MTE100/MTE121 FINAL PROJECT

ASL Robotic Hand

Group 8-6
Date: 12/05/23

Evie Bouganim 21063520
Khushi Patel 21065236
Meghan Dang 21076546
Rubie Luo 21066766

Abstract

In North America, over half a million people use American Sign Language (ASL) as their main method of communication, however, the country's population is over 579 million people [1]. The team's project, *Thing*, whose name is inspired by the "Addam's Family" is a robotic hand that can sign ASL letters, serving as an instructional aid and interpreter. According to the constraints from the MTE 100 teaching team, the goal is to create a robotic system using the LEGO EV3 Mindstorm software and hardware. *Thing* is a combination of LEGO as well as 3D printed components, and fishing line. Each finger is individually actuated and contains one degree of freedom and three joints. The robot operates using six motors, a touch sensor, and a gyroscope. During operation, *Thing* will take input from a downloaded file, filter for profanity, sign each word, and "wave" goodbye following a "high five" from the user. Through providing a kinesthetic and interactive learning experience, *Thing* encourages users to learn the basic hand movements and alphabet required for ASL.

Acknowledgements

Prof. Nassar, thank you for being the goat.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

Each year, the first year Mechatronics Engineering students at the University of Waterloo are expected to produce a robot using LEGO EV3 Mindstorm kits provided in class. The team was required to identify a problem and design a robotic system that uses sensors, and motors, and receives input from a source. Following a five-week developing period, the team presented their project, *Thing*, at the "Demo Day" showcase.

1.2 Problem Description

In North America, over half a million people use American Sign Language (ASL) as their means of communication, making it the third most common language in the United States [2]. With the continent's population approaching 530 million people, this leaves approximately 99.9% of people incapable of communicating in ASL. Despite a variety of online instructional aids, they often lack the ability to provide kinesthetic feedback while learning, and hiring interpreters can become expensive.

2 Scope

Thing's primary function is to sign words in ASL letter by letter, serving as either an instructional aid or interpreter for users.

2.1 Robot Specifications

Thing uses five large EV3 servo motors, a medium EV3 servo motor, motor encoders, a touch sensor, a gyroscope, and file input. In order to determine which letter to sign in ASL, the robot must receive a string from a file downloaded to the EV3 brick. To sign, each finger is individually actuated by a large motor and the medium motor controls the roll joint. According to each letter's sign, the motors move to preset encoder values to replicate the position. See **Figure 1**: ASL Letter Chart [3, Fig. 1]. In addition to encoder values, each finger has a hard stop, preventing it from bending backwards. At the end of the program, *Thing* will wait until the touch sensor is pressed before "waving" goodbye and terminating the program.

2.2 Scope Iterations

Initially, *Thing*'s sole task was to convert inputted characters into sign language. As the design progressed, the team widened the scope to include a profanity checker for file input, the capability to sign multiple words, and implement a "wave" feature as part of the shutdown procedure.

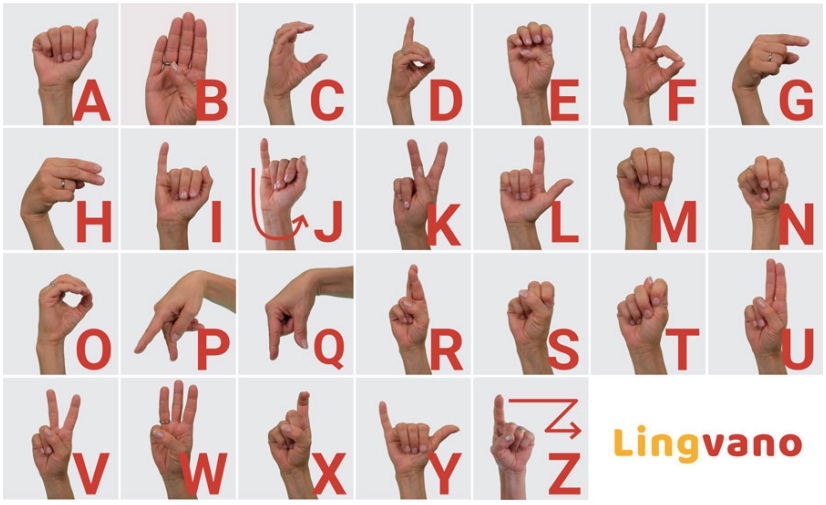


Figure 1: ASL letter chart

2.3 Functionality

Start-up

The robot begins with each finger in a vertical position with the palm parallel to the base of the structure (facing forwards). See **Figure 2**: Robot Home Position. Prior to starting the program, a user needs to download a text file to the EV3 with the desired word(s) to sign. If there are multiple words, they can be separated by spaces or special characters. Once the file is loaded, the robot will wait until the enter button is pressed before executing the main program.

Operation

Following the press of the enter button, the robot filters the file input and determines whether it includes profanity. If it does, *Thing* will not sign the word; Otherwise it will sign each letter in the word, waiting two seconds in between words. For each letter, the corresponding fingers will move up and down to preset encoder values, as well as rotating the wrist if required.

Termination

After all of the letters have been signed, *Thing* will wait for the user to give it a "high five." By pressing the touch sensor, the hand will wave the fingers up and down then terminate the program.

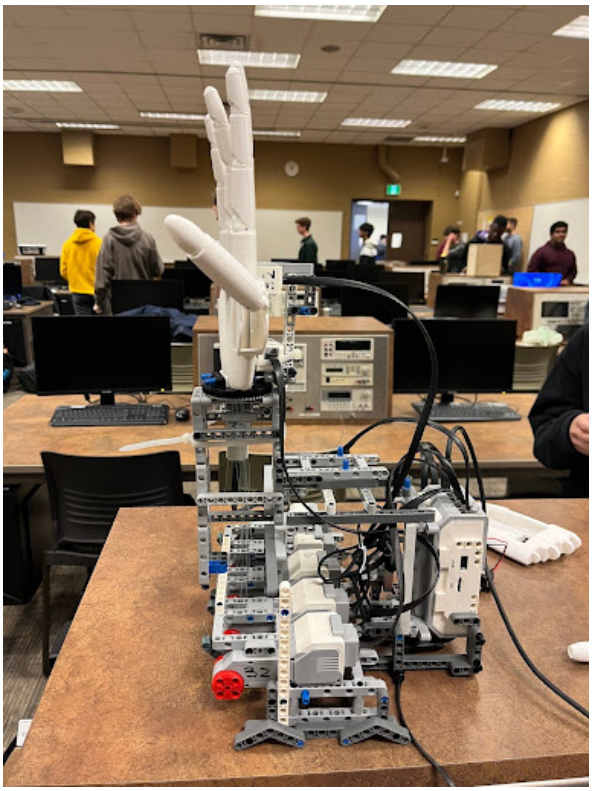


Figure 2: Robot Home Position

3 Design Considerations

3.1 Constraints

Since *Thing* is designed for the MTE 100 final project, there are several constraints: the project has to use the LEGO EV3 Mindstorm motors and sensors from the kit provided in class, the robot must be programmed using RobotC, it has to be to be completed in the five week design window, and follow the design requirements provided by the teaching team. The class requirements include using at least 3 motors, 4 distinct types of input, timers, loops, conditional statements, and six non-trivial functions.

3.2 Criteria

The principle criteria of the project is that *Thing* is capable of signing the entire alphabet such that an ASL user can interpret the words. To accomplish this, the team created a list of criteria that would make an ideal robotic hand, then modified it to realistically fit the project constraints.

3.2.1 Ideal

To produce an ideal robotic hand, the mechanical criteria requires the fingers to be individually actuated with three joints, the hand to approximately match average human hand size, have the

ability to cross fingers with side-to-side actuation, and have pitch, yaw, and roll freedom of motion.

3.2.2 Realistic

In accordance with the constraints and as the project progressed, the team modified the ideal criteria to a more achievable list: each finger is individually actuated with three joints, the hand size approximately matches a human hand, and the hand can rotate 90 degrees clockwise. The side-to-side actuation, pitch, and yaw are not realistic criteria given the time and resource constraints.

4 Mechanical Design and Implementation

4.1 Overall Design

Thing is made up of four primary subsystems: the 3D-printed hand, LEGO integration, support structure, and EV3 brick. The hand is connected to the turntable that is fastened to the main support structure. See **Figure 3**: Completed Version of *Thing*.

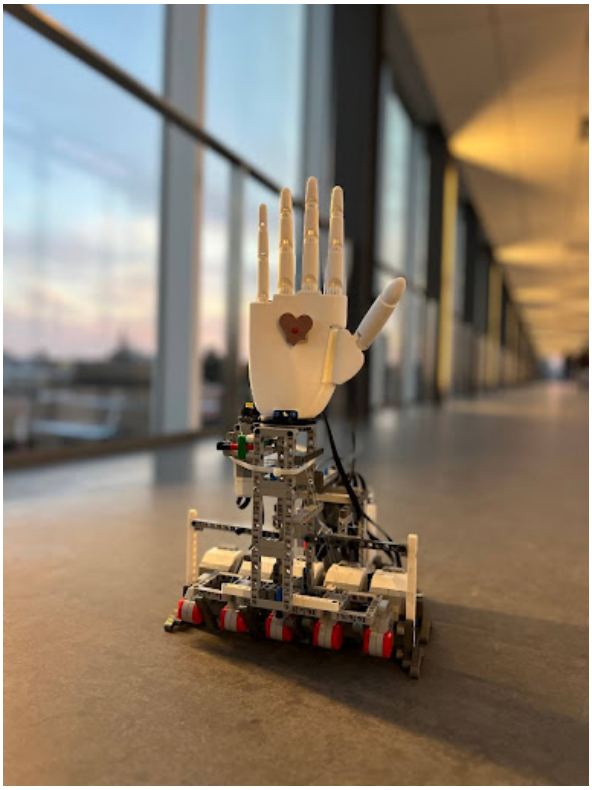


Figure 3: Completed Version of *Thing*

4.2 Hand Design Process

The design of *Thing* was inspired by prostheses currently on the market and by studying the human hand signing ASL letters. The human hand alone has 27 degrees of freedom (DOF) [4] which is

extremely costly and technically difficult to mimic. Despite advances in assistive technology, no current prosthesis can match the dexterity, flexibility and fluidity of the human hand. *Thing* is designed with one DOF per finger; Specifically, the MCP 3rd DOF which is the bending motion at the base of the finger. See **Figure 4**: Degrees of Freedom in the Finger [5].

In order to maintain movement in the PIP and DIP, the finger possesses a joint that moves in conjunction with the MCP. All hand designs were completed from scratch using SOLIDWORKS.

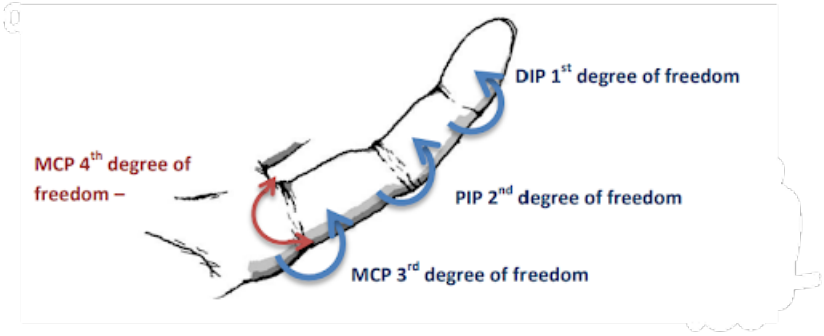


Figure 4: Degrees of Freedom in the Finger

4.2.1 Prototype 1

The first design has a cord running through the front of the finger and an elastic through the back. The cord is fastened to the motor, while the elastic is fastened to the largest finger segment. See **Figure 5**: Finger Design 1. The first version also has the motors attaching directly below each finger within the palm. With the actuator that close to the hand, it would be easy to tension the cord and provide a cleaner and more aesthetic design.

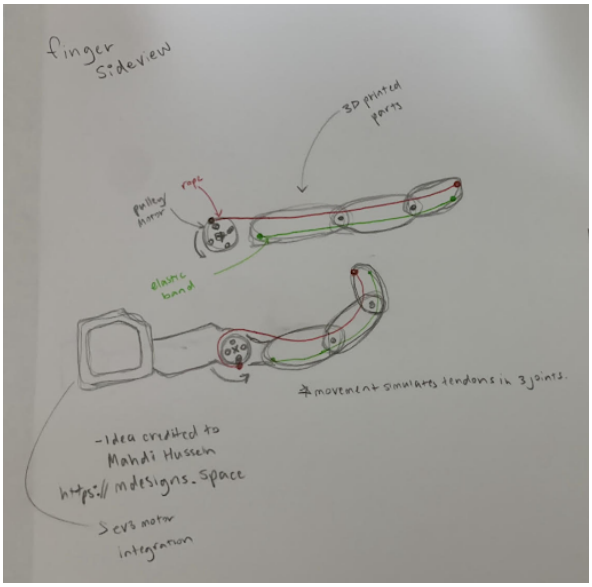


Figure 5: Finger Design 1

This concept was discarded upon consideration of the size of the motors. Side by side, four of these motors measure over 6", whereas an average palm is approximately 3".

4.2.2 Prototype 2

Due to the size of the motors, the team moved towards a design wherein the motors would be embedded in an external base. The second design consists of four fingers with three pin joints, offering a simple finger and palm design. See **Figure 6**: Prototype 2 CAD.

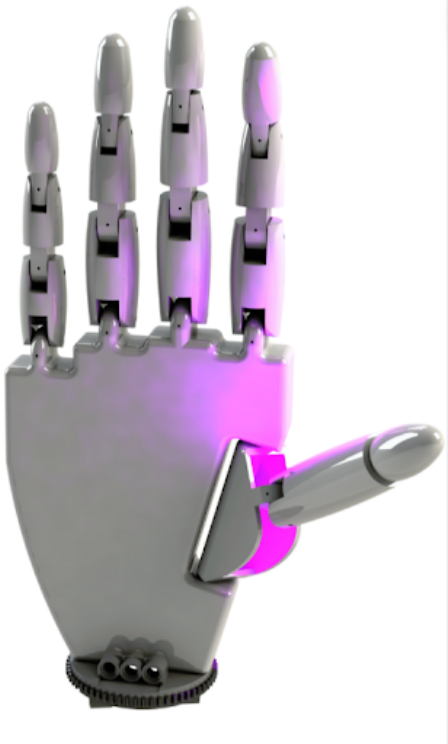


Figure 6: Prototype 2 CAD

After prototyping the first design, the team concluded that the fingers actuated smoother using the fishing line as opposed to the elastic. See **Figure 7**: Prototype 2 Finger Design. Each finger has fishing line running through the front and back, allowing it to bend up and down with variable string tension. As the motor turns, the direction of the cord is pulled, while the other end slacks, pulling the finger to the desired position.

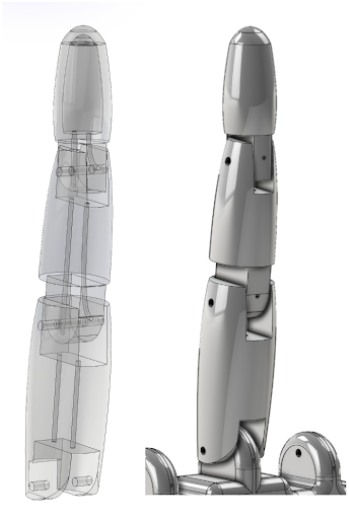


Figure 7: Prototype 2 Finger Design

The finished mechanical prototype is seen in **Figure 8**: Manufactured Prototype 2. While the second prototype exceeded the minimum viable product and completed all of the required mechanical tasks, the team desired a higher degree of fluidity of motion. The MCP joint was often prone to wobbling from side to side, and the little clearance at the joint created increased amounts of friction when the fingers changed position.

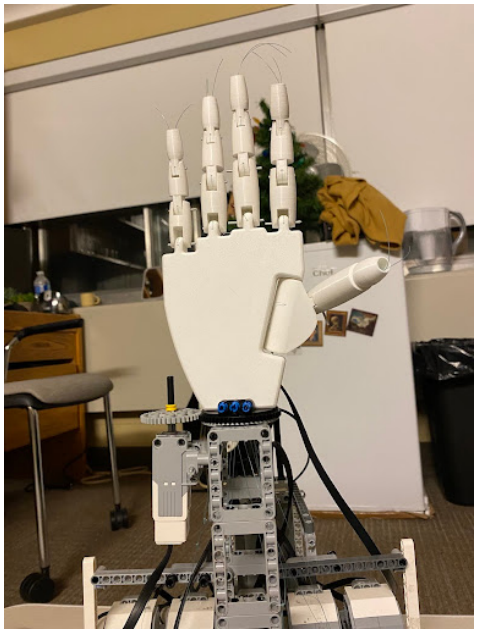


Figure 8: Manufactured Prototype 2

4.2.3 Prototype 3

Subsequently, the final design features three ball joints for each finger as opposed to the previous pin joints. While it is considerably harder to design, it can complete the necessary tasks with increased fluidity, reduced friction, and better tolerances. The joints are designed with a clearance of 0.2mm and are printed at a 0.16mm tolerance. In addition, the prototype 3 palm is surface modelled to increase the hand's realistic appearance and contains a slot for the touch sensor. See **Figure 9**: Prototype 3 CAD.

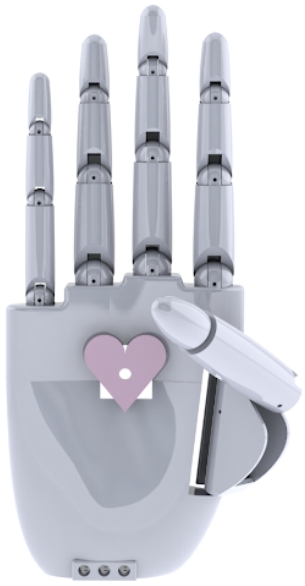


Figure 9: Prototype 3 CAD

The third prototype also includes small mounting holes at the base of the palm which attach the hand to the LEGO turntable and ten, 1 mm holes at the base of the palm to separate each cord as they are strung to the motors. During the prototyping phase, the team discovered that using 3D filament as pins for the joints functioned better than any fasteners both custom and on the market.

The finished mechanical prototype is seen in **Figure 10**: Manufactured Prototype 3.

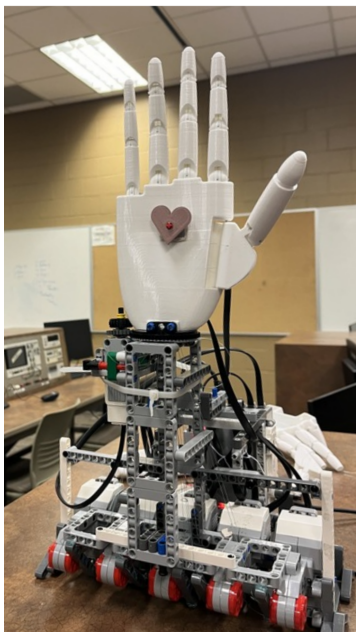


Figure 10: Manufactured Prototype 3

4.3 3D Printing

In order to meet the design criteria, the team decided to 3D print the parts for the hand. Keeping time and resource constraints in mind, the team brought Rubie's Creality Ender 5 3D printer to Evie's dorm room for the duration of the project. This allowed the team to have efficient, all hours printing without the additional cost of the University's printers.

4.3.1 Print Settings

The team used UltiMaker Cura (See **Figure 11**: STL File for Palm Print) to generate GCODE for all of the parts for the hand, printing with generic PLA, polylactic acid, filament. For the printing of the second prototype, the team used a medium quality print at a 0.2mm tolerance. For the third prototype, the team changed the print settings to dynamic quality at a 0.16mm tolerance to improve the clearance issues at the finger joints found in the testing phase of the second prototype. The finger joints have 0.2mm of clearance, yet despite the tight tolerance, the joint is free-spinning. The smallest printed feature are the holes that run inside the finger joints for the fishing line. Each hole is 1mm in diameter, the optimal size for 10lb fishing line.

In order to optimize time spent printing, the longer prints such as the palm were printed overnight, while the fingers were printed during the day. This enabled the team to assemble and test the fingers as the next round of prints were pending. In total, the team spent upwards of 80 hours of printing between printer setup, 2 fully printed prototypes, multiple test cases, and error avoidance.

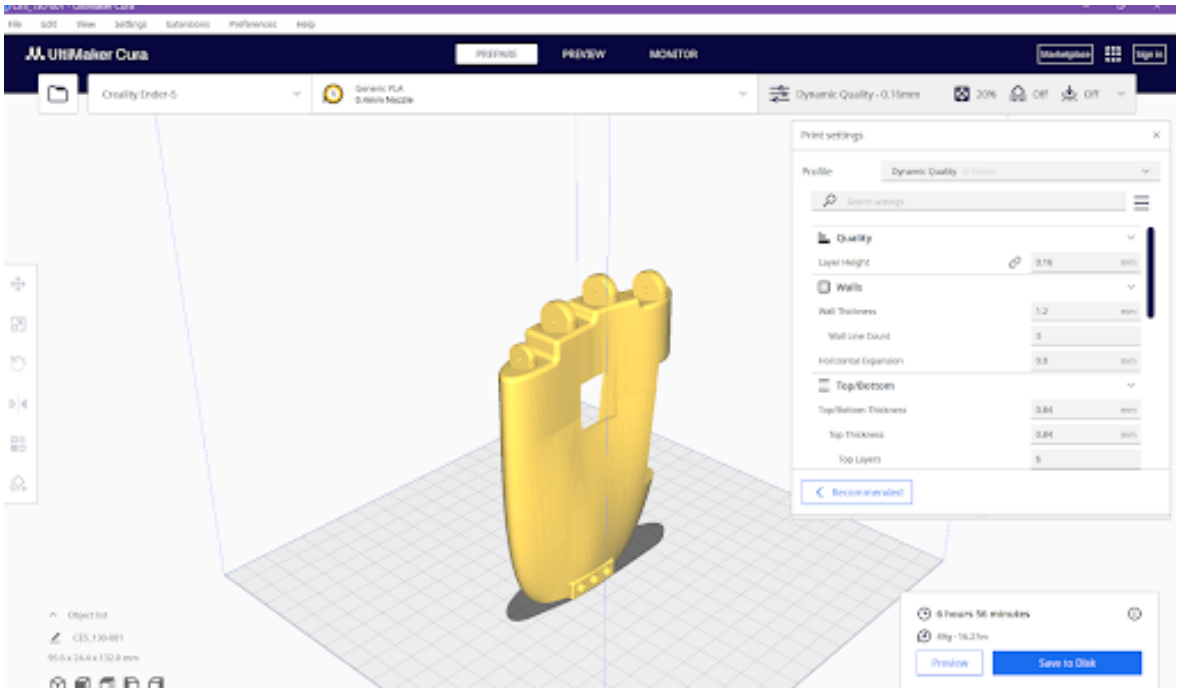


Figure 11: STL File for Palm Print

4.3.2 OctoPrint

Since the project required a lot of printing, the team invested time into setting up a print server, OctoPrint, using a Raspberry Pi 4. This enabled the team to upload their files to the printer without touching the SD card, better monitor the temperature, view print times, receive emails when the prints were completed, and install an intuitive AI to pause prints and notify the team if the printer runs out of filament or if a print failed.

4.3.3 OctoEverywhere

The main limitation of Octoprint is that the server functions through the Raspberry Pi communicating with a computer on a shared network. Since the team are students in residence, the network is the public University of Waterloo eduroam. Thus, the assigned IP address of the Pi is masked. To overcome this, the team connected the Pi and computer to a router, wired as to comply with university IT rules, so that the IP address was accessible. This allowed the print server to run while both devices were connected to the router.

The router solved the IP address problem, however, when printing all day, the team needed the ability to start printing remotely. The only way to access the server from different networks was to use a portal/VPN, OctoEverywhere, that both the printer and external devices could access remotely. Using both the print server and VPN, the team was able to print continuously throughout the day as the next round of CAD was completed and ready to print, drastically reducing manufacturing time. See **Figure 12**: Network Diagram for 3D Printing. Additionally, team members could view print times and live updates from the mobile app.

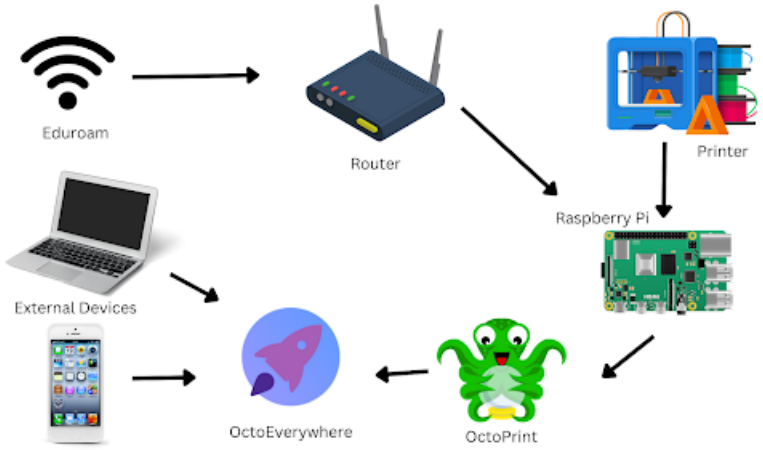


Figure 12: Network Diagram for 3D Printing

4.4 Lego Integration

Once the printing was complete, the fingers needed to actuate using the LEGO motors. The biggest constraint with LEGO integration was ensuring that the string was accurately and consistently tensioned across all fingers. If they were not, the finger would not flex far enough down, or the encoder limit value would never be met.

4.4.1 Version 1

The first tensioning prototype was fastening the fishing line to small LEGO pulleys attached to the motor shafts. See **Figure 13**: LEGO Pulley. It worked well, however the team was limited by the amount of pulleys provided in the Mindstorms kit.

Figure 13: LEGO Pulley

4.4.2 Version 2

Following the success of the original LEGO pulleys, the team decided to replicate them using SOLIDWORKS, and 3D print them instead. See **Figure 14**: 3D Printed Pulley. After each pulley was printed and multiple test cases performed, the team found that over time, the strings would tangle, or it would lose tension if the string rolled off the pulley and onto the shaft.

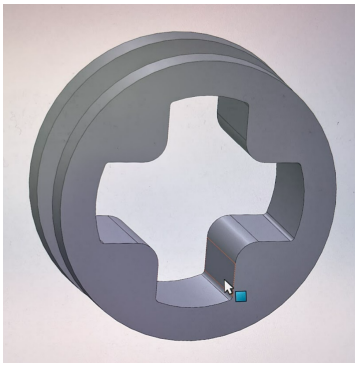


Figure 14: 3D Printed Pulley

4.4.3 Version 3

Since the string was slipping off the pulleys, the team decided to increase the size of the pulleys. The LEGO kit provided did not include large pulleys, however the wheel hubs with the rubber exterior removed functioned as a good substitute. See **Figure 15**: Motor Base with Wheel Hubs. After testing this new design, it was found that the string tied around the wheel hub would shift up and down, making it difficult to properly tension. For the mechanical review with the MTE 100 teaching assistants, the team temporarily used sticky tack to secure the strings, however, this was not a viable long-term solution.

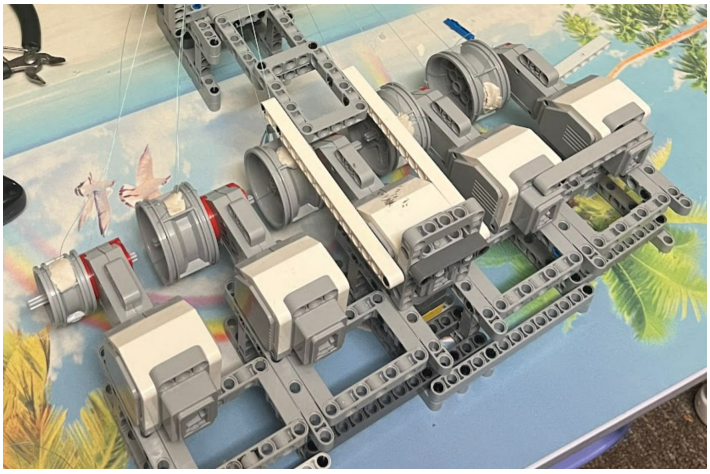


Figure 15: Motor Base with Wheel Hubs

4.4.4 Version 4

After much deliberation, the team had the idea to use LEGO beams instead of a pulley system. The beam functions similarly to a teeter-totter: with a string tied to each end, when the beam tips vertically, one string is fully tensioned, while the other is slack. The 7-hole LEGO beam maintained tension after multiple rounds of testing. See **Figure 16**: LEGO Beam Motor Base.

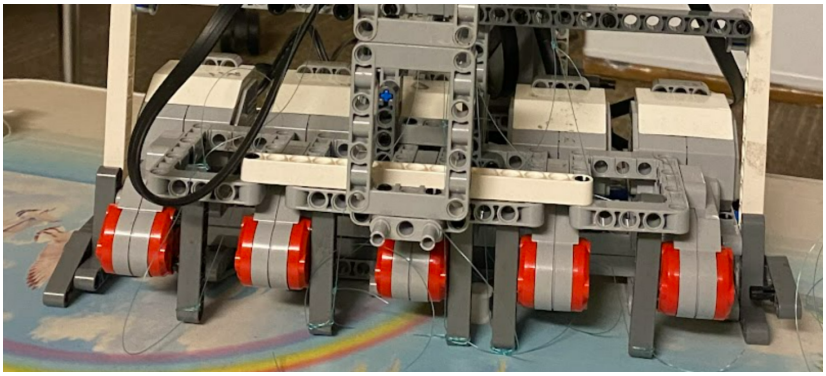


Figure 16: LEGO Beam Motor Base

4.5 LEGO Support Structure

In order to mount the hand on top of the motors, incorporate a turntable, and have a strong base, a LEGO support structure is required.

4.5.1 Version 1

The initial design featured three elements, the motor structure, hand support structure, and EV3 block as shown in **Figure 17**: Initial LEGO Support Structure. This first iteration had all of the key subsystems separated. With testing, the team observed the strings pulling or pushing the entire hand rather than moving the fingers. Even when the structure was secured, having the hand positioned in front of the motors placed the strings at an angle that increased the amount of force required to move each finger. This prototype proved to be inaccurate and difficult to test.

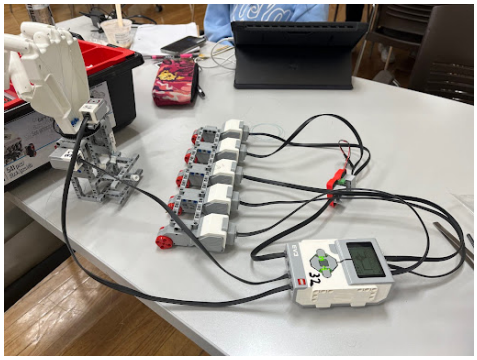


Figure 17: Initial LEGO Support Structure

4.5.2 Version 2

The final prototype of the support structure consists of the hand mounted to the turntable that sits atop a LEGO tower fastened to the motor base. See **Figure 18**: Final Support Structure Design. The strings from the fingers feed through the hole in the center of the LEGO turntable, and tie to each LEGO beam. Placing the hand on top of the motors allows for smoother actuation from the improved tensioning angle. The team also found through testing that the fingers reached their encoder targets more accurately.

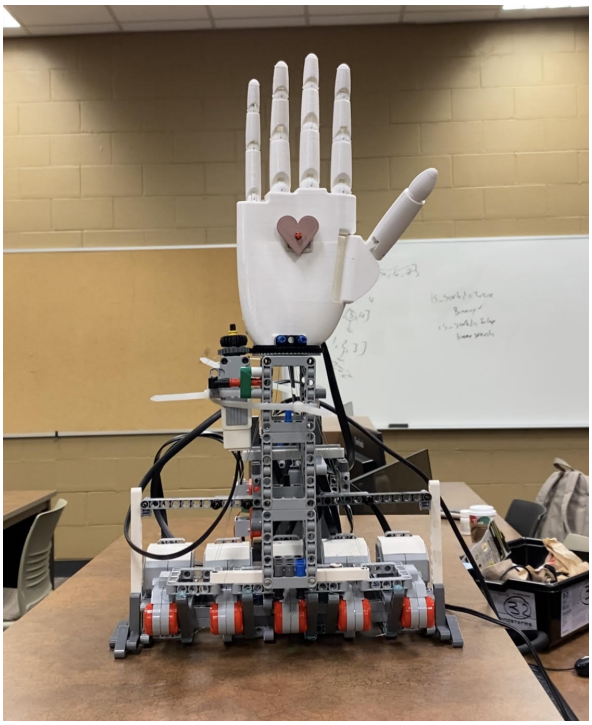


Figure 18: Final Support Structure Design

4.6 EV3 Placement

The team decided to include the EV3 brick onto the main structure; By increasing the weight of the LEGO base, the motors stay fixed in position instead of lifting up when the string is tensioned. Additionally, additional support beams and columns were added to either side of the motor structure to increase the stability of the structure and reduce flexing during operation.

4.7 Sensor Attachment

Since *Thing* is designed to be an interactive robot, the team placed emphasis on sensor attachment in order to promote user accessibility.

4.7.1 Touch Sensor

Halfway through the design process, the team decided to implement the use of a touch sensor, so that *Thing* could "high-five" its users. This created a constraint of making the hand and touch sensor mounting withstand the force of a human hand. As such, the touch sensor mounts to a support bracket located inside of the palm. See **Figure 19**: Touch Sensor Mounting. In addition to the bracket, the team used sticky tack to fasten and fill the gaps created between the touch sensor and the curvature of the palm.

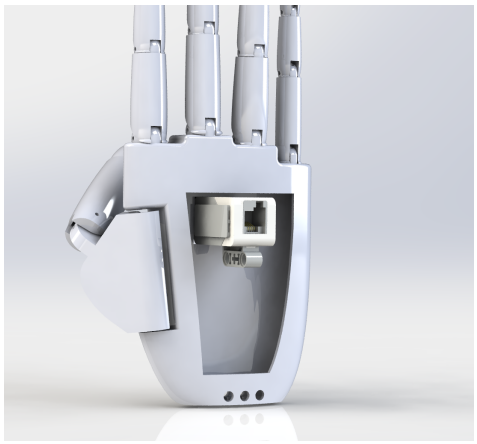


Figure 19: Touch Sensor Mounting

4.7.2 Gyroscope

To mount the gyroscope, the team used two LEGO connectors from the turntable onto the sensor as well as zip tying it for additional strength. See **Figure 20**: Gyroscope Mounting. It was important to mount the gyroscope to the turntable to ensure that the angle of rotation was accurately measured.

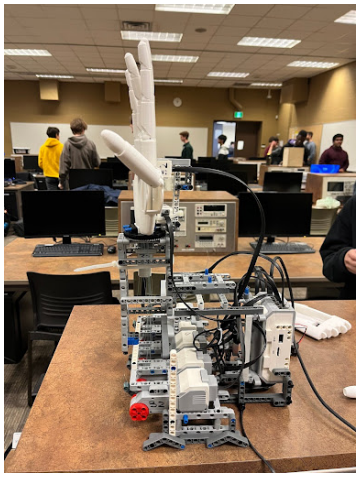


Figure 20: Gyroscope Mounting

5 Software Design and Implementation

5.1 Overall Design

Thing is programmed using RobotC, the native programming language for the LEGO EV3 brick. The code is segmented into eleven individual functions with three main subsections: file input and validation, finger motion, and response to human interaction.

Each main subsection controls a different aspect of the robot operation. The file input and validation portion ensures that nonprofane words are properly loaded into the EV3 brick. The functions that directly correspond to file input and validation include: "profanityCheck," "cipherText," and "toLower." The finger motion subsection ensures each finger, thumb, and turntable move to their corresponding encoder value. The functions that control this are "moveFinger," "moveRoll," "resetHand," "resetNCase," and "sign". Lastly, *Thing* checks for human interaction by pressing the touch sensor. This is divided between checking sensor input in the main function and the "wave" function.

An important consideration in the software design process was the amount of motor ports available on the EV3. The EV3s come with four motor ports, however, *Thing* requires six motors. To overcome this constraint, the team was able to use a multiplexer to control two of the motors separately. Since the multiplexer is not a part of the EV3 by default, it requires a different syntax to control those motors.

5.2 Initial Task List

The original version of the software was a simplified version of the task list in order to provide a mechanical proof of concept and fit the initial constraints. As per the flowchart in **Figure 21**: Initial Flowchart, the program followed the following steps: wait for the enter button to be pressed, read each character from the file input, move to the corresponding ASL letter sign, wait ten seconds, and then move to the home position. If the touch sensor is pressed in between letters, *Thing* would wave and then proceed to the next letter.

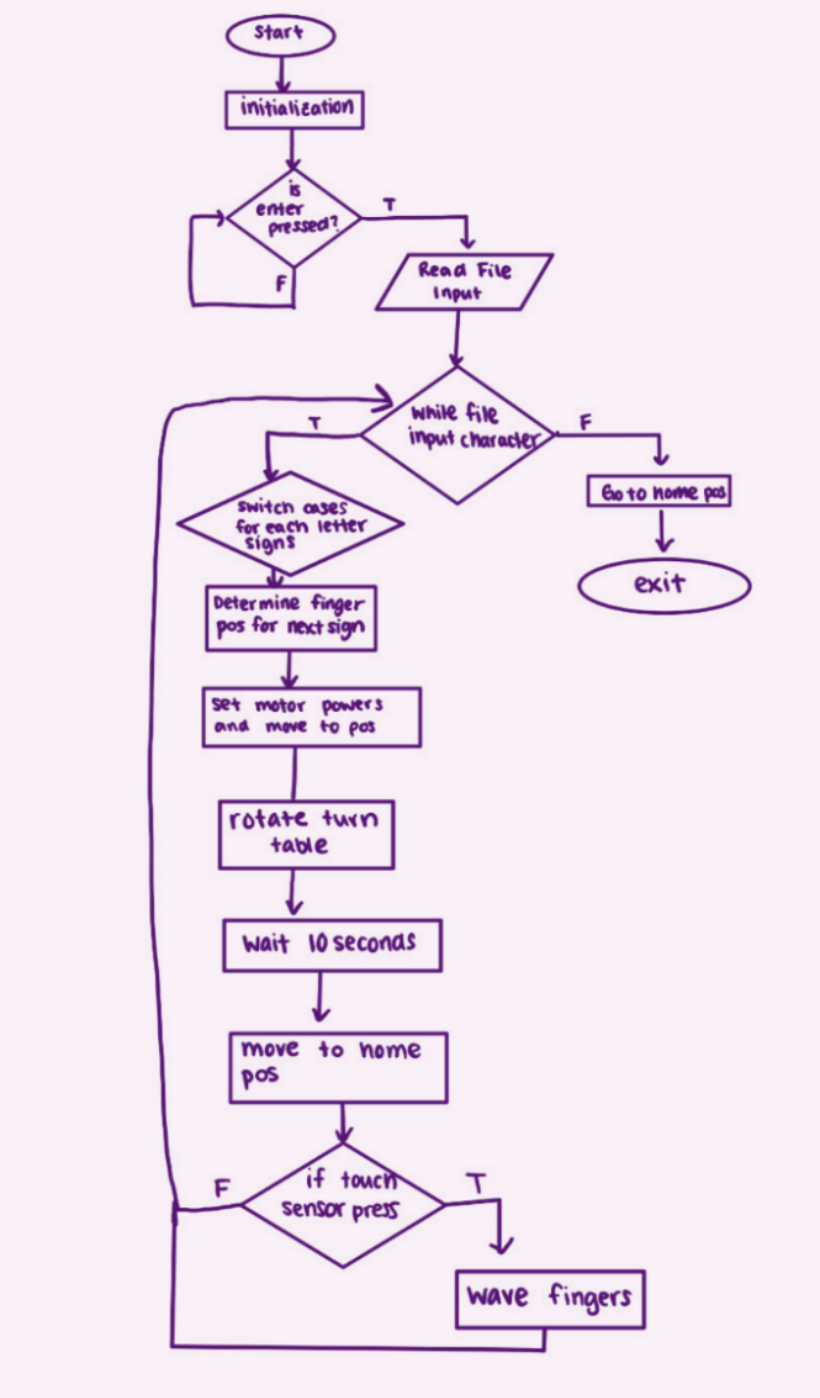


Figure 21: Initial Flowchart

5.3 Final Task List

The final version of the software is designed to meet all of the MTE 121 software requirements. The task list was updated to include a profanity checker, the ability to sign multiple words, and the touch

sensor “high five” became the program termination procedure instead of in between characters. See **Figure 22**: Final Flowchart for the overall structure of the code.

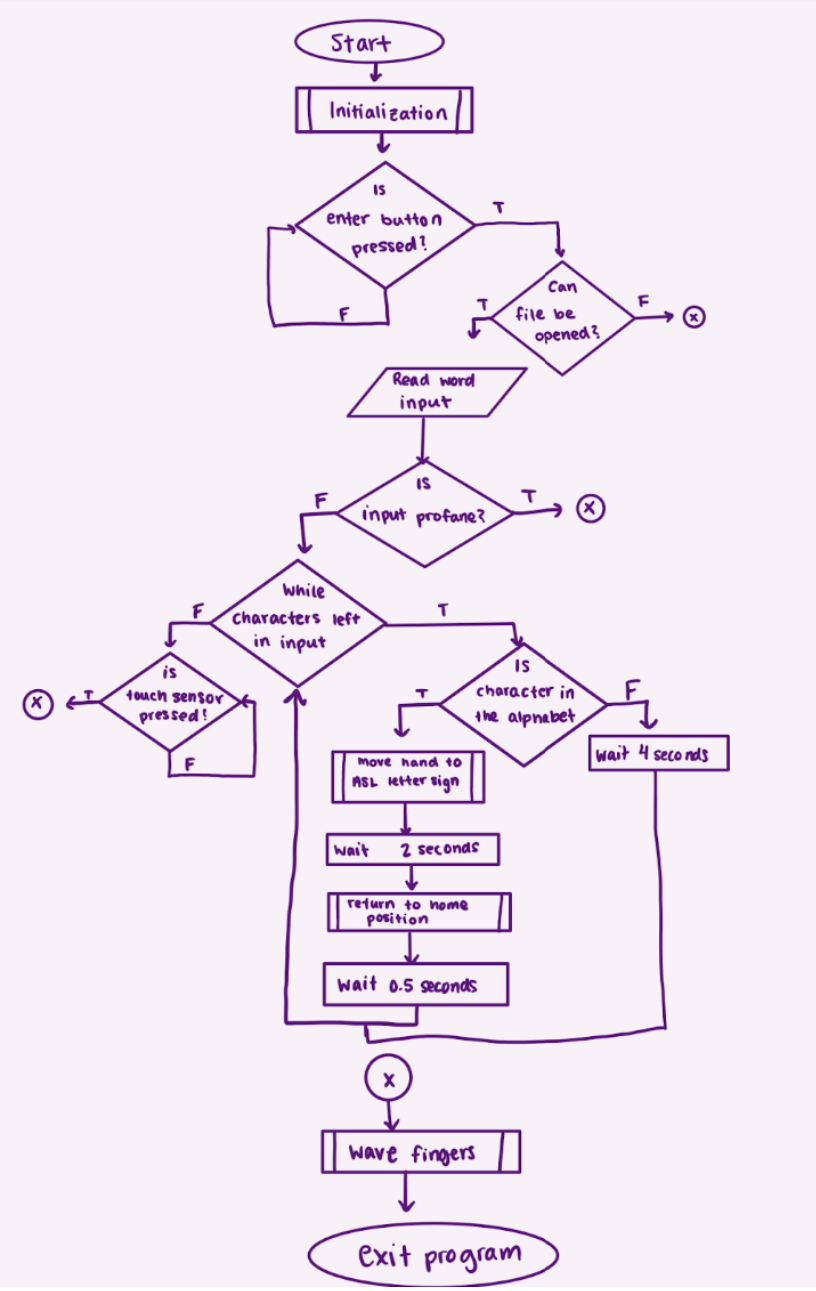


Figure 22: Final Flowchart

5.4 Function Breakdown

5.4.1 Sensor Configuration

The "configureAllSensors()" function initializes sensor ports, motor encoders, and the EV3 multiplexer. This function is based off of the MTE 121 sensor configuration functions, tailored to match each motor with *Thing*'s configuration opposed to "standard configuration". See **Figure 23**: Sensor Configuration Prototype. Meghan wrote this function.

```
34 void configureAllSensors();
```

Figure 23: Sensor Configuration Prototype

5.4.2 Timer

The "waitSeconds()" function takes an integer of seconds, starts a timer, and waits for the specified time to pass before completing the function. See **Figure 24**: waitSeconds() Flowchart and **Figure 25**: waitSeconds() Prototype. Khushi wrote this function.

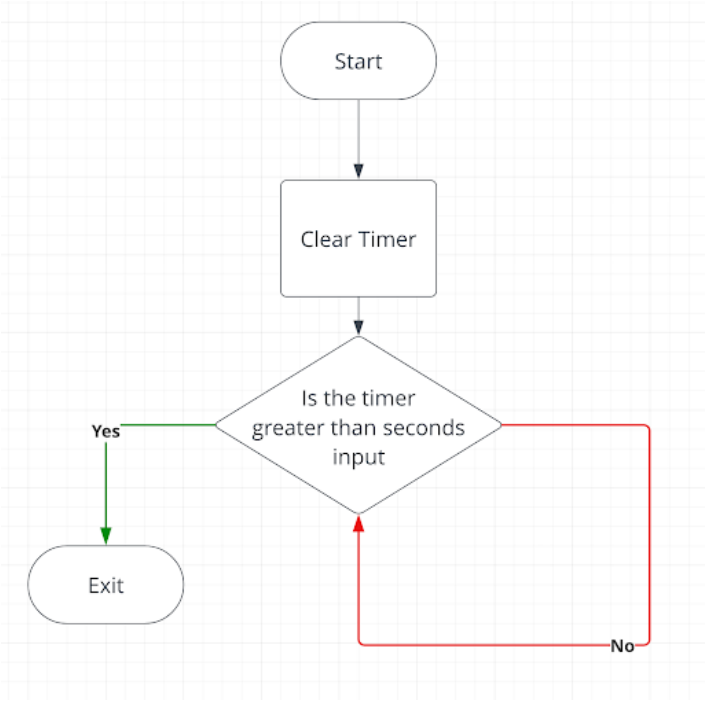


Figure 24: waitSeconds() Flowchart

```
36 // Waits for amount of seconds
37 void waitSeconds(int seconds);
```

Figure 25: waitSeconds() Prototyped

5.4.3 Finger Actuation

The "moveFinger()" function takes a Fingers enum and encoder target integer as parameters, and moves the corresponding finger to that position with a series of switch case statements (See **Figure 26**: moveFinger() Flowchart and **Figure 27**: moveFinger() Prototype). The finger data is stored in an enum to increase readability, and make the switch cases more coherent. The switch case statement is required as opposed to a generic move motor function since the syntax is different for controlling motors plugged into the EV3 port versus the motors plugged into the multiplexer. Evie wrote this function.

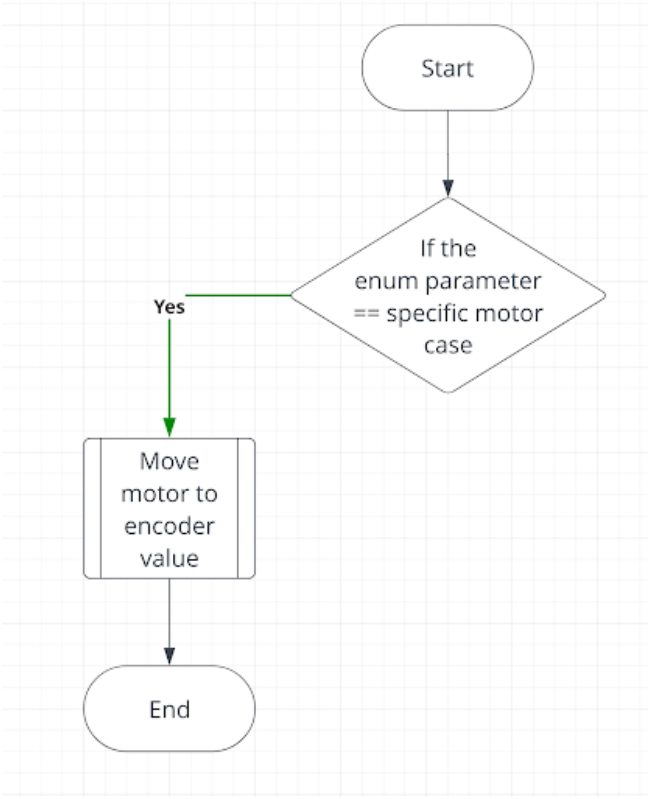


Figure 26: moveFinger() Flowchart

```
40 void moveFinger(enum Fingers finger, int encoderTarget);
```

Figure 27: moveFinger() Prototype

5.4.4 Roll Joint

The "moveRoll()" function is responsible for controlling the turntable rotation. The function takes in a gyro limit integer as well as an integer for the motor power. If the motor power is less than zero, the hand will rotate counterclockwise and if it is positive it will rotate clockwise. To not break the strings, the hand never moves more than 35 degrees, as measured by the gyroscope (See **Figure 28**: moveRoll() Flowchart and **Figure 29**: moveRoll() Prototype). Evie wrote this function.

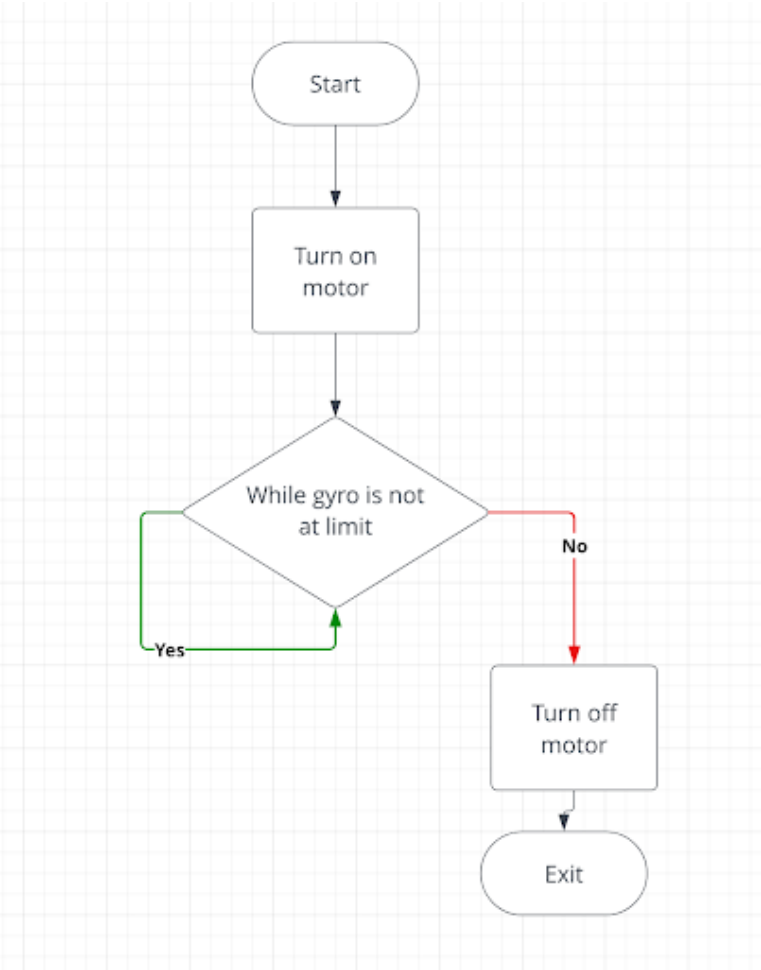


Figure 28: moveRoll() Flowchart

```
void moveRoll(int encoderLimit, int motor_power);
```

Figure 29: moveRoll() Prototype

5.4.5 Reset to Home Position

The "resetHand()" function has no parameters or return type. The flow chart is seen in **Figure 30**: resetHand() Flowchart. After each letter is signed, this function is called to move each finger back to its home position: an encoder value of zero in an upright position. The fingers to reset are called from pinky to thumb to reduce conflicts between finger positions. Khushi wrote this function.

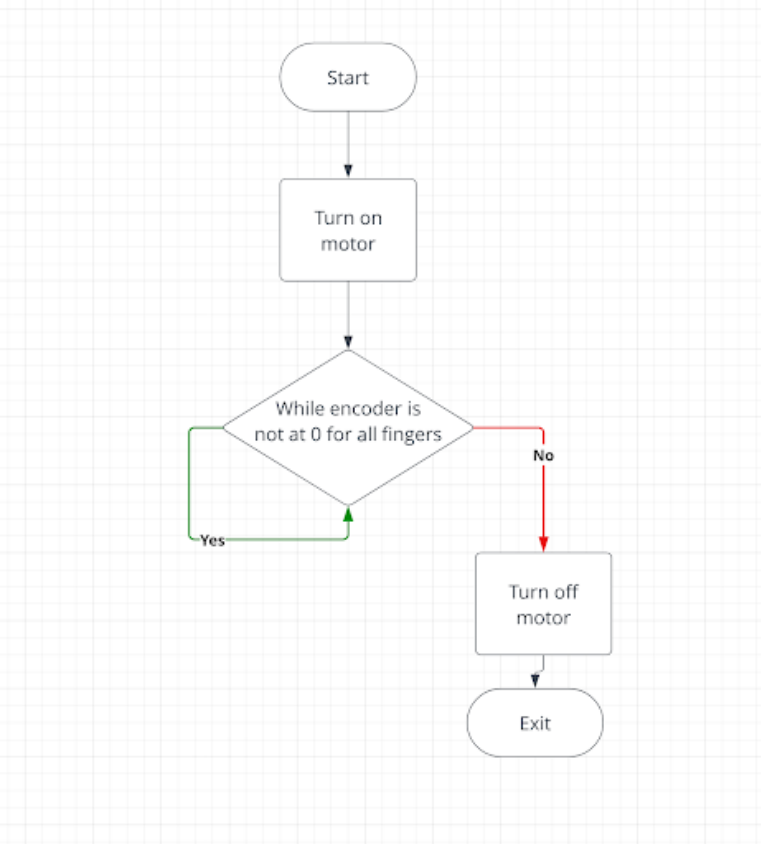


Figure 30: resetHand() Flowchart

Special Case

The "resetNCase()" function is another version of resetting the hand back to the home position. Since the ASL sign for 'n' involves the thumb being placed on top of the middle finger with the index finger on top of the thumb, the normal reset case will result in conflict. To resolve this, a specific function was made to reset the motors in a different order. The function prototypes for resetHand() and resetNCase() are shown in **Figure 31**: resetHand() and resetNCase() Prototypes. Meghan wrote this function.

```
45 // Reset Position
46 void resetHand();
47 void resetNCase();
```

Figure 31: resetHand() and resetNCase() Prototypes

5.4.6 ASL Letter Signs

The "sign()" function is what dictates the position for each ASL sign (See **Figure 32**: sign() Flowchart and **Figure 33**: sign() Prototype). Pending the character that is passed into the function, the program will move each finger to the two main encoder positions: "CLOSE_LIM," "HALF_LIM," or no change. "sign()" contains twenty-seven switch cases for each letter of the alphabet as well as if the character is a space. If it is a space, that indicates to the program that the first word has finished. It will then wait for two seconds before continuing to the next character. Evie made this function.

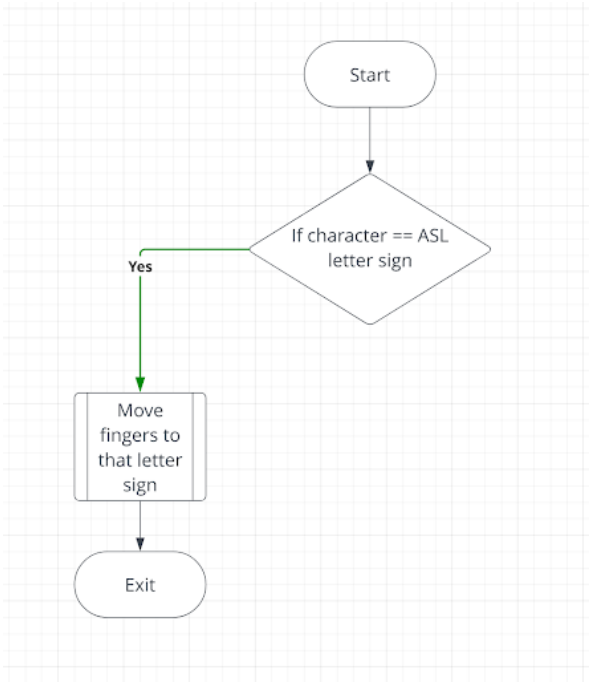


Figure 32: sign() Flowchart

```
49 // Sign letter
50 void sign(char letter);
```

Figure 33: sign() Prototype

5.4.7 Wave

When the touch sensor is pressed, the "wave()" function is called to move *Thing*'s four fingers to the halfway limit and back up twice, simulating a human wave. The function determines the direction to move the hand based on if the iteration of a for-loop is even or odd. For example, the first iteration, an odd number, will move the fingers down.See **Figure 34**: wave() Flowchart and **Figure 35**: wave() Prototype. The second iteration is an even number so it will move the fingers up. Evie made this function.

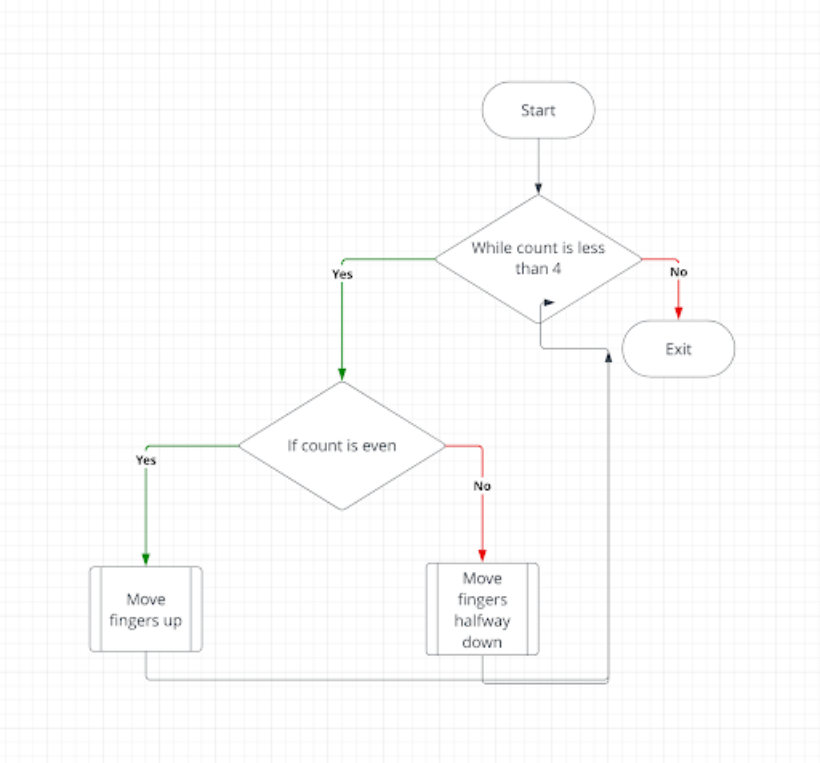


Figure 34: wave() Flowchart

```
53 void wave();
```

Figure 35: wave() Prototype

5.5 File Input and Data Organization

In order to achieve the desired functionality with the hand, simply iterating through each character of input and signing as it went would have been sufficient. However, one of the criteria was profanity checking. This meant that the data had to be structured more intentionally.

Inherently, a string is an array of characters. This is well-versed in the C language, where the string type does not exist. Instead, it is defined by char*, a pointer to the first character of the string, where it iterates through until it reaches the terminating null character. This property makes

string manipulation easy, since specific characters can be accessed through indexing. Unfortunately, this syntax does not directly translate to RobotC, very likely due to the fact that RobotC was never intended for string manipulation. In practice, it is not possible to index into RobotC strings. Thus, it was necessary to declare the variable as char * in order to index a character. While char * has iterative properties, it is not an allowed data type to read from a file into.

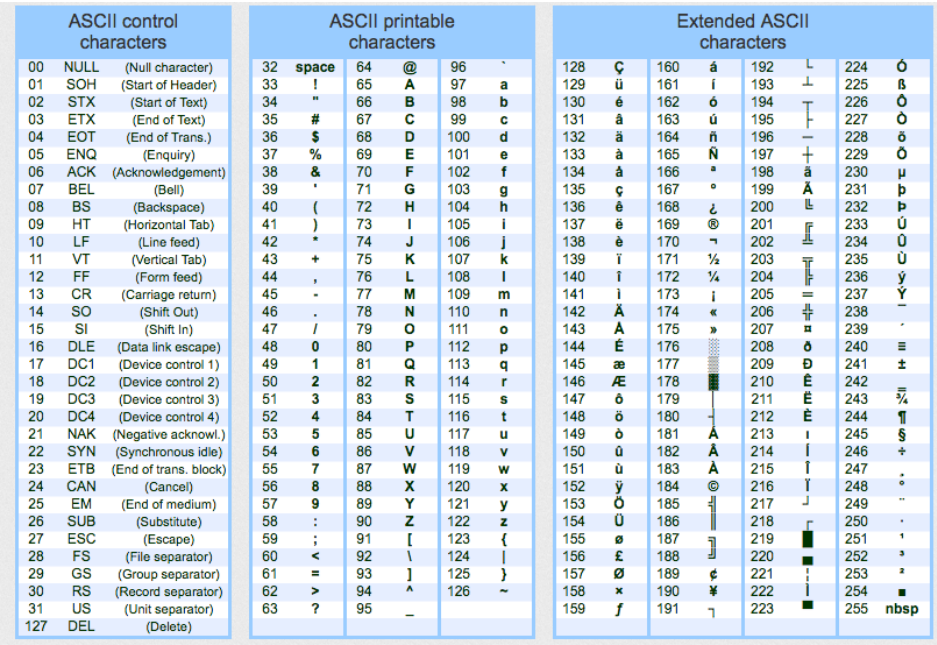
To overcome this challenge, seemingly redundant programming choices were made. A string variable is used to store the text read from the file, but another variable, the pointer to a character, is created, pointing to the first character of the string. This way, the program reads the word into a variable, and another variable creates the pointer which can be properly manipulated.

5.6 Lower Case

Another consideration was to make all the letters either uppercase or lowercase. This is important in order to eliminate the need for checking against both cases. This concept is available as an existing function in most languages such as Python (lower()) and C (tolower()). Again, due to the fact that RobotC was never created for string manipulation, it does not already contain this function.

This introduced the custom written toLower() function. The function takes a pointer to a character as the parameter, and returns nothing. It is important to understand the purpose of this pointer and its relation to the void return. The function does not take a character and return the character as a lowercase. It will inherently iterate through each character of the string, and change the character passed in and transform it to the lowercase version.

For each letter it iterates through, it compares the character to a number. Another concept in C that was necessary to code this functionality was the idea wherein a character is a number in the ASCII table, and therefore can be directly compared to a number to check its value. For the purposes of this function, it had to check whether the character was in the range to be an uppercase (41-90), and would add 32 to convert it to its lowercase (eg. A = 65, a = 97 = 65 + 32). See **Figure 36**: ASCII Table [6].



ASCII control characters

00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters

32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Extended ASCII characters

128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

Figure 36: ASCII Table

This function also deals with unexpected input. For example, if any special characters are read from the input, the function converts that character into a space, which can be dealt with in "sign()". See **Figure 37**: toLower() Flowchart and **Figure 38**: toLower() Prototype. Rubie wrote this function.

Figure 37: toLower() Flowchart

```
58 void toLower(char *input);
```

Figure 38: toLower() Prototype

5.7 Cipher Text

The easiest implementation of checking if a word is profane is to compare the input string with an array of the unacceptable words. If it matched anywhere, the robot would not sign the word. This was a good idea conceptually, however it involves writing actual swear words in the code.

To overcome this, the "cipherText()" function takes a character as a parameter and returns the same character shifted 5 down the alphabet. This allows for both the input and array of profanity to be practically unreadable to a user. The function will then check whether the operation has moved the character out of the alphabet, if the input is greater than 122, then subtracts 26 if it has overflowed allowing the character to wrap around back to the start of the alphabet. See **Figure 39**: cipherText() Flowchart and **Figure 40**: cipherText() Prototype. Rubie wrote this function.

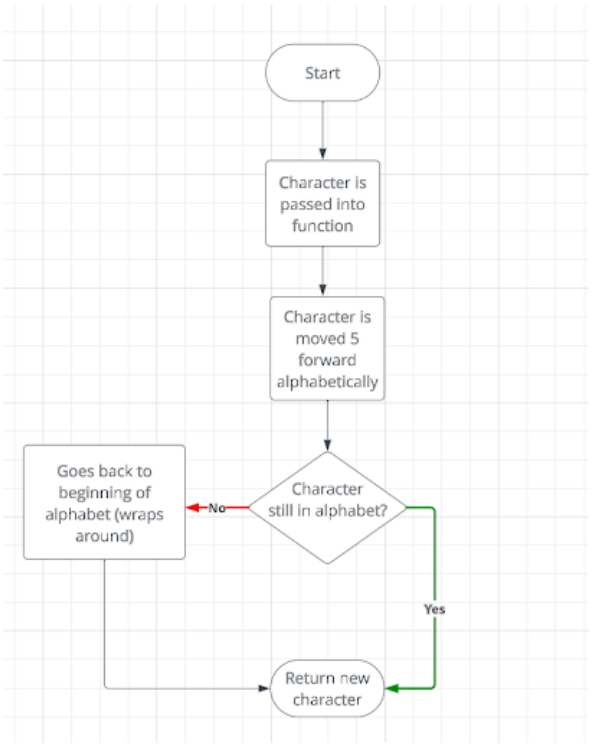


Figure 39: cipherText() Flowchart

```
57 char cipherText(char input);
```

Figure 40: cipherText() Prototype

5.8 Profanity Checker

The "profanityChecker()" function takes in a char * as an argument, and returns a boolean of whether profanity was found or not. The string is iterated through and each character is fed through the "cipherText()" function. After all the characters are ciphered, the "stringFromChars" function in the RobotC library is used to create a string from the array of characters. This string can then be compared against an array of other ciphered profanities to check whether the input itself is profane. It is also notable that stringFind is used rather than a simpler string comparison. It was decided that the profanity checker would not only check whether the whole word was profane, but to check whether profanity was found anywhere in the input, just for extra security. If any profanity was found in the input, the function displays that it will not sign the text, and returns true. See **Figure 41**: profanityChecker() Flowchart and **Figure 42**: profanityChecker() Prototype. Rubie wrote this function.

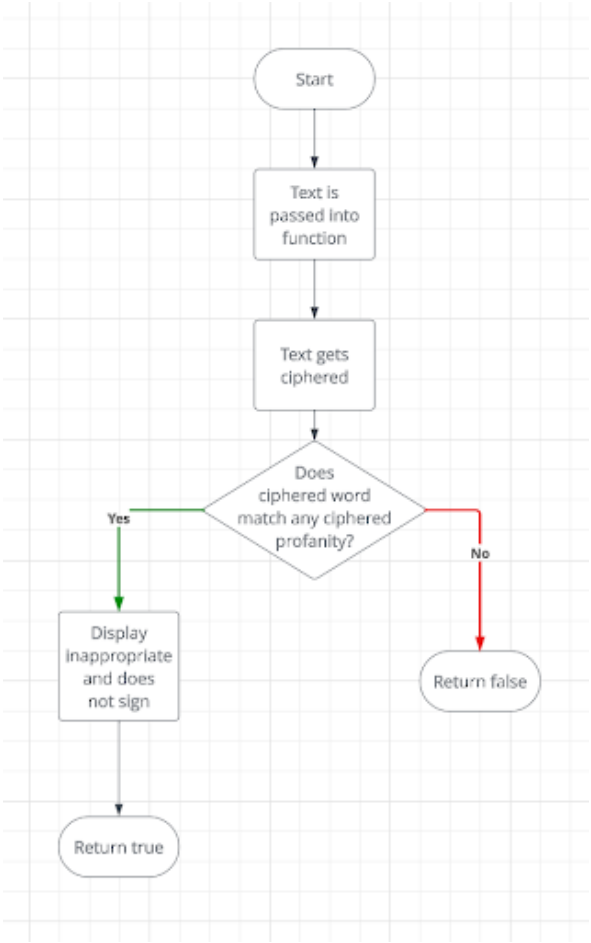


Figure 41: profanityChecker() Flowchart

```
56 bool profanityCheck(char *input);
```

Figure 42: profanityChecker() Prototype

5.9 Software Testing

The team used a shared GitHub project in order to properly collaborate on the code. Source control is also important so that group members' code is not overwritten. Due to the size of the code, there is a lot of room for error if something is programmed incorrectly. As such, before running code, all group members agreed that at least one person would review the code that was attempting to be run.

As a new function or process was being implemented, the team placed a large emphasis on proper integration and testing: First, the features the function were to perform were tested individually in isolation from the other code; Then, the full function was programmed and tested, still in isolation from the full code; Once this successfully completed testing, the function was then integrated into the rest of the code to be tested. This iterative testing cycle reduced the amount of error in the code, and eliminated common errors such as string snapping and unnecessary mechanical strain.

Another feature that had to be repeatedly tested was what the encoder limits were. Before choosing an encoder value, the team programmed and ran a function that moved the motors forward and back with the use of the EV3 buttons, displaying the current encoder value. When the team could visually see what the approximate encoder values were, then limits could be selected and tested.

5.9.1 Problem Identification

As previously mentioned, a lack of string manipulation capabilities in RobotC was a problem, resulting in custom string functions. Additionally, the multiplexer provided a large source of error in the project. Since it is powered separately from the EV3 and not connected to official motor ports, sometimes the fingers controlled by the multiplexer would not move. This was not a result of the code, but a mere electrical failure.

6 Verification

As described in both the mechanical and software sections, the majority of the criteria and constraints for the project were met. Each finger is individually actuated with three joints, the hand size is approximately the size of a human hand, the hand possesses a roll joint, and the file input is scanned for profanity.

The only constraints that were not fully met are the full 90 degree rotation of the roll joint, and a fully comprehensive profanity checker. Since the strings of the hand are tensioned tightly, the turntable is incapable of reaching a full 90 degree rotation. Instead, it could reach a maximum of 30 degrees. After this, the gears would begin to skip, as the motor did not have enough torque to overcome the force of the strings. Additionally, although the profanity checker was functional, it did not check every single swear word, as this is not realistic nor the goal of the project. The profanity checker serves as a proof of concept that input validation is possible and the team could limit a select few swear words.

7 Project Plan

7.1 Task Division

Mechanical

Rubie and Evie worked together to CAD and manufacture the hand, while Meghan and Khushi worked on the LEGO base integration design and assembly.

Software

Each group member collaborated in finishing the software; programming multiple functions each. These are more thoroughly discussed in **Subsection 5.4 Function Breakdown**.

7.2 Timeline

The team created a timeline to ensure that all of the deliverables would be completed with ample time for verification. See Table 1: Project Deliverables. The team stayed on track to each of the tasks: the first prototype and 3D print was completed by the deadlines, however, as errors arose along with new design ideas, the mechanical timeline was extended.

Table 1: Project Deliverables

Task/Deliverable	Submit/Complete	Due Date
Prosthetic CAD	n/a	Nov 4
3D print all pieces	n/a	Nov 7
Formal Presentation	In person (slideshow)	Nov 10 (MTE100 lab)
Complete Software design	n/a	Nov 10
Write preliminary software	n/a	Nov 14
Software design review	In person	Nov 13-14 (MTE121 tut)
Build/assemble hand	n/a	Nov 15
Test software and debug	n/a	Nov 23
Demo Day	In person, presented twice	Nov 24 (MTE100 lab)
Final Report	Crowdmark	Dec 5
Return robot parts	n/a	Dec 1st (MTE100 tut)

8 Conclusion

In addressing the communication barriers faced by many in the community, *Thing* aims to create an innovative robotic system using LEGO EV3 Mindstorm technology, CAD, 3D printing, and mechanical elements. The primary objective is to increase the accessibility and ease of learning ASL.

The mechanical design of *Thing* consists of one degree of freedom per finger, specifically targeting the MCP joint's bending motion. The 3D-printed hand, six motors, and tensioning system facilitated the intricate movements required to sign each letter of a word.

This system successfully met many of the outlined constraints and criteria. *Thing*'s functionality extended beyond the initial scope, evolving to include a profanity checker for file input, the ability to

sign multiple words, and a wave feature during program termination. While *Thing* was incapable of rotating 90°, the 35° wrist rotation still enables users to see how the letter should be signed

The RobotC software processes user input, enabling *Thing* to convert text files into ASL letters. Through extensive testing and knowledge of the C language, *Thing* was able to validate input to ensure user safety. Further, the integration of sensors throughout the robot provides a physical and interactive learning experience.

At the end of the project, the team presented *Thing* at the University of Waterloo's "Demo Day." As part of the showcase, the robot signed each professor and teaching assistant's name as a proof of concept.

Ultimately, *Thing* stands as a successful integration of mechanical and software design, breaking down communication barriers for ASL users and extending its capabilities beyond the initial project scope.

9 Recommendations

With the short timeline of one month to complete this project, design and software trade-offs were made to accommodate the time constraint. Provided more time and resources, the team would have chosen alternate mechanical solutions for wrist motor attachment and size, as well as additional degrees of movement. Additionally, with increased time, complexity could have been added to the movements between letters, removing the resetHand function entirely, as well as optimizing the profanity checker.

9.1 Mechanical Changes

As previously discussed, based on the verification data observed during the demo, the team would implement an enhanced wrist movement system for the roll joint. A possible solution would be to swap out the current medium motor for a large motor, providing more torque to overcome the skipping issue that is currently faced. Another option would be to 3D print or machine a custom gear or shaft to prevent the flexing observed in the Lego pieces used for the roll joint. This would allow the wrist to meet the 90-degree rotation constraint and move smoothly, without skipping, to minimize interference while completing tasks.

With *Thing*'s current degrees of movement, it can complete the majority of signs with considerable likeliness to their true forms. Certain exceptions include letters such as "G" and "H" which require the "yaw" movement. With only 6 sensors, this was impossible to integrate into the current design. Provided with additional large motors, the addition of yaw could be added through a ball joint in the wrist, similar to those in the fingers, however, upscaled to fit the base. The joint could be controlled using strings attached to a single motor, and would allow for 2 additional degrees of movement.

9.2 Software Changes

The current system for moving between letters features the resetHand function, which returns all fingers to standard position before moving forward with the next letter. This system works the majority of the time, except under special conditions such as for letter "N". The motion of completely resetting the hand is unlike what is used in proper ASL. Typically, fingers that do not require additional

movement will stay in place, while those that do will move according to the given sign. If given more time, the team would have implemented a completely different system for switching between letters.

Instead of resetting after each letter, which takes additional time and power, the motor encoder values would be read, and based on the next letter, the required fingers would move accordingly. This system would remove the need for special condition functions, as it would only move fingers when necessary. The creation of this system would allow for easier code maintenance, as there is only 1 function used between the movement of fingers, instead of several.

Moreover, the profanity-checking system could be optimized to include a more extensive list. In industry, it can be assumed that the use of stronger hardware would allow for a database of profanities that can be compared to the input. Currently, the profanity checker contains only 7 elements, which is not ideal for mainstream use. With improved hardware and storage, the profanity checker could instead compare the input to all entries in the profanity database. Not only would this ensure no profanities are let through, but it would also eliminate any errors when profanities are contained within regular words.

References

[1] "American sign language," National Institute of Deafness and Other Communication Disorders, https://www.nidcd.nih.gov/health/american-sign-language (accessed Dec. 5, 2023).

[2] "American sign language," American Sign Language | Commission on the Deaf and Hard of Hearing (CDHH), https://cdhh.ri.gov/information-referral/american-sign-language.php (accessed Dec. 5, 2023).

[3] "How do you sign the alphabet in ASL? sign language online learning tips," The Alphabet in ASL Sign Language | Learn American Sign Language, https://www.lingvano.com/asl/blog/sign-language-alphabet/ (accessed Dec. 5, 2023).

[4] A. Rahman and A. Al-Jumaily, "Design and Development of a Bilateral Therapeutic Hand Device for Stroke Rehabilitation," International Journal of Advanced Robotic Systems, vol. 10, no. 12, p. 405, Jul. 2013. doi:10.5772/56809 (accessed Dec. 5, 2023).

[5] A. Pawar and S. Matekar, "Design of Prosthetic Arm," AIP Conference Proceedings, 2022. (accessed Dec. 5, 2023).

[6] "ASCII Chart," Microsoft on GitHub, https://microsoft.github.io/makecode-csp/unit-6/day-12/ascii-chart/ (accessed Dec. 5, 2023).

Appendix A

```
/**
 * Authors: Khushi Patel, Meghan Dang, Rubie Luo, Evie Bouganim
 * Description: Final 1A Mechatronics project. 'Thing' is a robotic
 *        hand that can perform American Sign Language (ASL)
 *        letters from file input. Each inputted word is screened
 *        for profanity. After signing all words, 'high five'
 *        the touch sensor for it to wave goodbye.
 * Version: 4.0
 * Acknowledgements: We would like to thank the MTE 121 and MTE 100
 *           teaching team for supporting us and providing
 *           some materials throughout the project development.
 */

#include "mindsensors-motormux.h"
#include "PC_FileIO.c"

// Got approval from Prof. Nassar to use global constants here
enum Fingers
{
  THUMB,
  INDEX,
  MIDDLE,
  RING,
  PINKY
};

const int ENCODER_HOME = 0;
const int CLOSE_LIM = 140;
const int HALF_LIM = 35;
const int MOTOR_POWER = 35;
const int TURN_POWER = 13;
const int ROLL_JOINT = 55;
const int WAVE_POWER = 20;

// Configures all sensors
void configureAllSensors();

// Waits for amount of seconds
void waitSeconds(int seconds);

// Moves each finger
void moveFinger(enum Fingers finger, int encoderTarget);

// Rotate wrist joint
```

```
45 void moveRoll(int encoderLimit, int motor_power);
46
47 // Reset Position
48 void resetHand();
49 void resetNCase();
50
51 // Sign letter
52 void sign(char letter);
53
54 // Waves hello
55 void wave();
56
57 // Checks of input is profane
58 bool profanityCheck(char *input);
59 char cipherText(char input);
60 void toLower(char *input);
61
62 task main()
63 {
64   configureAllSensors();
65
66   // Open file
67   TFileHandle fin;
68   bool fileOkay = openReadPC(fin, "sign.txt");
69
70   // If file does not open, display error for 10 seconds and end program
71   if (!fileOkay)
72   {
73     displayString(5, "Error! Could not open file.");
74     waitSeconds(5);
75     wave();
76     return;
77   }
78
79   // Wait for enter button to be clicked to begin sign language
80   while (!getButtonPress(buttonEnter))
81   {
82   }
83   while (getButtonPress(buttonEnter))
84   {
85   }
86   waitSeconds(3);
87
88   // Declare word input string to store the input
89   string wordInput;
90   // Create pointer to the first letter of the string, used to divide each char
```

```
  char *letter = &wordInput;
  // Read text from file
  readTextPC(fin, wordInput);
  // Convert the string to lowercase for consistency
  toLower(letter);

  // Check if word contains profanity, if it does, it will not sign.
  if (!profanityCheck(wordInput))
  {
    // If word does not have profanity, iterate through each letter and sign it
    for (int i = 0; i < strlen(wordInput); i++)
    {
      sign(*(letter + i)); // Pointer to the char to sign
      waitSeconds(2);
      if (*(letter + i) != 'n')
      {
        resetHand();
        waitSeconds(0.5);
      }
    }
    displayString(5, "Done! High five to exit :)");
  }

  while (!SensorValue[S4])
  {
  }
  while (SensorValue[S4])
  {
  }
  wave();

  return;
}

/**
 * Configures all sensors
 *
 * @return None
 */
void configureAllSensors()
{
  SensorType[S4] = sensorEV3_Touch;
  wait1Msec(50);
  SensorType[S3] = sensorEV3_Gyro;
  wait1Msec(50);
  SensorMode[S3] = modeEV3Gyro_Calibration;
```

```
137   wait1Msec(100);
138   SensorMode[S3] = modeEV3Gyro_RateAndAngle;
139   wait1Msec(50);
140
141   nMotorEncoder[motorA] = 0;
142   nMotorEncoder[motorB] = 0;
143   nMotorEncoder[motorC] = 0;
144   nMotorEncoder[motorD] = 0;
145
146   // Initialize, for the multiplexer connected to S1
147   SensorType[S1] = sensorI2CCustom;
148   MSMMUXinit();
149
150   // Reset multiplexer motor encoders
151   MSMMotorEncoderReset(mmotor_S1_1);
152   MSMMotorEncoderReset(mmotor_S1_2);
153 }
154
155 /**
156  * Waits given amount of time
157  *
158  * @param seconds Time in seconds to wait
159  * @return None
160  */
161 void waitSeconds(int seconds)
162 {
163   clearTimer(T1);
164   while (time1[T1] <= seconds * 1000)
165   {
166   }
167 }
168
169 /**
170  * Move a specified finger to desired position
171  *
172  * @param finger Finger (enum) to move
173  * @param encoderTarget Position to move specified finger
174  * @return None
175  */
176 void moveFinger(enum Fingers finger, int encoderTarget)
177 {
178   // Switch case statement to move the finger that is passed in the function
179   switch (finger)
180   {
181   case THUMB:
182     motor[motorD] = MOTOR_POWER;
```

```
    while (nMotorEncoder[motorD] < encoderTarget)
    {
    }
    motor[motorD] = 0;
    break;

  case INDEX:
    motor[motorC] = MOTOR_POWER;
    while (nMotorEncoder[motorC] < encoderTarget)
    {
    }
    motor[motorC] = 0;
    break;

  case MIDDLE:
    motor[motorB] = MOTOR_POWER;
    while (nMotorEncoder[motorB] < encoderTarget)
    {
    }
    motor[motorB] = 0;
    break;

  case RING:
    MSMMotor(mmotor_S1_1, MOTOR_POWER);
    while (MSMMotorEncoder(mmotor_S1_1) < encoderTarget)
    {
    }
    MSMotorStop(mmotor_S1_1);
    break;

  case PINKY:
    MSMMotor(mmotor_S1_2, MOTOR_POWER);
    while (MSMMotorEncoder(mmotor_S1_2) < encoderTarget)
    {
    }
    MSMotorStop(mmotor_S1_2);
    break;
  }
}

/**
 * Move roll joint to specified angle
 *
 * @param encoderLimit Position to rotate to
 * @param motor_power motor power for rotation
 * @return None
```

```
 */
void moveRoll(int gyrorLimit, int motor_power)
{
  motor[motorA] = motor_power;

  // case for counterclockwise and clockwise rotation
  if (motor_power < 0)
    while (abs(getGyroDegrees(S3)) < gyroLimit)
    {
    }
  else
    while (getGyroDegrees(S3) > gyroLimit)
    {
    }

  motor[motorA] = 0;
}

/**
 * Resets position for all motors
 *
 * @return None
 */
void resetHand()
{
  moveRoll(ENCODER_HOME, TURN_POWER);

  MSMMotor(mmotor_S1_2, -MOTOR_POWER);
  while (MSMMotorEncoder(mmotor_S1_2) > 0)
  {
  }
  MSMotorStop(mmotor_S1_2);

  MSMMotor(mmotor_S1_1, -MOTOR_POWER);
  while (MSMMotorEncoder(mmotor_S1_1) > 0)
  {
  }
  MSMotorStop(mmotor_S1_1);

  motor[motorB] = -MOTOR_POWER;
  while (nMotorEncoder[motorB] > 0)
  {
  }
  motor[motorB] = 0;

  motor[motorC] = -MOTOR_POWER;
```

```
  while (nMotorEncoder[motorC] > 0)
  {
  }
  motor[motorC] = 0;

  motor[motorD] = -MOTOR_POWER;
  while (nMotorEncoder[motorD] > 0)
  {
  }
  motor[motorD] = 0;
}

/**
 * Resets specific fingers first for the 'n' sign so that no positional conflicts
 *
 * @return None
 */
void resetNCase()
{
  motor[motorC] = -MOTOR_POWER;
  while (nMotorEncoder[motorC] > 0)
  {
  }
  motor[motorC] = 0;

  motor[motorB] = -MOTOR_POWER;
  while (nMotorEncoder[motorB] > 0)
  {
  }
  motor[motorB] = 0;

  motor[motorD] = -MOTOR_POWER;
  while (nMotorEncoder[motorD] > 0)
  {
  }
  motor[motorD] = 0;

  MSMMotor(mmotor_S1_1, -MOTOR_POWER);
  while (MSMMotorEncoder(mmotor_S1_1) > 0)
  {
  }
  MSMotorStop(mmotor_S1_1);

  MSMMotor(mmotor_S1_2, -MOTOR_POWER);
  while (MSMMotorEncoder(mmotor_S1_2) > 0)
  {
```

```
  }
  MSMotorStop(mmotor_S1_2);
}

/**
 * Signs a letter in ASL by moving each finger to position
 *
 * @param letter Letter to sign
 * @return None
 */
void sign(char letter)
{
  switch (letter)
  {
  case 'a':
    moveFinger(INDEX, CLOSE_LIM);
    moveFinger(MIDDLE, CLOSE_LIM);
    moveFinger(RING, CLOSE_LIM);
    moveFinger(PINKY, CLOSE_LIM);
    moveFinger(THUMB, HALF_LIM);
    break;

  case 'b':
    moveFinger(THUMB, CLOSE_LIM);
    break;

  case 'c':
    moveRoll(ROLL_JOINT, -TURN_POWER);
    moveFinger(INDEX, HALF_LIM);
    moveFinger(MIDDLE, HALF_LIM);
    moveFinger(RING, HALF_LIM);
    moveFinger(PINKY, HALF_LIM);
    moveFinger(THUMB, HALF_LIM);
    break;

  case 'd':
    moveRoll(ROLL_JOINT, -TURN_POWER);
    moveFinger(THUMB, HALF_LIM);
    moveFinger(MIDDLE, CLOSE_LIM);
    moveFinger(RING, CLOSE_LIM);
    moveFinger(PINKY, CLOSE_LIM);
    break;

  case 'e':
    moveFinger(THUMB, CLOSE_LIM);
    moveFinger(INDEX, CLOSE_LIM);
```

```
367     moveFinger(MIDDLE, CLOSE_LIM);
368     moveFinger(RING, CLOSE_LIM);
369     moveFinger(PINKY, CLOSE_LIM);
370     break;
371
372   case 'f':
373     moveFinger(THUMB, HALF_LIM);
374     moveFinger(INDEX, HALF_LIM);
375     break;
376
377   case 'g':
378     moveRoll(ROLL_JOINT, -TURN_POWER);
379     moveFinger(MIDDLE, CLOSE_LIM);
380     moveFinger(RING, CLOSE_LIM);
381     moveFinger(PINKY, CLOSE_LIM);
382     moveFinger(THUMB, CLOSE_LIM);
383     break;
384
385   case 'h':
386     moveRoll(ROLL_JOINT, -TURN_POWER);
387     moveFinger(PINKY, CLOSE_LIM);
388     moveFinger(RING, CLOSE_LIM);
389     moveFinger(THUMB, CLOSE_LIM);
390     break;
391
392   case 'i':
393     moveFinger(INDEX, CLOSE_LIM);
394     moveFinger(MIDDLE, CLOSE_LIM);
395     moveFinger(RING, CLOSE_LIM);
396     moveFinger(THUMB, HALF_LIM);
397     break;
398
399   case 'j':
400     moveRoll(ROLL_JOINT, -TURN_POWER);
401     moveFinger(INDEX, CLOSE_LIM);
402     moveFinger(MIDDLE, CLOSE_LIM);
403     moveFinger(RING, CLOSE_LIM);
404     moveFinger(THUMB, HALF_LIM);
405     moveRoll(ENCODER_HOME, TURN_POWER);
406     break;
407
408   case 'k':
409     moveFinger(PINKY, CLOSE_LIM);
410     moveFinger(RING, CLOSE_LIM);
411     moveFinger(THUMB, CLOSE_LIM);
412     break;
```

```
413
414  case 'l':
415    moveFinger(PINKY, CLOSE_LIM);
416    moveFinger(RING, CLOSE_LIM);
417    moveFinger(MIDDLE, CLOSE_LIM);
418    break;
419
420  case 'm':
421    moveFinger(PINKY, CLOSE_LIM);
422    moveFinger(THUMB, CLOSE_LIM);
423    moveFinger(RING, CLOSE_LIM);
424    moveFinger(MIDDLE, CLOSE_LIM);
425    moveFinger(INDEX, CLOSE_LIM);
426    break;
427
428  case 'n':
429    moveFinger(PINKY, CLOSE_LIM);
430    moveFinger(RING, CLOSE_LIM);
431    moveFinger(THUMB, CLOSE_LIM);
432    moveFinger(MIDDLE, CLOSE_LIM);
433    moveFinger(INDEX, CLOSE_LIM);
434    waitSeconds(2);
435    resetNCase();
436    break;
437
438  case 'o':
439    moveRoll(ROLL_JOINT, -TURN_POWER);
440    moveFinger(INDEX, CLOSE_LIM);
441    moveFinger(MIDDLE, CLOSE_LIM);
442    moveFinger(RING, CLOSE_LIM);
443    moveFinger(PINKY, CLOSE_LIM);
444    moveFinger(THUMB, HALF_LIM);
445    break;
446
447  case 'p':
448    moveRoll(ROLL_JOINT, -TURN_POWER);
449    moveFinger(RING, CLOSE_LIM);
450    moveFinger(PINKY, CLOSE_LIM);
451    moveFinger(THUMB, HALF_LIM);
452    moveFinger(MIDDLE, HALF_LIM);
453    break;
454
455  case 'q':
456    moveRoll(ROLL_JOINT, -TURN_POWER);
457    moveFinger(PINKY, CLOSE_LIM);
458    moveFinger(RING, CLOSE_LIM);
```

```
459     moveFinger(MIDDLE, CLOSE_LIM);
460     break;
461
462   case 'r':
463     moveFinger(PINKY, CLOSE_LIM);
464     moveFinger(RING, CLOSE_LIM);
465     moveFinger(THUMB, CLOSE_LIM);
466     break;
467
468   case 's':
469     moveFinger(INDEX, CLOSE_LIM);
470     moveFinger(MIDDLE, CLOSE_LIM);
471     moveFinger(RING, CLOSE_LIM);
472     moveFinger(PINKY, CLOSE_LIM);
473     moveFinger(THUMB, HALF_LIM);
474     break;
475
476   case 't':
477     moveFinger(PINKY, CLOSE_LIM);
478     moveFinger(RING, CLOSE_LIM);
479     moveFinger(MIDDLE, CLOSE_LIM);
480     moveFinger(THUMB, CLOSE_LIM);
481     moveFinger(INDEX, CLOSE_LIM);
482     break;
483
484   case 'u':
485     moveFinger(PINKY, CLOSE_LIM);
486     moveFinger(RING, CLOSE_LIM);
487     moveFinger(THUMB, CLOSE_LIM);
488     break;
489
490   case 'v':
491     moveFinger(PINKY, CLOSE_LIM);
492     moveFinger(RING, CLOSE_LIM);
493     moveFinger(THUMB, HALF_LIM);
494     break;
495
496   case 'w':
497     moveFinger(PINKY, CLOSE_LIM);
498     moveFinger(THUMB, CLOSE_LIM);
499     break;
500
501   case 'x':
502     moveFinger(PINKY, CLOSE_LIM);
503     moveFinger(RING, CLOSE_LIM);
504     moveFinger(MIDDLE, CLOSE_LIM);
```

```
    moveFinger(THUMB, CLOSE_LIM);
    moveFinger(INDEX, HALF_LIM);
    break;

  case 'y':
    moveFinger(INDEX, CLOSE_LIM);
    moveFinger(MIDDLE, CLOSE_LIM);
    moveFinger(RING, CLOSE_LIM);
    break;

  case 'z':
    moveRoll(ROLL_JOINT, -TURN_POWER);
    moveFinger(PINKY, CLOSE_LIM);
    moveFinger(RING, CLOSE_LIM);
    moveFinger(MIDDLE, CLOSE_LIM);
    moveFinger(THUMB, CLOSE_LIM);
    moveRoll(0, TURN_POWER);
    break;

  case ' ':
    waitSeconds(2);
    break;
  }
}

/**
 * 'Waves' fingers goodbye
 *
 * @return None
 */
void wave()
{
  int direction = 1;

  for (int count = 1; count <= 4; count++)
  {
    if (count % 2 == 0)
    {
      direction = -1;
      motor[motorC] = WAVE_POWER * direction;
      motor[motorB] = WAVE_POWER * direction;
      MSMMotor(mmotor_S1_2, WAVE_POWER * direction);
      MSMMotor(mmotor_S1_1, WAVE_POWER * direction);
      while (MSMMotorEncoder(mmotor_S1_1) > 0)
      {
      }
```

```
551       motor[motorC] = 0;
552       motor[motorB] = 0;
553       MSMMotor(mmotor_S1_2, 0);
554       MSMMotor(mmotor_S1_1, 0);
555       waitSeconds(0.75);
556     }
557     else
558     {
559       direction = 1;
560       motor[motorC] = WAVE_POWER * direction;
561       motor[motorB] = WAVE_POWER * direction;
562       MSMMotor(mmotor_S1_2, WAVE_POWER * direction);
563       MSMMotor(mmotor_S1_1, WAVE_POWER * direction);
564       while (MSMMotorEncoder(mmotor_S1_1) < 40)
565       {
566       }
567       motor[motorC] = 0;
568       motor[motorB] = 0;
569       MSMMotor(mmotor_S1_2, 0);
570       MSMMotor(mmotor_S1_1, 0);
571       waitSeconds(0.75);
572     }
573   }
574 }
575
576 /**
577  * Checks if string input is profane
578  *
579  * @param input Char pointer to first letter in input word
580  * @return whether the word is profane or not
581  */
582 bool profanityCheck(char *input)
583 {
584   // Constant number of words that have been censored.
585   const int NUM_PROFANE = 7;
586   // Array of characters (string) to store text (MAX OF 21 CHARACTERS)
587   char cipheredWord[20] = " ";
588   // Initialize string to store array of chars as a string
589   string cipheredString = "";
590
591   // Pass each character of the array of chars to cipher it
592   for (int i = 0; i < strlen(input); i++)
593   {
594     cipheredWord[i] = cipherText(input[i]);
595   }
596
```

```
597   // Convert array of chars into one string
598   stringFromChars(cipheredString, cipheredWord);
599
600   // Array of profane words to be censored, all ciphered
601   string profaneWords[NUM_PROFANE] = {
602     "kzhp",
603     "xmny",
604     "gnyhm",
605     "xjc",
606     "hzsy",
607     "xqzy",
608     "bmtwj"
609   };
610
611   // Iterate through each word in profanity array
612   for (int i = 0; i < NUM_PROFANE; i++)
613   {
614     // If the string does contain profanity, display will not sign and return 1.
615     if (stringFind(cipheredString, profaneWords[i]) != -1)
616     {
617       displayString(3, "This word is inappropriate, will not sign.");
618       return true;
619     }
620   }
621
622   return false; // If not profane
623 }
624
625 /**
626  * Ciphers text, shifting characters 5 ahead in the alphabet
627  *
628  * @param input Character to shift
629  * @return shifted character
630  */
631 char cipherText(char input)
632 {
633   // Add 5 to each character
634   char ciphered = input + 5;
635
636   // If the character has gone past the alphabet, wrap around
637   if (input > 122)
638   {
639     input = input - 26;
640   }
641
642   return ciphered;
```

```
}

/**
 * Converts string to all lowercase and removes special characters
 *    'toLower()' function not available with RobotC
 *
 * @param input Char pointer to first character in string
 * @return None
 */
void toLower(char *input)
{
  // Iterate through each letter of the string
  for (int i = 0; i < strlen(input); i++)
  {
    // If character is a special character (i.e, not alphabet), remove.
    if (input[i] < 64 || (input[i] > 90 && input[i] < 97))
    {
      input[i] = ' ';
    }
    // If character is uppercase, change to lowercase.
    else if (input[i] < 90)
    {
      input[i] = input[i] + 32;
    }
  }
}
```