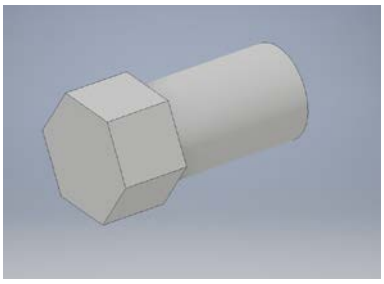

Figure 5: Back Wheel Connection Piece

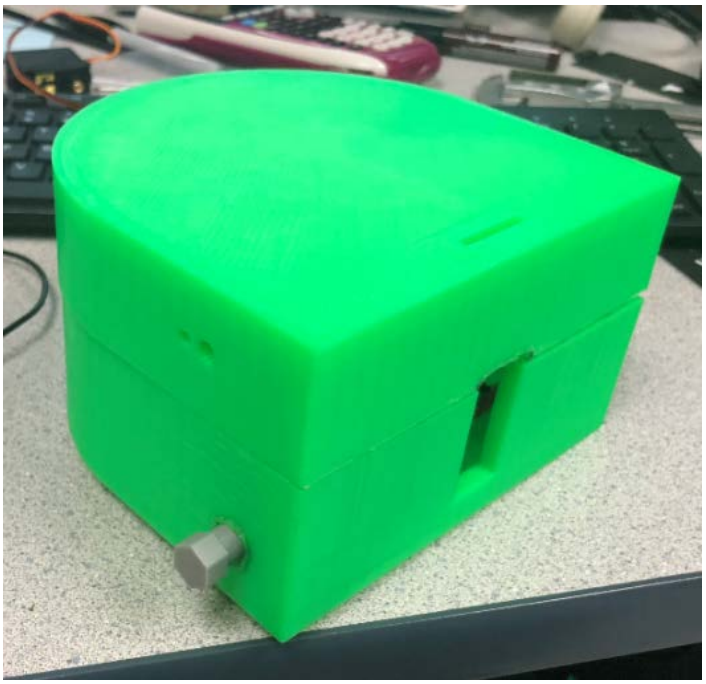

Figure 6: Printed Top and Bottom of the base fitting together

An additional feature to the top of the base that cannot be seen in figure 4 is the tab that allows the vertical lifting piece to snap onto the base for a strong connection. It can be seen on the top of figure 6.

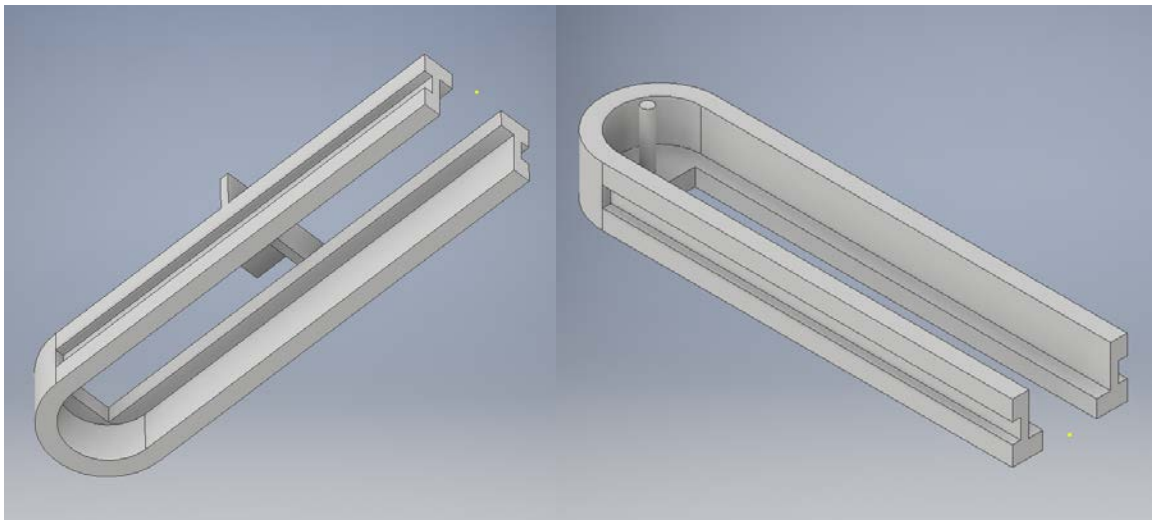

Figure 7: 3D models of the two pieces that form the vertical piece

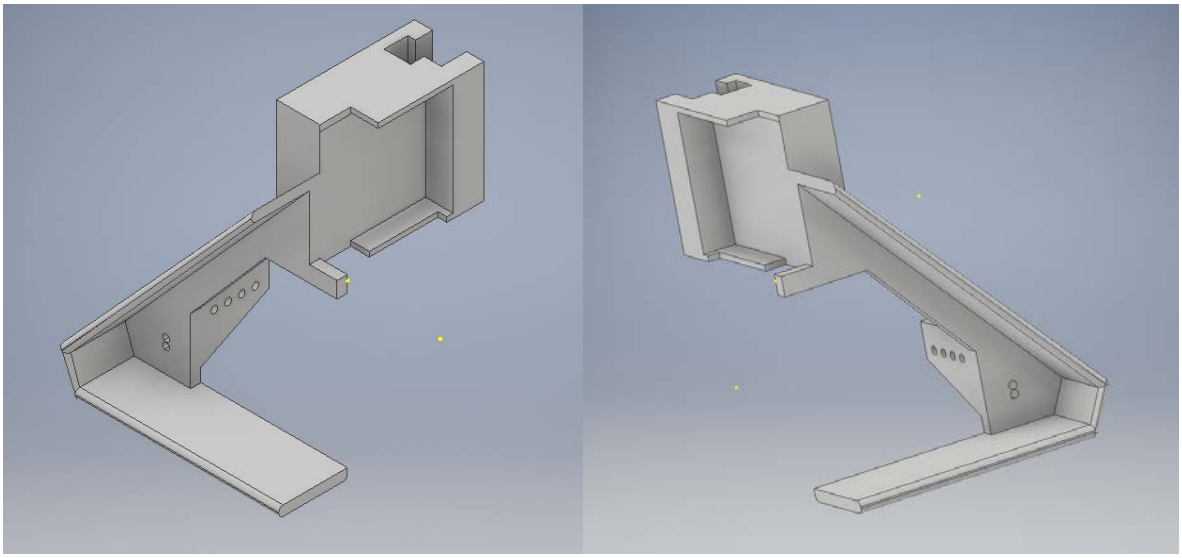

Figure 8: 3D models of the two pieces that form the horizontal piece

The lifting pieces were a lot more complex in design than the base of the robot. The 3D pieces were difficult to model, since there were a lot of considerations to make. The pieces needed to be firmly connected and needed to have enough space that things could move around

inside and outside of them. The finalized designs can be seen in the figures 7 and 8. Each piece was designed in half, since the vertical piece was too long to be printed in one piece and the horizontal piece needed to have a tight fit around the vertical piece it was going to be easier to attach if it was in two separate pieces that could be glued together. The space in the middle of both pieces are where the belts which servo motors turn are located.

The printed vertical pieces can be seen fitted together in figure 9, which also features an extra connection piece that helps hold the spinning gear on the top. The model of this piece can be seen in figure 10. It was added because it became apparent that the plastic pole would snap off when weight was added to it. The two printed horizontal pieces fitted together can be seen in figure 11, which also shows the claw that is moves back and forth. Extra support had to be designed into this piece because the claw is relatively heavy, especially because it houses its own servo motor. With the current design all the weight from the claw is supported behind the belts by metal poles, which the claw easily glides over. This also allows the belt to move more freely, since there is not a lot of weight on it. The spinning gears of this piece are also held by metal rods, because they were receiving too much friction from the plastic pieces, which also broke off easily. It became simpler to just implement metal poles instead of capping each end of the system, as that would limit the area in which the claw can move.

The horizontal piece comes together to snap into the ridges of the vertical lifting piece, which can be seen in figure 12. As mentioned before, the vertical piece hooks onto the base, which can be seen in figure 13.

Figure 9: Printed version of the vertical lifting pieces fitted together

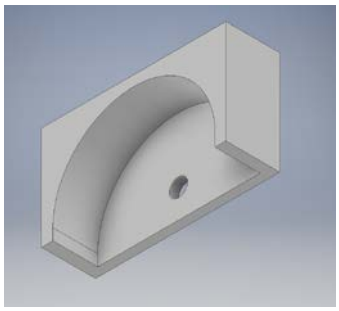

Figure 10: 3D model of additional vertical piece

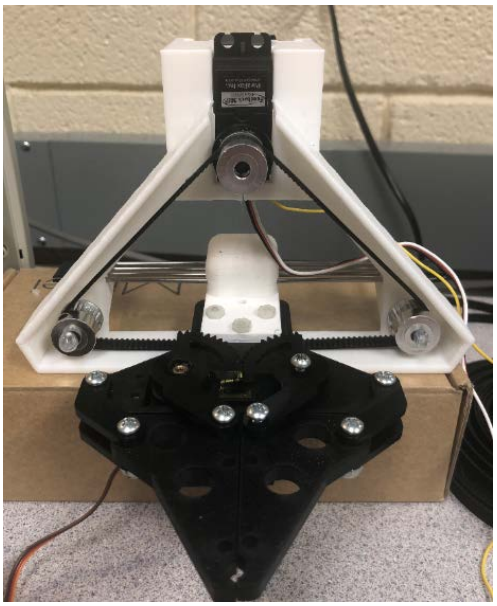

Figure 11: Printed version of horizontal lifting pieces fitted together

Figure 12: Lifting mechanism

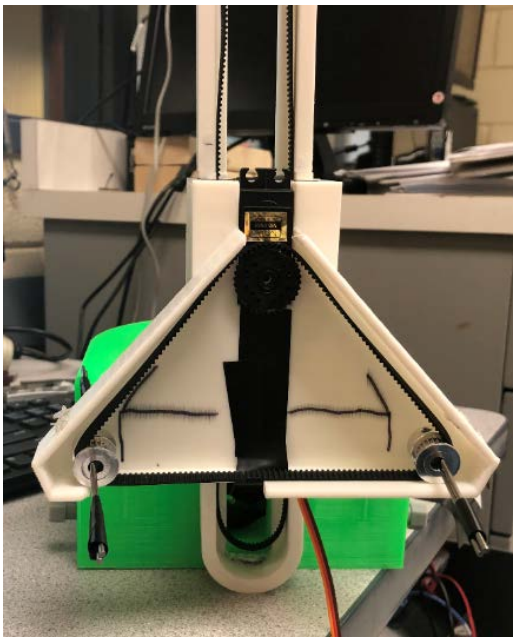

Figure 13: Lifting mechanism connected to the base

Belts are used in the vertical and horizontal pieces to allow for movement, which are driven by continuous motion servos, which also allow for position feedback. The belt of the vertical piece is very straightforward and can be seen in figure 9. A servo motor that sits in the base of

the robot rotates the belt, which in turn drags the horizontal piece up and down. The belt of the horizontal piece can be seen in figure 11. The servo motor sits at the top of the triangle and rotates the belt, which attaches to the claw, and moves it back and forth. The triangle design was decided upon for space and aesthetic reasons.

The claw that was chosen is a parallel gripper kit A from Actobotics [1], it holds its own servo motor and it able to open wide enough to hold a small wooden block. The claw can be seen in figure 14, both open and closed. It attaches to the horizontal lifting piece with a connector that it screws into (shown in figure 15). The claw is attached to both ends of the belt, which allows it to be moved from side to side and keeps the belt taut, this can be seen in figure 11. There is no reason why the belt ever needs to loop completely around, so attaching the belt to the claw and not making it a continuous loop allows for the servo motor to make the motion smoother and skip less.

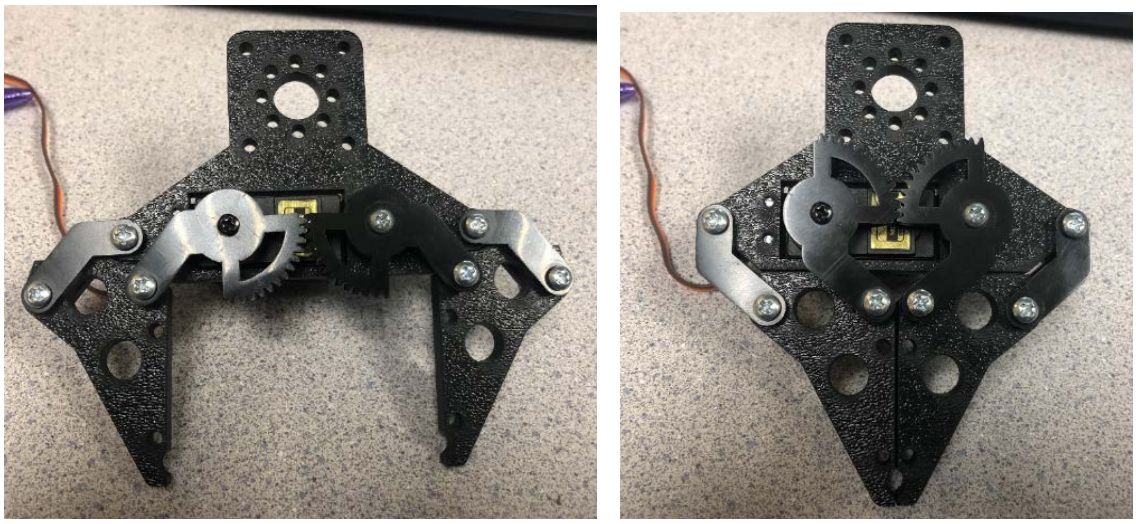

Figure 14: The claw

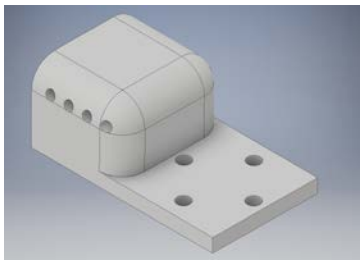

Figure 15: Claw connection piece

Aside from the 3D modeling that needed to happen, the sensors that were chosen were very important to the success of the project. An RGB color sensor needed to be implemented to help the robot know where it is in space, a proximity sensor needed to be used to allow the claw to know how far away the block it's picking up is, and IR emitters and receivers were chosen for use to allow the robots to communicate with one another.

The chosen RGB color sensor[1] and proximity sensor[2] are shown in figure 16 and were both purchased from AdaFruit. They both use I2C protocols for use and share a bus, the diagram for which is shown in figure 17. The RGB color sensor is connected to on the base of the robot with an extender piece that allows it to be in front of the robot to better follow the colored tape stage laid on the floor. There are two implemented proximity sensor, once housed near the RGB color sensor and the other mounted on the claw. Both allow the robot to very precisely know where objects in front of the robot are located.

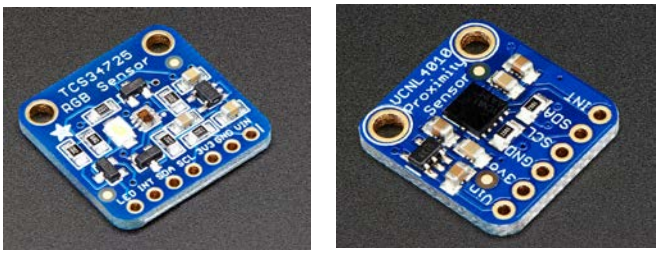

Figure 16: (a) RGB Color Sensor – TCS34725 (b) Proximity Sensor VCNL 4010

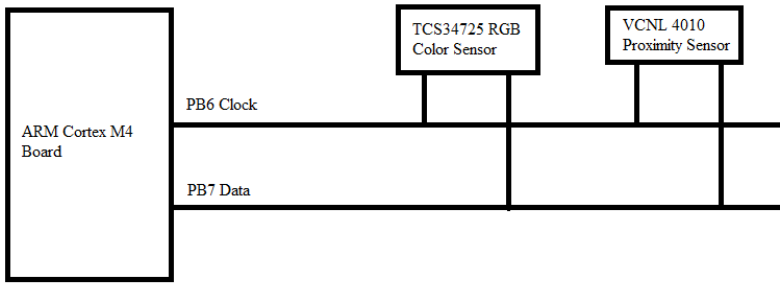

Figure 17: I2C protocol bus

The robot is outfitted with three pairs of IR emitters and receivers to work, all mounted on the top piece of the base facing away from the robot. The circuits for these pairs are shown in figure 18. IR sensors were chosen because they are relatively cheap and sensing things that are very far away is not a priority for the communication between two robots – they only need to know if another robot is in the vicinity and if it the other robot is blinking its IR sensors on and off.

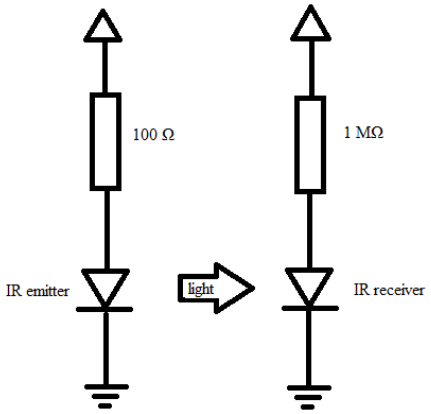

Figure 18: IR emitter and receiver circuit diagram

C. Software Design

The software for this project was all programmed onto the ARM STM32F446 Nucleo-64 board using Keil uVision 5. There are several important protocols for the project, namely the algorithm that builds the structure and communicating with another robot.

The algorithm has a very simple flow, however, may have to be changed as the system is grown and unforeseen errors arise. Since the blocks will have to be placed in a grid, the algorithm will have the robots begin building in the middle of the given structure and move outward in the grid. However, this method could lead to crowding if enough robots are all working at once, which is be accounted for by having robots wake each other up as each section of building is completed. Having the robots begin building in the middle of the structure should help cut down on the amount of human oversight, as there will not need to be a human to place the first block.

The flow for the system is detailed in the block diagram in Figure 20, and the sample grid is shown in Figure 21. The sample grid is what the robot sees, it will begin building in zone 1, which is indicated by a 1 in the box of the grid. Each time the robot approaches the grid, it will randomly decide which square it will attempt to put a block in. It will drive to the block, then check to see if there is already a block in place there, also checking to see if there are any blocks in the way. The motion protocols that allow the robot to move without accidentally bumping placed blocks will be detailed later. If there is a block in place in the spot that the robot randomly closennand no more blocks need to be stacked in that space, the robot will move to a different square in the zone until the block that it is holding is placed.

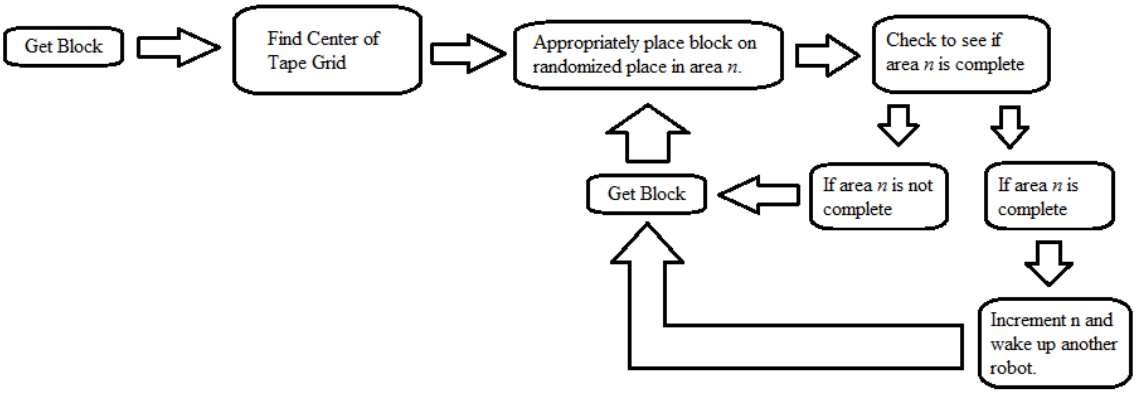

Figure 20: Block Diagram of Center Building Algorithm

5	5	5	5	5	5	5	5	5	5
5	4	4	4	4	4	4	4	4	5
5	4	3	3	3	3	3	3	4	5
5	4	3	2	2	2	2	3	4	5
5	4	3	2	1	1	2	3	4	5
5	4	3	2	1	1	2	3	4	5
5	4	3	2	2	2	2	3	4	5
5	4	3	3	3	3	3	3	4	5
5	4	4	4	4	4	4	4	4	5
5	5	5	5	5	5	5	5	5	5

Figure 21: Grid that the robot works on

As stated before, communication is very important with this project, as it keeps the robots from running into each other and allows them to transmit information between them. The robots communicate using IR sensors and will be continuously checking for other robots in the vicinity. The exact protocol that the robots will use is detailed in figure 22.

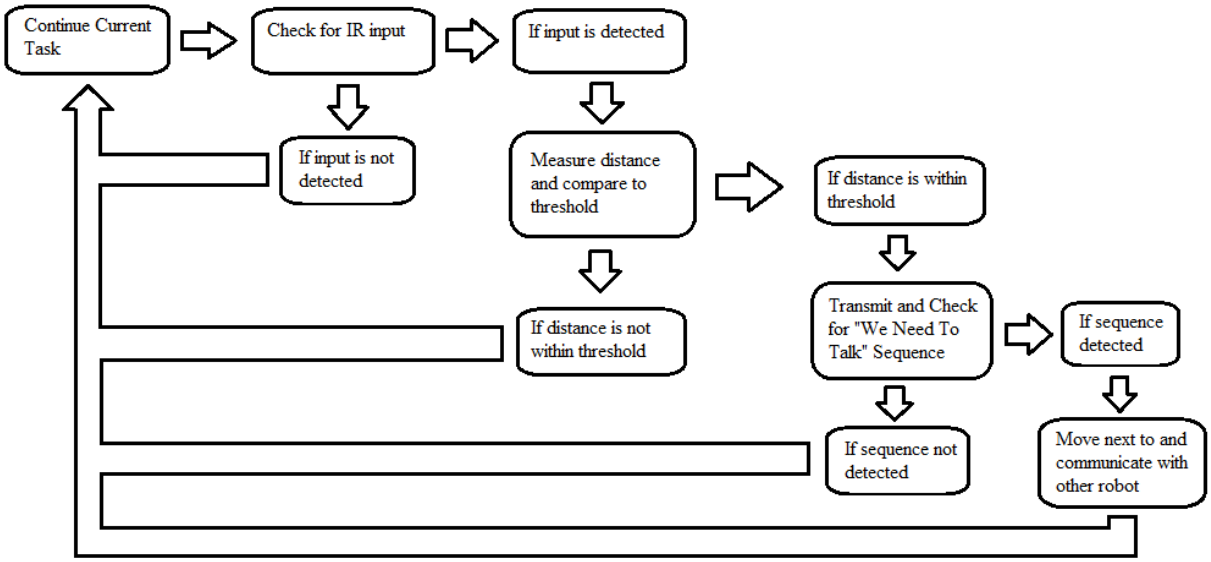

Figure 22: Communication protocol flowchart

The way that the robot tells where it is in space depends heavily on hardware but is driven completely by software. The way that the robot moved turned out to be much more complex than what was initially imagined, since the chosen wheels cause the robot not to drive completely straight and ambiguity of which direction the robot starts needed to be accounted for. This made the robot being able to correct its course during area changes very important, so the robot approaches the colored tape and then turns either left or right in the current implementation based upon which direction was deemed most likely for error. The robot also internally keeps track of which area it is in and what color of tape it needs to find.

Another important component to movement is how the robot indexes itself into the grid, since it needs to be very precise in order to correctly place the gathered blocks. The way that it does this also needs to account for other robots placing blocks, since this system is meant to be a

robotic swarm, which lead to the addition of another proximity sensor in the front of the robot with its own I2C bus. This proximity sensor is much more precise than using IR transmitters and receivers, but the precision is needed since the IR light would not properly bounce off the building materials. There are many cases that the robot needs to account for when moving in the grid, all of which are summarized in figure 23.

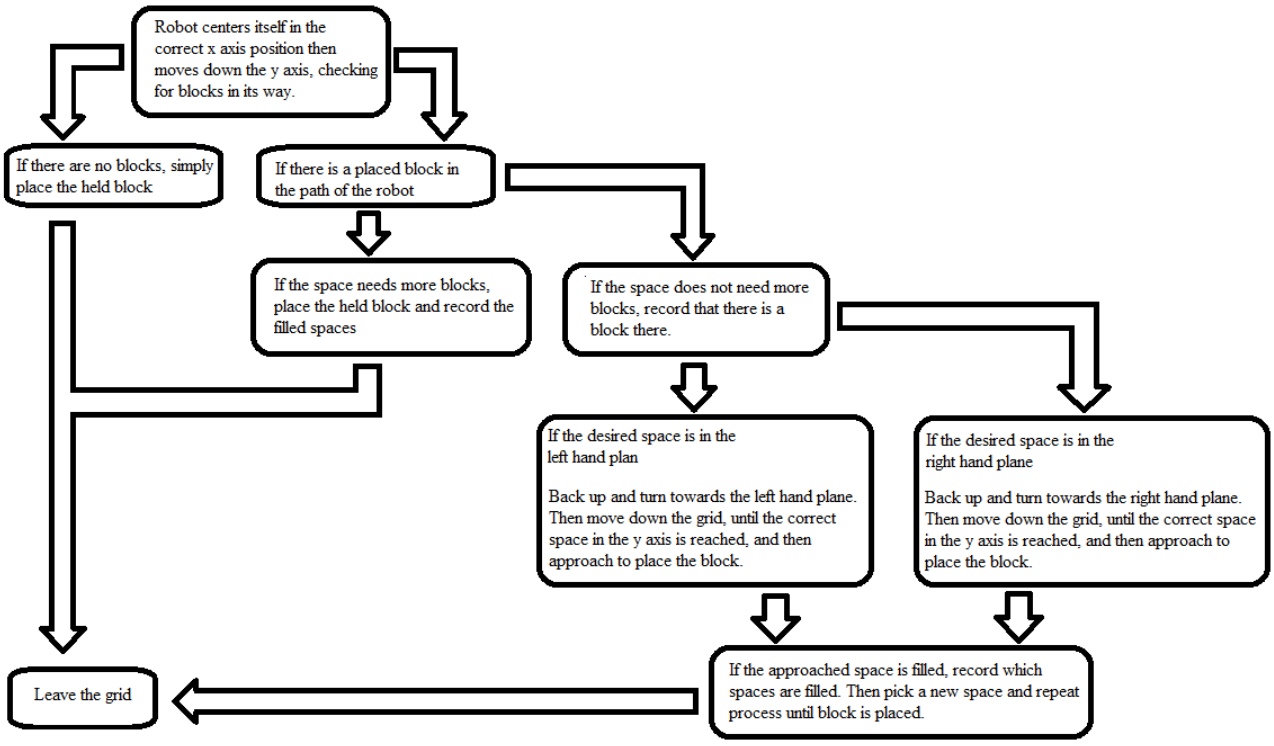

Figure 23: Grid Indexing Flowchart

Essentially, if the robot finds a block while it is taking the most direct route to its destination, it reroutes itself based upon which side of the plane it is in. What the robot considers to be the right and left hand planes can be seen in figure 24, which is a simple illustration that shows what the robot sees as far as actual indexable numbers and areas. The point (5, 5) corresponds to where the robot would enter the grid. If the block is in the left hand plane, it will turn and take a more direct route to that square by turning and moving further down the grid,

then turning to approach the empty square from the side instead of the grid instead of directly downward. If the block is more easily accessed by the bottom, this is also accounted for. This allows for the system to take less time in building and for multiple robots to be placing blocks in the areas at random. As the robot places and encounters blocks, it will record them internally, and if the area is completed then it will index the area counter, as detailed in the overall robot flow.

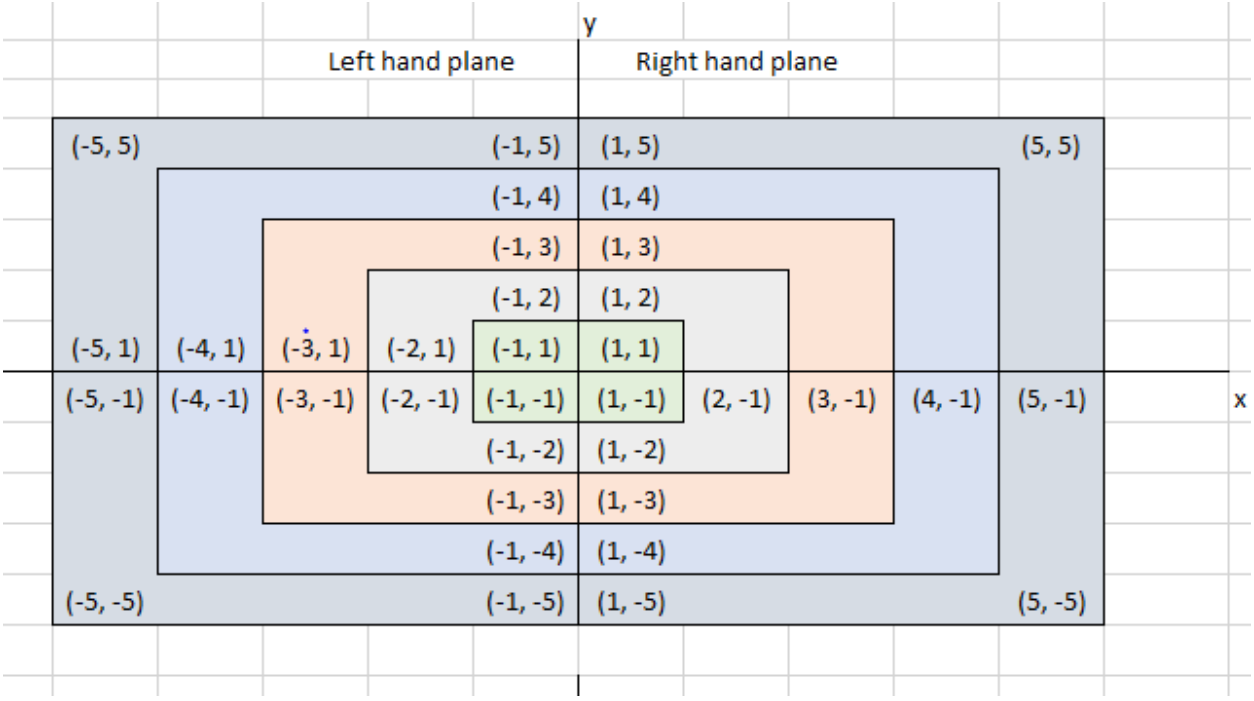

Figure 24: Grid with areas and indexing values

A sample of the way that the robot might move is shown in figure 25. In this example, there is a block in the way of the robot, marked as a square, and its destination is marked with a circle. The robot approached and senses the block in its path, determines that the block is wants

to place lies in the left half plane, and then turns to approach that space. In this example, there is not a block in the desired location already. If there were, then the robot would randomly choose another space, check internally if it was open, and continue this process until the area was finished. Once the area was finished, the robot would move to the next area.

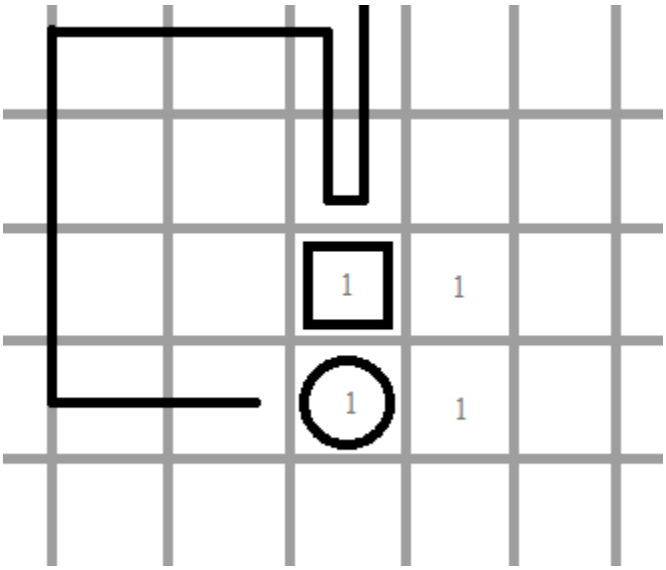

Figure 25: Example Robot Movement

Similar to the movement, the lifting mechanism is heavily dependent on hardware, however, is completely driven by software. The flow of how a robot lifts a block is as follows; the robot first orients itself in front of the block, then scanning vertically, finds the top block in a stack. It then centers itself on the block horizontally so it can then lift the block. The claw knows where a block is using the proximity sensor in Figure 16b.

D. Standards and Constraints

There do not appear to be any applicable IEEE standards for this project's hardware, because rechargeable batteries are not being used and the communication is system specific, so does not need to comply to any given standards. There also do not appear to be any applicable

IEEE software standards that apply to this project, since it does not use any standardized communication protocols other than I2C, which does not appear to have any IEEE standards.

The manufacturability of this project is very high once a working prototype is made and optimized, because the robots need to be identical to be able to work together. This project is possibly eco-friendly, depending on what type of materials would be used for the final product. The prototypes built are inherently not eco-friendly, since they use so much materials for testing, but the final product could be modified to be ecologically friendly by using things such as on-board rechargeable batteries and recycled plastic. The possible political impact of this project is that if it were implemented to make low cost housing or free easily maintainable housing, the action of doing so could upset some people who believe that those who are without homes should have to work to have a roof over their heads. The homes that the robots could build would not be stylish, however, and would give those who would not otherwise have it a place to stay without burdening the community if an entire swarm were bought once by the city. It would ultimately cut the spending that cities use for homeless people, since the housing only needs to be built once and then maintained.

A possible danger that the prototype at least could pose is that the robots will not sense for people in their path, which could make them dangerous to people passing through the area where the robots are working; especially if they are working on a large scale housing project. Adding a system to see if there is something in their path would be another good addition to a final product. Once a final model is created, the system should be relatively sustainable, as the robots can be modified to withstand many given conditions and can be easily replaced and interchanged. The important part of this project is not really the robots, it is the algorithm, which should be able to be applied to various situations.

IV. Results

The result of this project is a system that successfully senses and lifts blocks and moves around the track. The lifting algorithm was successfully implemented and debugged, as was the movement algorithm. However, the indexing system into the grid lacks the proper troubleshooting to be called finished, although the theoretical algorithm should be able robust enough to be successful once properly tested.

The main reason that the grid indexing system was unable to be completed was because of some unforeseen hardware issues that caused major issues. The color sensor needed to be moved forward in the system to allow the sensor to be able to react to immediate changes on track. In a future implementation, the color sensor would be eliminated, however it was necessary for this implementation. The piece that was made to account for this physical change was difficult to put on the system, which made troubleshooting difficult. Despite this, the movement algorithm was almost finished before the project deadline, and works almost as planned.

The lack of the fully developed indexing array capabilities made the fully integrated system not possible. However, as previously mentioned, the lifting system as a whole worked well when sensing and lifting blocks. The lifting system was able to independently center itself and lift blocks consistently. The movement between two areas also worked well and was rigorously tested, with many errors in the color system found and accounted for. Overall this system is very well tested and successful, it simply lacked the final piece for complete integration.

V. Conclusion

The implemented algorithm theoretically proves that swarm robotics can build simple structures out of whatever material is presented. The algorithm can be used for whatever robots need its application, which was the main goal of the project. The created system can be used for demonstrations of how useful swam robotics can be to potential students or could be refined at this scale to be sold as a toy to children who cannot play on the floor with their own blocks. Obviously, this system has only been shown to work for one robot with the theoretical implication of more, so in the future the system can be refined to work for more and more robots so that the structures can be built quicker. A companion app can also be developed if the system is implemented for a children's play toy or even on a larger scale for housing purposes; having a control to monitor the status of the robots and to act as a kill switch would be very important to a finalized product. If the project were to be put onto the market, the robot would also require a system to sense where it is without having to follow a tape grid. This allows for the area of things that can be built can to be expanded. The claw systems would also have to be changed to allow for multiple building materials, so that things can be made with Lego blocks or wooden sticks or whatever else is required.

Robotic swarms are obviously a very useful technology, and the presented system expands the technology further into the future with a new algorithm and physical system. Construction is an easy task for robots to preform, although they lack the ability to come up with how the structures should look. This system allows for humans and robots to work together to build simple structures.

Appendix A

This appendix includes the code used to drive the robots.

```
//Senior Project main code
//This code will drive the robots to do all the things

#include <math.h>
#include <stdbool.h>
#include <stdint.h>
#include "stm32f446.h"
#include "TimerDelay.h"
#include <stdlib.h>

//Program Status: Works and Tested
//Programs: Move_Full/Left/Right_Forward/Backward();
//These programs move the Left/Right wheels forward or backwards, or both wheels
forward and
// backwards. The program keeps them running in that direction for one (1) second.
void Move_Full_Forward(void);
void Move_Half_Forward(void);
void Move_Quart_Forward(void);
void Move_Left_Forward(void);
void Move_Quart_Backward(void);
void Move_Right_Forward(void);
void Move_Full_Backward(void);

//Program Status: Tested 0 -> 1 (works fine) and 1 -> 2 (should work?) but not the
others
//Program: Switch_Area(int desired)
//Makes the robot go between two areas, from the current area to the desired area.
void Switch_Area(int desired);

//Program Status: Not tested - only psuedocoded
//Program: Grid_Index(int x, int y)
//Makes the robot move into the grid and then position itself so it can lay a block in
it
void Grid_index(signed int x, signed int y, signed int z);

//Program Status: Not tested - only psuedocoded
//Program: Grid_exit(int x, int y)
//Makes the robot move out of the grid based on the current position and stop on the
white
void Grid_exit(void);

//Program Status: Works
//Program: I2C_set_up();
//This program sets up the I2C clock
void I2C_set_up(void);

//Program Status: Works and tested with receiver and transmitter
```

```
//Program: master(int slave_addr, int R_W, bool stop, bool ack, int
number_of_data_items, int * data);
//This program uses I2C interface that is on PB6 (Clock) and PB7(Data) to either send
or get data
// to/from the slave peripherial. It requires the slave address, if it is in read (1)
or write(0)
// mode, if a stop should be sent, the number of data items to be sent, and the data
itself. The data
// array will either be filled or read from by the program, depending on its mode.
bool master(int slave_addr, int R_W, bool stop, bool ack, int number_of_data_items,
int * data);

//Program Status: Tested and works
//Program: get_proximity_data();
//This program contacts the proximity sensor via and I2C interface that is on PB6
(Clock) and
// PB7(Data) and then returns a single number that indicated the proximity from the
sensor.
unsigned long get_proximity_data(void);

//Program Status: Tested and works
//Program: get_color_data(unsigned int * data);
//This program contacts the RGB color sensor via and I2C interface that is on PB6
(Clock) and
// PB7(Data) the program returns a number which indicates the color. The number code
is as follows:
//   Error = 0
//   Red = 1
//   Yellow = 2
//   Blue = 3
//   Green = 4
//   White = 5
int get_color_data(void);

//Program Status: Works and Tested
//Program: ADC_Setup();
//Sets up the ADC1 for the IR emitters on PC4, PC5, and PA7 for continuous
conversions, and the data
// is sent to a global ADC_data by DMA
void ADC_Setup(void);

//Program Status: Works, not tested much
//Program: check_nearby();
//This program should check to see if there is another robot nearby by checking to see
if the IR
// sensor is picking up light above a certain signal.
bool check_nearby(void);

//Program Status: Skeleton is built, but not tested
//Program: Communication();
//Once another robot is found nearby, this program activates to talk to it.
void Communication(bool area_switch, bool wake_up);

//Program Status: Fully Works
//Program: PWM_Setup();
```

```
//Sets up PWM on pins PA6,
void PWM_Setup(void);

//Program Status: Fully Works
//Program: Claw_Motor_Control(double degrees);
//Moves the Claw to percentage open, 0 to 100%
void Claw_Motor_Control(double percent);

//Program Status: Fully Works
//Program: PWM_Capture_Setup();
//Sets up the capture on TIM4
void PWM_Capture_Setup(void);

//Program Status: Tested and Works
//Program: Horz_Scanning
//Assuming that there is a block in front of the sensor, find the horizontial edges
and then center the claw
// to the block. Right now we are not checking if the block can be grabbed or not.
void Horz_Scanning(void);

//Program Status: Tested and Works
//Program: Vert_Scanning
//Assuming that there is a block in front of the sensor, finds the top of the stack
and moves the proximity
// sensor in front of it.
void Vert_Scanning(void);

//Program Status:
//Program: Grab_Block();
//Assuming that there is a block in front of the sensor, grabs the block using other
programs
void Grab_Block(void);

short ADC_data[3];
int curr_area = 0;
double duty_PB8, duty_PB9;
//Grid
	bool lower[10][10], ideal_lower[10][10], middle[10][10], ideal_mid[10][10],
		top[10][10], ideal_top[10][10];

int main()
{
	//FMPI2C peripheral clock must be configured and enabled in the clock
controller
	RCC_APB1ENR |= (1 << 21); //Enables I2C1

	//Pin Set-Up
	RCC_AHB1ENR |= 7; //Enable GPIOA and GPIOB

	//Proximity and Color Sensor Set up
	//PB6 (Clock) and PB7(Data) are for I2C1
	GPIOB_MODER |= (2 << 12); //Set PB6 to be in alternate function mode
	GPIOB_MODER |= (2 << 14); //Set PB7 to be in alternate function mode
	GPIOB_OTYPER |= (3 << 6); //Set PB7&PB6 to be open drain
	//Use PA5 as an enable for the sensors
```

```
//GPIOA_MODER |= (1 << 10);
//AF4 is for I2C1
GPIOB_AFRL |= (4 << 24);
GPIOB_AFRL |= (4 << 28);

//IR Sensor Pin Set up (PA7, PA2, PA3)
GPIOA_MODER |= (3 << 14); //PA7
GPIOC_MODER |= (3 << 8); //PC4
GPIOC_MODER |= (3 << 10); //PC5

//Motion Protocols
//PC0 & PC1 will be right wheel positive and negative
//PC6 & PC7 will be left wheel positive and negative
GPIOC_MODER |= (1 << 0);
GPIOC_MODER |= (1 << 2);
GPIOC_MODER |= (1 << 12);
GPIOC_MODER |= (1 << 14);

//Servo Motor Protocols
//PA6 is TIM3_CH1 - AF2
GPIOA_MODER |= (2 << 12);
GPIOA_AFRL |= (2 << 24);
//PB0 is TIM3_CH3 - AF2
GPIOB_MODER |= 2;
GPIOB_AFRL |= 2;
//PB1 is TIM3_CH4 - AF2
GPIOB_MODER |= (2 << 2);
GPIOB_AFRL |= (2 << 4);

//PB8 is TIM4_CH3 - AF2
GPIOB_MODER |= (2 << 16);
GPIOB_AFRH |= 2;
//PB9 is TIM4_CH4 - AF2
GPIOB_MODER |= (2 << 18);
GPIOB_AFRH |= (2 << 4);

I2C_set_up();
ADC_Setup();
PWM_Setup();
PWM_Capture_Setup();

//Testing LEDS PC7 - PC11
GPIOC_MODER |= (1 << 14);
GPIOC_MODER |= (1 << 16);
GPIOC_MODER |= (1 << 18);
GPIOC_MODER |= (1 << 20);
GPIOC_MODER |= (1 << 22);

//bool nearby = false;
while(1)
{
	Grab_Block();

	Delay_ms(2000, 0);
```

```
			TIM3_CCR1 = 500;
			TIM3_CCR4 = 770;

			Delay_min(1, 0);
		}

		//1. Collect Proximity Baseline
		//2. Am in the Start Area - move out of the start area
}

void Move_Full_Forward()
{
		//PC0 & PC6 to be 1
		//PC1 & PC7 to be 0
		GPIOC_ODR |= (1 << 0);
		GPIOC_ODR |= (1 << 6);

		//int j, i;
		//Add padding time
		Delay_ms(100, 0); //Run for .1 second

		GPIOC_ODR &= ~3;
		GPIOC_ODR &= ~(3 << 6);

}

void Move_Half_Forward()
{
		//PC0 & PC6 to be 1
		//PC1 & PC7 to be 0
		GPIOC_ODR |= (1 << 0);
		GPIOC_ODR |= (1 << 6);

		//int j, i;
		//Add padding time
		Delay_ms(50, 0); //Run for .1 second

		GPIOC_ODR &= ~3;
		GPIOC_ODR &= ~(3 << 6);
}

void Move_Quart_Forward()
{
		//PC0 & PC6 to be 1
		//PC1 & PC7 to be 0
		GPIOC_ODR |= (1 << 0);
		GPIOC_ODR |= (1 << 6);

		//int j, i;
		//Add padding time
		Delay_ms(25, 0); //Run for .1 second

		GPIOC_ODR &= ~3;
		GPIOC_ODR &= ~(3 << 6);
}
```

```
void Move_Left_Forward()
{
	//PC0 & PC7 to be 0
	//PC1 & PC6 to be 1
	GPIOC_ODR |= (1 << 1);
	GPIOC_ODR |= (1 << 6);

	//Add padding time
	Delay_ms(50, 0); //Run for .1 second

	GPIOC_ODR &= ~3;
	GPIOC_ODR &= ~(3 << 6);
}

void Move_Right_Forward()
{
	//PC1 & PC7 to be 1
	//PC6 & PC0 to be 0
	GPIOC_ODR |= (1 << 0);
	GPIOC_ODR |= (1 << 7);

	//Add padding time
  Delay_ms(50, 0); //Run for .1 second

	GPIOC_ODR &= ~3;
	GPIOC_ODR &= ~(3 << 6);
}

void Move_Full_Backward()
{
	//PC1 & PC7 to be 1
	//PC0 & PC6 to be 0
	GPIOC_ODR |= (1 << 1);
	GPIOC_ODR |= (1 << 7);

	//Add padding time
	Delay_ms(100, 0); //Run for .1 second

	GPIOC_ODR &= ~3;
	GPIOC_ODR &= ~(3 << 6);
}
void Move_Quart_Backward()
{
	//PC1 & PC7 to be 1
	//PC0 & PC6 to be 0
	GPIOC_ODR |= (1 << 1);
	GPIOC_ODR |= (1 << 7);

	//Add padding time
	Delay_ms(25, 0); //Run for .1 second

	GPIOC_ODR &= ~3;
	GPIOC_ODR &= ~(3 << 6);
}
```

```
void Switch_Area(int desired)
{
	int color;
	color = get_color_data();

	//If in the starting area and wanting to go to the materials area
	if(curr_area == 0 && desired == 1)
	{
		//Move to the outside of the zone
		while(color != 2)
		{
			Move_Full_Forward();
			color = get_color_data();
		}

		//Follow the yellow tape
		while(color == 2)
		{
			Move_Full_Forward();
			color = get_color_data();

			//Turn Left if the tape changes color to not be green or blue
			while(color != 2 && color != 4 && color != 3)
			{
				Move_Left_Forward();
				color = get_color_data();
			}
		}

		//If we found the white tape
		while(color == 5)
		{
			//Move forward until the blue tape is found
			while(color != 3)
			{
				Move_Full_Forward();
				color = get_color_data();

				//If the robot goes off center
				while(color != 5 && color != 3 && color != 2)
				{
					Move_Right_Forward();
					color = get_color_data();
				}
			}
			//Might need to add in error checking if the robot is at an angle.
		}


		//If we found the blue tape, follow it to the materials area
		while(color == 3)
		{
			Move_Full_Forward();
			color = get_color_data();
```

```
		//Turn if the tape changes color to not be white
		while(color != 3 && color != 5)
		{
			Move_Right_Forward();
			color = get_color_data();
		}
	}

	//Once you've reached the red area, you're in the materials area!
	curr_area = 1;
}

//If in materials area and wanting to go to build
if(curr_area == 1 && desired == 2)
{
	//Find the red
	while(color != 1)
	{
		Move_Full_Forward();
		color = get_color_data();
	}

	//Follow the red until it turns yellow
	while(color == 1)
	{
		Move_Full_Forward();
		color = get_color_data();

		//Turn Left if color changes to not be yellow or white
		while(color != 1 && color != 2 && color != 5)
		{
			Move_Left_Forward();
			color = get_color_data();
		}

		//If we found the white, move through it
		while(color == 5)
		{
			Move_Full_Forward();
			color = get_color_data();

			//Turn left if the color changes to be not white or red
			while(color != 1 && color != 5)
			{
				Move_Left_Forward();
				color = get_color_data();
			}
		}
	}

	//Follow the yellow until it turns white
	while(color == 2)
	{
		Move_Full_Forward();
		color = get_color_data();
```

```
                //Turn right if color changes to be not yellow, red, or white
                while(color != 2 && color != 1 && color != 5)
                {
                        Move_Right_Forward();
                        color = get_color_data();
                }

                //If we found the red, move through it
                while(color == 1)
                {
                        Move_Full_Forward();
                        color = get_color_data();

                        //Turn right if the color changes to be not yellow or red
                        while(color != 2 && color != 1)
                        {
                                Move_Right_Forward();
                                color = get_color_data();
                        }
                }
        }

        //Update current area
        curr_area = 2;
    }

    //If in the build area and wanting to go to start - assuming that we're
starting on the yellow
    if(curr_area == 2 && desired == 0)
    {
        //Follow the yellow until it turns red
        while(color == 2)
        {
                Move_Full_Forward();
                color = get_color_data();

                //Turn right if color not yellow or red
                while(color != 2 && color != 1)
                {
                        Move_Right_Forward();
                        color = get_color_data();
                }
        }

        //Follow the red until it turns green
        while(color == 1)
        {
                Move_Full_Forward();
                color = get_color_data();

                //Turn right if the color is not red or green
                while(color != 1 && color != 4)
                {
                        Move_Right_Forward();
```

```
                        color = get_color_data();
                }
        }

        //Follow the green until it turns white
        while(color == 4)
        {
                Move_Full_Forward();
                color = get_color_data();

                //Turn left if the color is not green or white
                while(color != 4 && color != 5)
                {
                        Move_Left_Forward();
                        color = get_color_data();
                }
        }

        //Update current area
        curr_area = 0;

    }

    //If in the materials area and wanting to go to start - I don't think we're
ever going to need this case
    if(curr_area == 1 && desired == 0)
    {
        //Find the red, follow it to blue

        //Follow the blue until it turns yellow

        //Update current area
    }

    //If in the build area and want to go to materials
    if(curr_area == 2 && desired == 1)
    {
        //Follow the white until it turns yellow

        //Follow the yellow until it hits 2nd red
        int red = 0, last_color;
        while(color == 2)
        {
                Move_Full_Forward();
                last_color = color;
                color = get_color_data();

                //Turn right if color not yellow or red
                while(color != 2 && color != 1)
                {
                        Move_Right_Forward();
                        color = get_color_data();
                }

                //The first time we hit red, move over it
```

```
                        if(red == 0)
                        {
                                while(color == 1)
                                {
                                        Move_Full_Forward();
                                        last_color = color;
                                        color = get_color_data();

                                        //Turn right if color not yellow or red
                                        while(color != 2 && color != 1)
                                        {
                                                Move_Right_Forward();
                                                color = get_color_data();
                                        }
                                }
                                red = 1; //After we get through the red, we want the next
one
                        }
                }

                //Update current area
                curr_area = 1;
        }
}

void Grid_index(signed int x, signed int y, signed int z)
{
        unsigned long proximity, threshold = 2000;
        //NEED TO DETERMINE THRESHOLD
        int d_blues, c_blues = 0, d_red, c_red = 0, color, last_color, center = 0;

        color = get_color_data();

//Go x over (counting the blues)
        d_blues = x + 5; //Normalize the number of x (-5 is now 0)
        //while counted blues is greater than desired blues
        while(c_blues != d_blues)
        {
                Move_Half_Forward();
                last_color = color;
                color = get_color_data();

                if(last_color == 5 && color == 3)
                        c_blues++;
                //Turn right if we go over the edge
                while(color != 5 && color != 3)
                {
                        Move_Right_Forward();
                        color = get_color_data();
                }
        }

//Once the correct blue is reached, center in the blue
        while(color == 3) //Go through the blue
        {
```

```
		color = get_color_data();
		Move_Half_Forward();

		//Turn right if we go over the edge
		while(color != 5 && color != 3)
		{
			Move_Right_Forward();
			color = get_color_data();
		}
	}

	while(color == 5)//Follow the white until it changes to blue
	{
		color = get_color_data();
		Move_Quart_Forward();
		center++;

		//Turn right if we go over the edge
		while(color != 5 && color != 3)
		{
			Move_Right_Forward();
			color = get_color_data();
		}
	}

	center = center/2 + 1;

	//Move backwards to the center
	for(int i = 0; i < center; i++)
	{
		Move_Quart_Backward();
		Delay_ms(700, 0);
	}

	//Turn left
	for(int i = 5; i < 5; i++)
	{
		Move_Left_Forward();
		Delay_ms(700,0);
	}


//Move down, counting the reds, continuously checking proximity
	d_red = y + 5; //Normalized the number of y (-5 is now 0)
	while(c_red > d_red)
	{
		Move_Full_Forward();
		last_color = color;
		color = get_color_data();

		if(last_color == 0 && color == 1)
			c_blues--;

		//Turn Left if going off the edge
		while(color == 3)
```

```
		{
			Move_Left_Forward();
			color = get_color_data();
		}

	//If proximity senses something, change approach
		proximity = get_proximity_data();
		if(proximity > threshold) //if there is something in front of the robot
that does not allow it to move forward.
		{
			//Check to see if that one needs a block anywhere, record which
spaces are filled
			if(ideal_lower[c_red][c_blues] == true)
			{

			}
			//If it does not need a block, change approach
			else
			{
				if(x > 0) //If it's in the right hand plane
				{
					//Back up 2 spaces - make sure to account for edge
cases

					//Turn left

					//Move over 3 blue spaces - make sure to account for
edge cases

					//Follow blue to appropriate red space

					//Center in the red

					//Turn left

					//Move forward until correct space, checking
proximity

					//if space is filled, record the data

					//Make X, Y, Z equal to 0, so that the robot will
just leave the area
				}
				else //If it's in the left hand plane
				{
					//Back up 2 spaces - make sure to account for edge
cases

					//Turn right

					//Move over 3 blue spaces - make sure to account for
edge cases

					//Follow blue to appropriate red space
```

```
                                    //Center in the red

                                    //Turn right

                                    //Move forward until correct space, checking
proximity

                                    //if space is filled, record the data
                                    //Make X, Y, Z equal to 0, so that the robot will
just leave the area
                              }
                        }
                }
        }

//Place Block
        if(x != 0 && y != 0 && z != 0)
        {
                //Move down correct depending on level

        }
}

void Grid_exit()
{
        int color;
        color = get_color_data();

        //Back up until white is found
        while(color != 5)
        {
                Move_Full_Backward();
                color = get_color_data();
        }

        //Follow the white edge back to the yellow
        while(color == 5)
        {
                Move_Full_Forward();
                color = get_color_data();

                //Turn left if color changes to not be blue or red
                while(color != 5 && color != 3 &&  color != 1)
                {
                        Move_Left_Forward();
                        color = get_color_data();
                }
        }
}
void I2C_set_up()
{
        //The max frequency for the RGB color sensor is 400kHz
        //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

	I2C1_CR1 |= (1 << 7); //Disable Clock Streatching

}

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;

	int data1[1] = {0};

	//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 = (slave_addr << 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 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))
                                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)
                                I2C1_CR1 |= (1 << 9); //STOP
                }
        }
        }

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

        //Reset Stop Flag
        I2C1_CR1 &= ~(1 << 9);
        //Reset ACK Flag
        I2C1_CR1 &= ~(1 << 10);

        //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;
	}

	if((I2C1_SR1 & (1 << 8)) == (1 << 8))
		I2C1_SR1 &= ~(1 << 8);

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

	return success;
}

//The address of the proximity peripherial is 0x13
unsigned long get_proximity_data(void)
{
	bool success;

	int proximity_data[5] = {0, 0};

	int data[2];
  data[0] = 0x83;
  data[1] = 10;

	//Intialize the registers
	//IR LED Current
	success = master(0x13, 0, true, true, 2, data);
	//Proximity Rate
	data[0] = 0x82;
	data[1] = 1;
	success = master(0x13, 0, true, true, 2, data);
	//Interrupt Control
	data[0] = 0x89;
	data[1] = 1;
	success = master(0x13, 0, true, true, 2, data);

	//Communicate with the slave and start the proximity data read
	if(success == true)
	{
		data[0] = 0x80;
		data[1] = (1 << 3);
		success = master(0x13, 0, true, true, 2, data);
	}

	//There is 1 measurement done every 1.95 seconds, so wait 2
	Delay_us(2000,0);

	//If the proximity data read is finished, read it
	data[0] = 0x87;
	success = master(0x13, 0, true, true, 1, data);
```

```
	success = master(0x13, 1, true, true, 2, proximity_data);

	//Return the number of the proximity
	unsigned long proximity = (proximity_data[0] << 8) + proximity_data[1];

	return proximity;
}


//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 send_data[2], color_data[6] = {0, 0, 0, 0, 0, 0}, color = 0;

	//Communicate to the slave to start the RGB sensing sequence
	send_data[0] = (1 << 7);
	send_data[1] = 3;
	success = master(0x29, 0, true, true, 2, send_data);

	//Wait for the RGB value to be taken (MESS WITH THIS - the wait time is 2.4 ms)
	Delay_ms(700, 0);

	//If previous was successful, ask for data
	if(success == true)
	{
		//Intiate a combined protocol interaction (see slave documentation for
details
		send_data[0] = (1 << 7) + (1 << 5) + 0x16; //Send the command code
(command + auto-increment + RDataL addr)
		success = master(0x29, 0, false, true, 1, send_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)
	{
		send_data[0] = (1 << 7);
		send_data[1] = 0;
		success = master(0x29, 0, true, true, 2, send_data);
	}

	//Process the collected 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;

 //Percentage of each value, with 2 being 100%
 //Each value will be either 2(max), 1(mid), and 0(none)
 red = round(((red_raw/max)*2));
 green = round((green_raw/max)*2);
 blue = round((blue_raw/max)*2);

 bool high_low;

 if(blue_raw > 1000 && red_raw > 1000 && green_raw > 1000)
       high_low = true;
 else
       high_low = false;

 //Error = 0
 //Red = 1
 //Yellow = 2
 //Blue = 3
 //Green = 4
 //White = 5

 //Determine if it's red
 if(red == 1 && green == 0 && blue == 0)
       color = 1;
 if(red == 2 && green == 1 && blue == 1)
       color = 1;
 if(red == 2 && green == 0 && blue == 0)
       color = 1;
 if(red == 2 && green == 1 && blue == 0)
       color = 1;

 //Determine if it's yellow
 if(red == 2 && green == 2 && blue == 1 && high_low == true)
       color = 2;
 if(red == 2 && green == 2 && blue == 0 && high_low == true)
       color = 2;
 if(red == 1 && green == 1 && blue == 0 && high_low == true)
       color = 2;

 //Determine if it's blue
```

```
	if(red == 0 && green == 0 && blue == 1)
		color = 3;
	if(red == 1 && green == 2 && blue == 2)
		color = 3;
	if(red == 0 && green == 2 && blue == 2)
		color = 3;
	if(red == 0 && green == 1 && blue == 1)
		color = 3;
	if(red == 1 && green == 1 && blue == 2)
		color = 3;
	if(red == 0 && green == 0 && blue == 2)
		color = 3;
	if(red == 0 && green == 1 && blue == 2)
		color = 3;

	//Determine if it's green
	if(red == 0 && green == 1 && blue == 0)
		color = 4;
	if(red == 1 && green == 2 && blue == 1)
		color = 4;
	if(red == 0 && green == 2 && blue == 0)
		color = 4;
	if(red == 1 && green == 2 && blue == 0)
		color = 4;
	if(red == 0 && green == 2 && blue == 1)
		color = 4;

	//Determine if it's white
	if(red == 2 && green == 2 && blue == 2 && high_low == true)
		color = 5;

	//Determine if it's black (error)
	if(red == 2 && green == 2 && blue == 2 && high_low == false)
		color = 0;

	return color;
}

void ADC_Setup()
{
	RCC_APB2ENR |= (1 << 9); //Enable ADC2

	ADC2_CR2 |= 2; //Continous Conversion Mode
	ADC2_CR1 |= (1 << 8); //Scan Mode Enabled Page 385 Ref Manual
	ADC2_SQR1 |= (3 << 20); //There will be 3 conversions in each cycle.
	ADC2_SQR3 |= 14; //1st conversion will be on channel 7 (PA7)
	ADC2_SQR3 |= (15 << 5); //2nd conversion will be on channel 2 (PA2)
	ADC2_SQR3 |= (7 << 10); //3rd converison will be on channel 3 (PA3)
	ADC_CCR |= 0x30000; //Divide the clock by 8
	ADC2_CR2 |= (1 << 8); //Direct memory access mode
	ADC2_CR2 |= (1 << 9); //DMA requests will be issued as long as data is
converted and DMA = 1

	//Enable DMA2
	RCC_AHB1ENR |= (1 << 22);
```

```
	//Reset the EN bit in the DMA_SxCR register
	DMA2_S2CR &= ~1;

	//ADC2 is Channel 1 of Stream 2 or 3
	//Set the peripheral port register address in the DMA_SxPAR register. The data
is moved to this address from
	// the peripheral port after the peripheral event.
	DMA2_S2PAR = 0x4001214C; //ADC2_DR
	//Set the memory address in the DMA_SxMA0R register. The data is written to
this memory after the peripheral
	// event.
	DMA2_S2M0AR = (unsigned long) ADC_data;
	//Configure the total number of data items to be transfered in the DMA_SxNDTR
register.
	DMA2_S2NDTR = 3; //3 data transfers
	//Select the DMA channel using CHSEL[2:0] in the DMA_SxCR register
	DMA2_S2CR |= (1 << 25); //ADC2 is channel 1
	//Configure the stream priority using the PL[1:0] buts in the DMA_SxCR register
	DMA2_S2CR |= (3 << 16);
	DMA2_S2CR |= (1 << 13); //Memory data size set to 16-bits, since the ADC does
12-bit conversions
	DMA2_S2CR |= (1 << 11); //Peripheral data size set to 16-bits
	DMA2_S2CR |= (1 << 10); //Memory address pointer will be incremented after each
data transfer
	DMA2_S2CR |= (1 << 8); //Enable Circular Mode
	//DMA2_S0CR |= (1 << 5); //The peripheral is the flow controller
	//Configure the FIFO usage
	//Configure the data transfer direction - Default is peripheral to memorr
	//Activate the stream, as soon as it is enable with DMA_SxCR, it can serve any
DMA request from the
	// peripheral
	DMA2_S2CR |= 1;
	ADC2_CR2 |= 1; //Enable the ADC channels
	//pg 224

	ADC2_CR2 |= (1 << 30); //Start conversion of regular channels, the bit will be
cleared whenever the
				// conversion starts.
	//Make sure to check for the OVR bit to be set, because that means that the
data was lost. The steps
	// to recover are on pg 369.


}

bool check_nearby()
{
	int threshold = 2000;
	bool nearby = false;

	for(int i = 3; i < 3; i++)
	{
		if(ADC_data[i] >= threshold)
			nearby = true;
```

```
	}

	return nearby;
}

void Communication(bool area_switch, bool wake_up)
{
	int blinks = 0;
	bool control = false;
	//Make sure all movement has stopped

	//Blink to begin communication

	//Generate a random number between 1 and 10
	int num = (rand() % (10 + 2)) + 1;
	for(int i = 0; i < num; i++)
	{
		//Blink on

		//Check for another blinking - if the blinking stops we're in control, if
not the other robot is
		//If there is no blinking detected on 1, restart the loop

		//Wait .5 seconds
		Delay_ms(5000, 0);

		//Blink off

		//Wait .5 seconds
		Delay_ms(5000, 0);
	}

	//If we're in control
	if(control == true)
	{
		//Transmit messages
		if(area_switch == true)
		{
			//Blink once

			//Wait a Second for the other robot to receive the message and
move backwards
		}
		else
		{
			if(wake_up == true)
			{
				//Blink twice

				//Wait a Second for the other robot to receive the message
			}

			//Blink three times

			//Wait a second for the other robot to recieve the message
```

```
                        //Blink the number of completed areas

                }

                //Either end transmission or ask the other robot if they have anything to
say
                if(area_switch == true)
                {
                        //End communication
                }
                else if(wake_up == false)
                {
                        //Blink 4 times to ask the other robot if they have any messages

                        //Recieve blinks - if they blink once then they have something to
say
                }
        }

        //If we're not in control
        if(control == false)
        {
                //Receieve Messages

                //If the other robot asks, send them our messages

        }
}

void PWM_Setup()
{
        //Enable TIM3
        RCC_APB1ENR |= (1 << 1);

        //Set up Claw PWM - Claw PA6
        //Channel 1 is in output capture mode by default
        TIM3_CCMR1 |= (6 << 4);
        TIM3_CCMR1 |= (1 << 3); //Preload enable, allows initial set up
        TIM3_CCER |= 1; //Signal will output on pin

        //Set up Horizontal PWM - PB0
        //Channel 3 is in output capture mode by default
        TIM3_CCMR2 |= (6 << 4);
        TIM3_CCMR2 |= (1 << 3); //Preload enable, allows initial set up
        TIM3_CCER |= (1 << 8); //Signal will output on pin

        //Set up Vertical PWM - PB1
        //Channel 4 is in output capture mode by default
        TIM3_CCMR2 |= (6 << 12);
        TIM3_CCMR2 |= (1 << 11); //Preload enable, allows initial set up
        TIM3_CCER |= (1 << 12); //Signal will output on pin

        //Set the Period to be 50 Hz
        TIM3_PSC = 31; //Set the prescaler to be 31 ((16MHz/(1 + 32)) = 500Hz)
```

```
	TIM3_ARR = 10000; //Set the auto reload to be 100000 so it autoreloads every 20
ms (which gives us
				// the required 50 Hz
	TIM3_CCR1 = 10;
	TIM3_CR1 |= (1 << 7);
	TIM3_EGR |= 1; //Event generation
	TIM3_CR1 |= 1; //Starts counting
}

void Claw_Motor_Control(double percent)
{
	//500 is fully open, 1000 is fully closed
	double real_percent = (100 - percent)/10;

	//To change the duty cycle change the CCR1
	TIM3_CCR1 = (500*real_percent) + 500;

	//Give it some time to change
	Delay_ms(500, 0); //Wait for .5 second <MIGHT HAVE TO CHANGE>

}

void PWM_Capture_Setup(void)
{
	//Activate TIM4
	RCC_APB1ENR |= (1 << 2);

	//Input Capture Mode
	TIM4_CCMR2 |= 1;
	TIM4_CCER |= (1 << 8);

	//Rising and Falling Edge Detection
	TIM4_CCER |= (1 << 9);
	TIM4_CCER |= (1 << 11);

	//Set up interrupt
	NVICISER0 |= (1 << 30);
	TIM4_DIER |= (1 << 3); //Allows Capture/Compare on Channel 3 to interrupt

	//TIM4_PSC = 1; //Makes the clock 8 MHz - 1 tick every 12 us
	//TIM4_ARR = 65535; //Makes the values able to be stored in an int

	TIM4_CR1 |= 1; //Starts counting
}

void TIM4_IRQHandler(void)
{
	int Timing_data[6];
	double raw_duty;

	//Need to account for both channels
	if((TIM4_SR & (1 << 3)) == (1 << 3)) //Channel 3
	{
		if((GPIOB_IDR & (1 << 8)) == (1 << 8)) //If the data is high - rising
edge
```

```
                Timing_data[0] = TIM4_CCR3;
            else //Data is low - falling edge
                Timing_data[1] = TIM4_CCR3;

            raw_duty = (Timing_data[1] - Timing_data[0]); //Current duty cycle =
falling - rising
            if (Timing_data[0] > Timing_data[1]) //If current is not a valid number
                raw_duty = 0;

            //With the current timing we're getting 1 tick every .125us - 8800
ticks/period
            duty_PB8 = ((double)raw_duty/17600) * 100; //Hold the duty cycle from 1 -
100

            if(duty_PB8 > 100)
                duty_PB8 = 0;
        }
}
void Horz_Scanning()
{
        double change, desired, left_cycles = 0, right_cycles = 0;
        double left_duty, right_duty;

        unsigned long proximity;
        proximity = get_proximity_data();

        TIM3_CCR3 = 720;

        //Move Left (CW) until the baseline is reached.
        while(proximity > 3400)
        {
            //if motor completes turn, increment cycles
            if((duty_PB8 >= 95.0 || duty_PB8 <= 5.0) && duty_PB8 != 0)
                left_cycles = 1;

            //Determine proximity
            proximity = get_proximity_data();
        }

        //Nab the duty cycle at this point
        left_duty = duty_PB8;
        while(left_duty == 0)
            left_duty = duty_PB8;

        //Switch rotation when the edge of the block has been found
        TIM3_CCR3 = 770;

        while(proximity <= 3800)
            proximity = get_proximity_data();

        //Move right (CCW) until the baseline is reached
        while(proximity >= 3400)
        {
            //if motor completes turn, increment cycles
            if((duty_PB8 >= 95.0 || duty_PB8 <= 5.0) && duty_PB8 != 0)
```

```
                right_cycles = 1;

        //Determine proximity
        proximity = get_proximity_data();
    }

    right_duty = duty_PB8;
    while(right_duty == 0)
        right_duty = duty_PB8;

    if(right_cycles > 0 && left_cycles > 0)
    {
        right_duty = 100 - right_duty;
        left_duty = 100 - left_duty;
    }

    //Stop rotation when the edge of the block has been found
    TIM3_CCR3 = 750;

    Delay_ms(500, 0);

    //Center the claw on the block

    desired = ((double)left_duty + (double)right_duty) / 2.0;

    change = duty_PB8;
    TIM3_CCR3 = 720;

    while(change > (desired + .1) || change < (desired - .1))
    {
        change = duty_PB8;
        if(change == 0)
            change = duty_PB8;
    }

    TIM3_CCR3 = 750;

}

void Vert_Scanning()
{
    unsigned long proximity;
    proximity = get_proximity_data();

    TIM3_CCR4 = 710;

    //Move up (CW) until the top is reached
    while(proximity > 3400)
    {
        //Determine proximity
        proximity = get_proximity_data();
    }

    //Move down a bit so the block is in front
    TIM3_CCR4 = 770;
```

```
	Delay_ms(1000,0);

	TIM3_CCR4 = 750;

}

void Grab_Block()
{
	TIM3_CCR1 = 510;

	//Vertical Scan
	Vert_Scanning();

	//Horizontial Scan
	Horz_Scanning();

	//Move down a bit so that the claw is around the block
	TIM3_CCR4 = 770;
	Delay_ms(2000,0);

	TIM3_CCR4 = 750;

	//Close the claw
	TIM3_CCR1 = 1000;

	Delay_ms(100,0);

	TIM3_CCR1 = 750;

	//Move up a bit
	TIM3_CCR4 = 710;
	Delay_ms(200, 0);
	TIM3_CCR4 = 750;
}
```

References

[1] Actobotics Parallel Gripper Kit A - https://www.servocity.com/parallel-gripper-kit-a

[2] RGB color sensor - https://cdn-shop.adafruit.com/datasheets/TCS34725.pdf

[3] Proximity sensor - https://cdn-shop.adafruit.com/datasheets/vcnl4000.pdf

[4] STM32 Microcontroller - https://www.mouser.com/ProductDetail/STMicroelectronics/NUCLEO-F030R8?qs=fK8dlpkaUMvL9GSuoYnNYw%3D%3D&gclid=Cj0KCQjw2IrmBRCJARIsAJZDdxCNfIHpPdzHj5rrwE6ZNXR-YmluyxFdeAt45M5K0QxFfj8fUYVMoEYaAvIIEALw_wcB