Department of Mechanical and Mechatronics Engineering

Robotic Bottle Cap Sorter

A Report Prepared For:

MTE 121/MTE 100

S8 - Group 5

Prepared By:
Tanay Jain (#21197865) , Dhyey Patel (#21197750) ,
Nitish Thumkunta Reddy (#21220722) , Prerak Mahajan (#21188148)

December 2nd, 2025

University of Waterloo
Waterloo, Ontario, N2L 3G1

December 2nd, 2025

Professors of MTE 121, MTE 100
Course Instructor, MTE 121
Department of Mechanical and Mechatronics Engineering
University of Waterloo
200 University Ave W.
Waterloo, Ontario, N2L 3G1

Dear Professors,

This report, titled Robotic Bottle Cap Sorter, was completed to fulfill the project requirements for MTE 121 and MTE 100 in the Mechatronics Engineering program at the University of Waterloo.

The project involved the mechanical and software development of an automated bottle-cap sorting system. Over the course of the term, our group designed, built, and tested a working prototype that integrates mechanical design principles with sensor-driven software control. The report describes the mechanical design process, software architecture, testing results, and overall system performance.

This work was completed entirely by the four group members listed on the title page. No external assistance was used during any part of the design, implementation, or testing. All references used while preparing this report are listed in the appendices.

Best regards,
Prerak Mahajan
On behalf of Stream 8, Group 5

Table of Contents

List of Figures and Tables:

Summary:

This report proposes a solution to improve the efficiency and reliability of small scale recycling and sorting systems. The project focuses on designing and implementing an automated bottle cap sorting robot capable of detecting, transporting, and classifying caps using onboard sensors and a custom pause and resume control system. Throughout this report, the mechanical design, software design, sensing strategy, testing procedures, and design decisions are discussed in detail to show how the system was developed and refined. The report concludes with an evaluation of whether the final prototype met the project constraints and criteria, along with recommendations for future improvements.

2. Introduction

As climate change acts as a constant threat to the environment, it's important to also acknowledge the little mistakes humans do, which also worsens the environment. The most basic mistake can be assorting the materials and waste to the right bin. Although it may seem easy to sort out recyclable materials with garbage, a major problem can be having plastic and metal within recycling bins. To recreate this scenario, an alternative option like metal and plastic bottle caps can be a great way to portray the problem. Some of the most common problems with mixing metal and plastic caps are cross-contamination. From the Association of Plastic Recyclers [1], they determined that metal caps on bottles can hinder the metal and plastic separation process, causing plastic bottles with metal caps to be detected as metal, due to the magnetic detection, during the MRF process (material recovery facility). Additionally, another process involves material being separated based on their ability to float. Hence, materials such as PP and HDPE which bottle caps are made of, will be differentiated from heavier plastic like PET which water bottles are made of. However, sometimes, since metal caps are also heavy, they fall along with PET materials, causing contamination issues with plastic and metal in the same area.

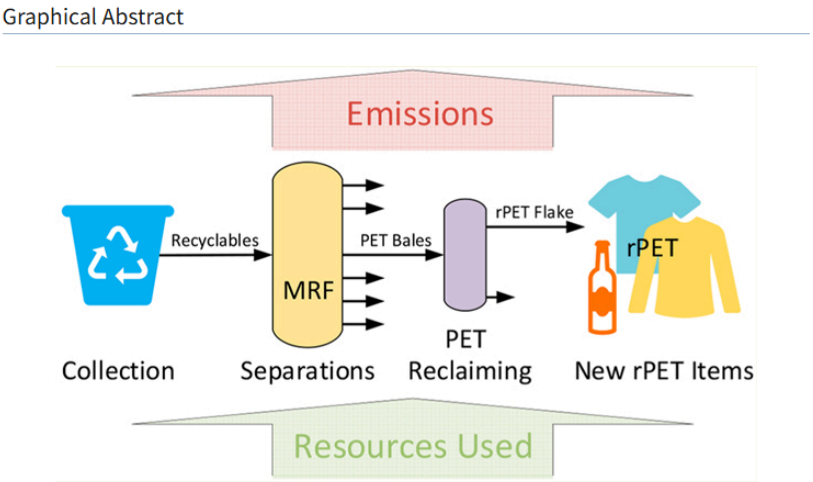

Figure 2.1

Hence, the main problem here is how sometimes plastic can be attached to metal, causing sorting issues, resulting in contamination. Considering the problem, sorting machines such as the float/sink process and the MRF magnetic sorting process can be quite inefficient. Hence, once all

PET material is sorted, it is then squished into a PET bale, as shown in Figure 1.0. However, due to the contamination, a statistic from the study by Smith et al. [2] states “Metals (i.e., Cr, Sr, Sb, Ti, Mg, Fe, Na, Al, and Ca) [...] were found at average concentrations of 512 […] ppm” in PET bottles. Then, in PET bales, Smith et al. [2] says they “outbound from an average MRF have 4.2% other materials in the bale”. The 4.2% contamination in PET bales include metals, which means that sorting processes are still not completely efficient. Therefore, alternatives must be considered. That is why this project aims to find a different way to differentiate plastic and metal, specifically bottle caps, through a reflectivity sorting system.

3. Scope

3.1 Task

The robot’s main task is to sort bottle caps such as plastic and metal, to avoid cross-contamination, when depositing them into bins, as a measure to protect the environment. The robot itself is a vex-based system following coordinated tasks, and with an additional 3-D printed feed. As for the tasks involved in sorting, they are separated into three smaller tasks. The first task is for the loader to pick up the bottle caps, one by one, from the feed, and drop it onto the conveyor belt. The second task is for the conveyor belt to move the cap, as it passes underneath the optical sensor. The third task is for the sorting arm to move left or right, guiding the bottle caps to the right bin (2 bins, one for metal, one for plastic).

3.2 Inputs

For the first task, the robot has a distance sensor, *Distance2* on Port 2, to detect whether a bottle cap is present on the feed. A smaller distance indicates that there is a bottle cap on the feed, a larger distance indicates that there isn’t. The next input will be for the second task, which is the optical sensor, *Optical6* on Port 6. The optical sensor has a feature, allowing it to detect reflectivity of a surface. A higher reflectivity indicates that it is a metal cap, a lower reflectivity is for plastic caps. Additionally, the third input will be from motor encoders. The first motor encoder is on the loader arm, *Loader* on Port 1. Once the distance sensor detects a cap, the motor will move at a certain rotation, and the program will track the motor rotation. The second motor encoder is in the sorting arm, *SortingArm* on Port 4. The motor will precisely move either right or left at a certain angle based on the cap’s material and the program will track the position of the

sorting arm. The last input is the E-stop button, which can be activated by pressing the Vex Brain button on the Brain Inertial.

3.3 Interaction with the environment and Usage of motors

The way the robot interacts with the environment is that it takes in physical bottle caps that are placed and then slid onto a feed, then moves them with a rotating arm, one by one, onto a conveyor belt. The conveyor belt acts as a moving pad for the cap, so that once it moves underneath the optical sensor, the optical sensor will collect data based on the cap's reflectivity. Lastly, the sorting arm will move the cap to its respective bin. Since the environment has different lighting that can affect the reflectivity data, the robot has a cover plate over the optical sensor for more optimal values for reflectivity.

The first motor, called Loader, has 2 arms/compartments separated by 180 degrees and will be used to move the bottle caps one by one from the feed onto the conveyor belt. It will move exactly 180 degrees once the cap has been detected, to avoid picking up a second bottle cap with the other arm. The conveyor motor *Conveyor* on Port 5 is attached with 2 other axles and wheels to a base, and they are covered with a belt, to act as a conveyor belt. This conveyor belt is used to move the bottle cap starting from the end of the Loader to the sorting arm, for a duration depending on a built-in timer. The sorting arm, which is the third motor, is placed over the bin, with the handle lying on the end of the conveyor belt. It will move exactly 10 degrees to the left or right, to guide and drop the right cap, based on its material, to its respective bin.

3.4 Recognition of completed task and Shutdown Procedure

The first task, highlighted in the Task section, will be recognized as complete, once the distance sensor detects a bottle cap, allowing the Loader to rotate. The second task is recognized as complete, once the in-built timer, which starts when the first task is done, runs for a set amount of time and stops, which also stops the conveyor motor and belt. The third task is recognized as complete when the sorting arm has moved left or right and when the in-built timer has been stopped.

For the robot to shutdown, there must be no more bottle caps present on the feed, which is detected by the distance sensor, for 30 seconds. Once completed, Brain Inertial will print "Process Complete" for a set time and then will shutdown the robot.

3.5 Changes made and Reasons

In terms of detection, the major change was going from detecting garbage and recycling, to simply detecting plastic and metal. The reason for this change is that VEX sensors cannot simply distinguish between garbage items and recyclable items, as there are tons of characteristics and factors that come into play. By cutting down the numerous possibilities of detection, detecting between metal and plastic was a safe option, as the optical sensor can safely detect reflectivity.

4. Constraints and Criteria

The team's project development was guided by the criteria of what the robot should achieve and the constraints which limited how it could be built. These developed from broad goals at the start of the term to more specific requirements as the mechanical and software parts started to get built and tested.

4.1 Design Criteria

Across the project, the key criteria for the system were:

- Sort metal vs. plastic bottle caps based on optical reflectivity.
- Operate autonomously from loading to sorting without manual control.
- Maintain stable, repeatable motor motion, with only one cap on the conveyor at a time.
- Achieve accurate sorting.
- Only runs when caps to sort are present.
- Return the sorting arm to its 0° position after each sort.
- Provide a predictable and safe user interaction, including a reliable emergency pause/resume.

These criteria became more precise as testing revealed the behaviours required for consistent performance:

- Accurate sorting became a concrete 80% accuracy threshold.
- Autonomous operation became clear behaviours such as idling until caps are present and automatic shutdown after inactivity.
- Stable motion became requirements like one cap on the belt at a time and sorting arm returns to 0°, because both affected mis-sorts.

- New practical criteria appeared, such as avoiding jams and ensuring safe, predictable behaviour when emergency stop was pressed.

4.2 Design Constraints

The main constraints throughout the project included:

- Only using approved VEX IQ and limited LEGO components, with no permanent modification or damage.
- Using only the allowed sensors (optical sensor, distance sensor, encoders, Brain button) and permitted VEXcode IQ functions.
- Designing within the structural dimensions and mechanical limits of VEX parts while operating safely.

As testing progressed, two additional practical constraints emerged:

1. Sensor delay and motor speeds:
   The optical sensor required a short delay to detect brightness accurately. The motor for the conveyor could not run too quickly, or caps would pass the sensor before a stable reading was captured. Additionally, the sorting arm motor needed to have enough time to sort and return to its original position. Overall, these created a new constraint of balancing speed with reliability for peak efficiency.
2. Reflectivity sensitivity and ambient lighting:
   Through testing the team realized the optical sensor's readings were easily affected by external light sources and cap orientation. To meet the 80% accuracy target, the team added a small physical shade around the sensor to block external light, relying solely on the LED from the optical sensor.

These constraints did not exist in the initial plan but became necessary to achieve efficient performance.

4.3 Most and Least Influential Criteria and Constraints

The constraints and criteria that most directly shaped the final design were:

- The single-cap requirement, which influenced feeder geometry and timing.

- The need for bottle caps to successfully land on and travel on the conveyor, which resulted in many mechanical adjustments.
- The two constraints discovered through testing: sensor delay and lighting sensitivity, both of which forced mechanical and software adaptations.

Some criteria were less influential:

- General energy efficiency was automatically satisfied once speeds were limited and shutdown was implemented.
- Basic robot stability stopped affecting design decisions once the base was wide enough.

Overall, the criteria and constraints that were specific and measurable, especially the ones that required testing and changes were the ones that most effectively guided the final design.

5. Mechanical Design and Implementation

5.1 Overview of the Mechanical System

The final design of the robot was inspired from industrial assembly-line structures that often follow a linear path, which helps break the overall task into smaller ones. The three sub-tasks that this robot completed in order to sort the bottle caps were: feeding, conveying, and sorting. Breaking the bigger process into smaller sections allowed each stage to be optimised and worked on independently while working towards the final goal. In terms of the physical components of the final design, the robot features a minimalistic chassis, a gravity-based feed track, a wheel feeder, a conveyor belt and a pivoting swing-arm, which is all controlled by a control panel.

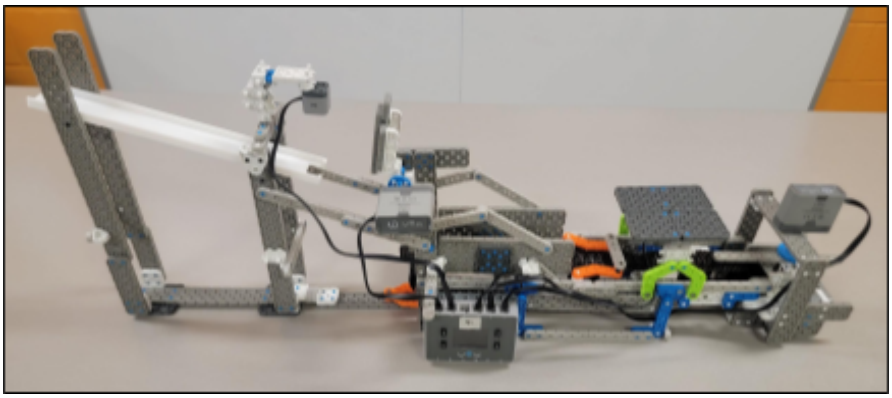

Figure 5.1

5.2 Chassis Design

The chassis used a modular ladder-frame structure, consisting of two symmetrical side walls that were connected by cross-members. These included three axles from the conveyor belt, one axle from the wheel feeder, a cross beam supporting the feeder subsystem and two cross beams supporting the chassis legs. Originally, the chassis was shorter, however lengthening the conveyor required the chassis to also become longer. However, this increase in length caused the system to sag near the center and the belt tension caused the chassis to deflect outwards. To address these issues, a support leg was added to each side of the chassis and the two were attached with cross members to prevent them from spreading.

Figure 5.2

5.3 Feeding Mechanism

5.3.1 Feed Track Design

The feed track was designed as a one-piece slide that was 3D-printed. The choice of 3D-printing in place of using vex parts came from the fact that the slide needed to be smooth so the bottle caps can easily slide on it and since it needed to be of custom size to ensure the bottle caps fit in a satisfactory way. The width and height of the slide were chosen so that only a single bottle cap could fit on the slide, ensuring only one bottle cap was fed to the conveyor belt at a time. The slide was mounted at an angle of 20° to allow gravity-based feeding, removing the need for a system to push the bottle caps towards the wheel feeder such as a spring. At the front of the slide, there was a short wall which held the caps and the loading end to allow the paddles on the wheels to lift the caps without interference. The bottom of the loading end featured a notch that was as wide as the paddles (with a +1 mm tolerance on both sides of the notch), allowing it to easily pass through and contain the caps.

The track was only attached to the scaffolding that held it at the opposite end of the notch. This was an intentional change that was implemented to allow flexibility in case the wheel snagged on the track. This was done since testing of the original design (where both ends of the track were attached to the scaffolding) showed that the slightest misalignment would cause the wheel to snag and cause structural damage. In reference to the main chassis of the robot, the scaffolding for the feed track was placed 148 mm away so that the back end of the notch would be within 1 mm of the paddle on the feeder wheel when it was parallel to the slide, to ensure the bottle caps were being lifted.

90 mm above the notch, a distance sensor was mounted to detect the presence of a cap. This sensor was mounted into the feed-track scaffolding to ensure the sensor was always aligned with the notch.

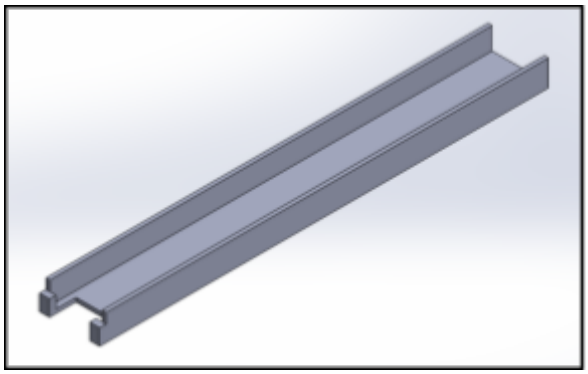

Figure 5.3

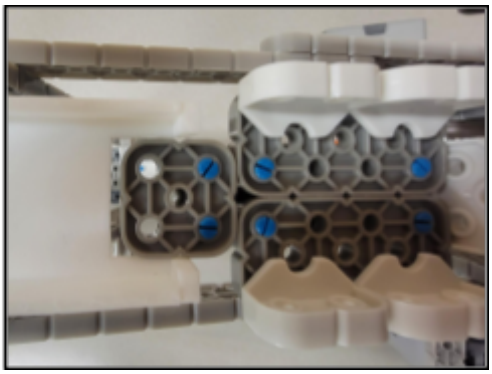

Figure 5.4

5.3.2 Wheel Feeder Design

The rotary feeder is made up of a vertically mounted wheel that has two paddles spaced 180° apart. The wheel is connected to an axle that is connected to the motor at one end and a passive support on the opposite end to stabilize the rotation. Both the motor and the support are connected to the chassis. The wheel feeder is designed so that every time it completes a 180° rotation, only one cap from the feed track is dispensed onto the conveyor.

Figure 5.5

The rotary feeder consists of a vertically mounted wheel with two paddles spaced 180° apart, driven directly by a single motor. The motor anchors one side of the axle, and a passive (loose) support on the opposite side stabilizes rotation. Each 180° rotation dispenses precisely one cap onto the conveyor. The original design called for six paddles on the wheel, however, while manufacturing the paddles, it became apparent that they were larger than anticipated. Since this would increase torque requirements and consequently reduce motor speed and braking performance, the final system used two paddles and rotated 180° instead of the originally intended 60°.

5.3.3 Barrier Systems

The first set of barriers on the robot ensured that the caps remained on the conveyor after being dropped from the feeder wheel. Since the caps landed in unpredictable orientations and sometimes bounced off the belt during operation of the robot, these wall-like barriers were rigidly attached to the chassis near the landing zone of the conveyor belt to prevent the caps from falling off.

The second set of barriers on the robot acted like a guide rail. Due to the limitations of the optical sensors viewing area, it was imperative that all caps were directly under the sensor. Since the caps landed at different locations on the conveyor belt, the funnel barriers were added to bring the caps to the middle of the belt so that they would be in line with the optical sensor. One side of each barrier was attached to the chassis while the other side was attached to the opposing rail so that the spacing between the two guides was narrower than the belt but slightly wider than the bottle caps, ensuring consistent placement of the caps.

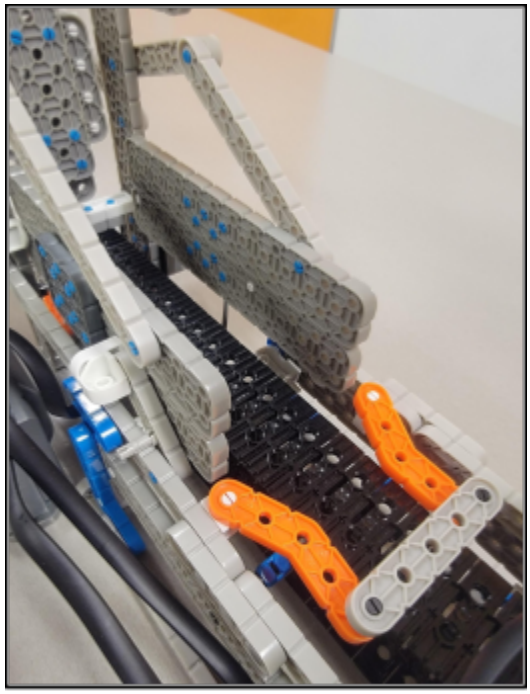

Figure 5.6

5.4 Conveyor Belt Design

The conveyor belt was used to translate the motors' rotational movement into linear movement over the 426 mm span of the belt. This was a single-motor-driven system that relied on three axles. Each axle had a sprocket on it to help support and guide the conveyor belt. The first axle was at the base of the feeding system and was connected to the motor on one end and a passive support on the other. The second axle was half way across the span of the belt to prevent the belt from sagging and was connected to the chassis with passive supports. The third axle was at the far end of the belt, near the bin. It was placed here using passive supports to keep the belt tight and maintain engagement with the sprockets.

Compared to the original design, the conveyor belt was doubled in length so that each part of the process can be physically separated. Although this was addressed by implementing this modification, the increased length caused the belt to sag in the center, requiring the addition of a central axle. Additionally, the conveyor motor was moved from the bin side to the feeder side to reduce clutter and counterbalance the wheel feeder motor, both improving the robots stability and preventing it from tipping over by lowering the robots center of mass on the feeder side.

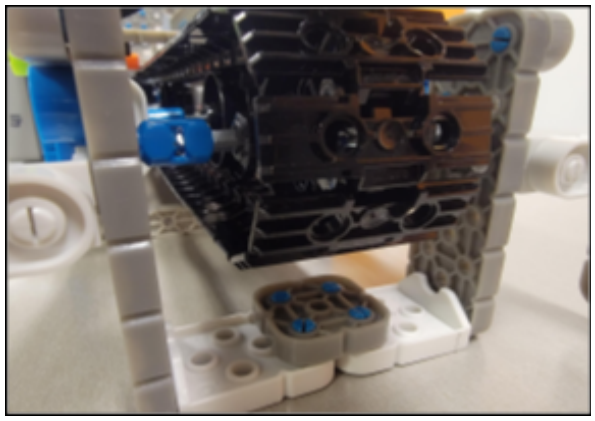

Figure 5.7

5.5 Sorting System

5.5.1 Optical Sensor Assembly Design

The optical sensor was mounted vertically, facing down at a height of 30 mm above the conveyor belt. This orientation allied readings from a consistent distance and kept the motor from

interfering with any moving parts such as the caps or the belt. An umbrella was also added on top of the sensor to block ambient light from affecting reflectivity measurements, ensuring the robot is usable without adjustment in most environments.

Before the final iteration of the sensor assembly was designed, two more configurations were considered. First, a forward facing sensor was considered, however, before robot manufacturing started, it was quickly understood that the sensor would get in the way of the sorting arm and bottle caps while also having inconsistent readings since the distance between itself and the caps would vary as the caps moved closer. The second iteration involved a side-mounted sensor that would look at the caps as they passed along the belt from the side. This idea was also rejected because in some cases minor displacements in belt height would prevent detection of bottle caps.

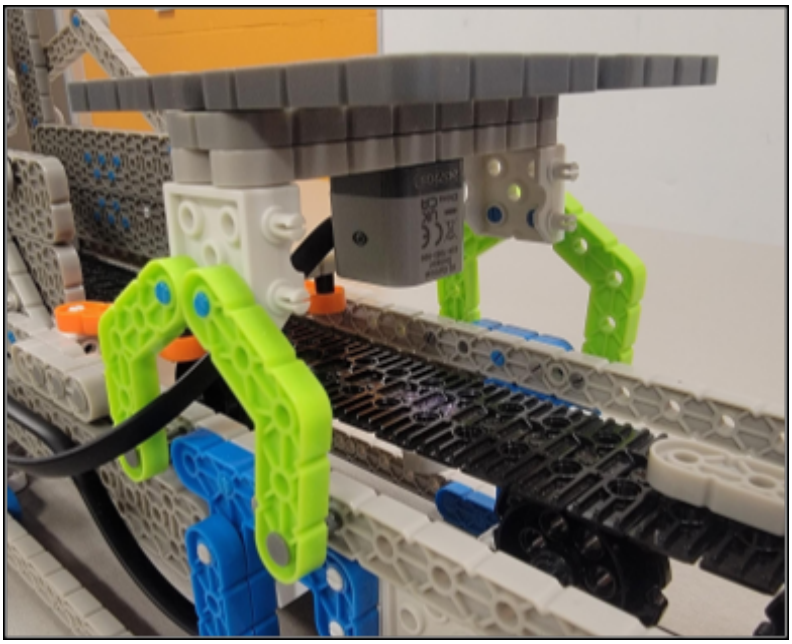

Figure 5.8

5.5.2 Swing-Arm Design

The sorting arm is a swing-arm that pivots at an axle which is powered by a vertically mounted motor. The axle is positioned so that ⅔ of the arm hangs 1 mm above the belt and ⅓ of the arm extends back past the pivot-point. The arm switched between +10° and -15° to guide the caps into their respective bins depending on their material.

As the original design failed to acknowledge the size of the swing-arm motor, the motor has been moved above the belt in the final design.

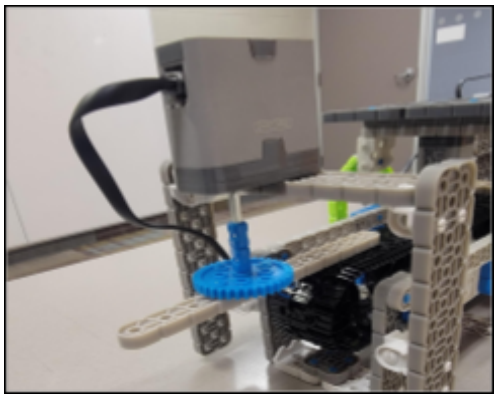

Figure 5.9

5.6 Control Panel Mounting

The VEX brain was mounted using a single pivot peg, allowing the control panel to rotate, similar to human-machine interfaces (HMI) found on many industrial machines. On this control panel, the check-mark button functioned as an emergency stop (E-stop), which would pause and resume the motors without resetting the cycle every time it was pressed.

Although the initial plan was to rigidly fix the panel onto the chassis, the pivoting redesign improved the user experience and ergonomics of the robot without sacrificing structural integrity, as it allowed for easier access to the program and E-stop controls.

Figure 5.10

6. Software Design & Implementation:

6.1 Overview:

The final software design was purposefully split into smaller, easily recognizable portions. Each of these parts is responsible for one aspect of the full process of sorting bottle caps. The modular design makes the software easier to understand, test, and troubleshoot. As the robot must react to both expected and unexpected inputs from their environment, splitting the program into small, self-contained parts greatly reduces the chance of errors and makes issues easier to isolate. For this reason, the final software was organized into six core sections.

6.1.1 Sensor initialization and system setup:

This section configures every sensor and motor into a known, safe starting state before any motion occurs. It includes:

- Calibration of the inertial sensor
- Zeroing the conveyor, sorting arm, and loader positions
- Turning on the optical sensor LED for consistent detection
- Setting up the Brain screen for debugging output

The Sensor initialization and system setup block allows for reliable control and starting conditions. Without a dedicated initialization step, every following task would behave unpredictably. Separating initialization prevents accidental movement and ensures all hardware is aligned before operation of the actual program.

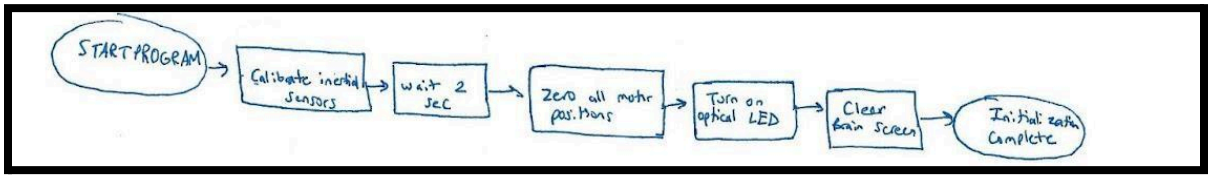

Figure 6.1

6.1.2 E-Stop System

Although no global variables were allowed throughout this assignment, the program maintains global-equivalent variables stated inside the main() function and passed as references into other functions.

The pause system revolves around:

- Detecting button presses
- Toggling between “running” and “paused” states
- Freezing all motors immediately when paused
- Adjusting timers so the conveyor cycle duration is not affected by pauses
- Displaying clear feedback (“E STOP ACTIVATED”)

The robot must be physically safe at all times. A fully working emergency stop system that can pause and resume the entire program is difficult without introducing tangled control logic that is unreliable under real time conditions. Designing a software block solely dedicated to an emergency stop made it possible to satisfy safety requirements and project constraints while keeping behavior predictable.

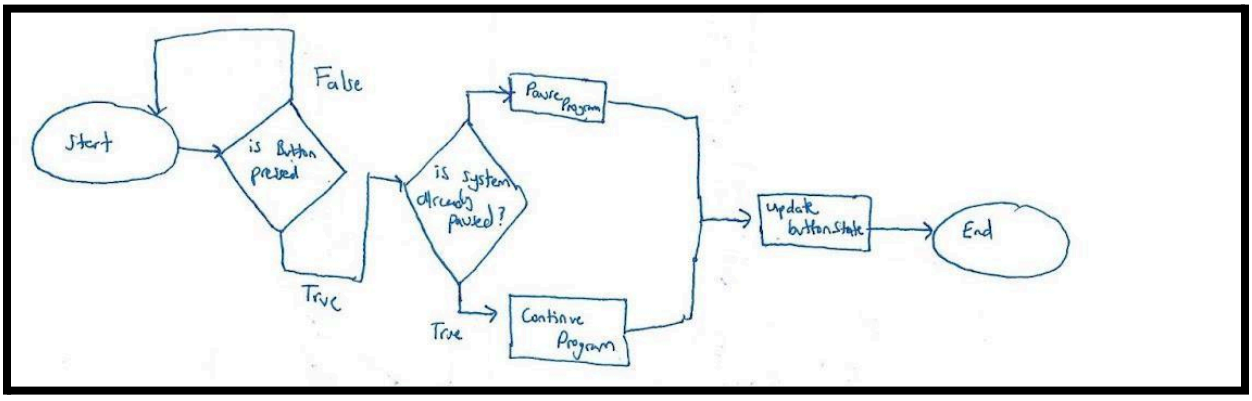

Figure 6.2

6.1.3 Cap Detection and Loading

This portion of the program handles everything from first detecting that a bottle cap is present to feeding it into the conveyor.

Major responsibilities:

- Checking the distance sensor to identify whether a cap is loaded
- Running the loader motor to push a cap into the conveyor
- Interrupting movement immediately if an E-Stop occurs
- Ensuring the loader motion finishes cleanly before continuing

The robot should not start sorting a bottle cap until one has been properly inserted into the mechanism. Splitting the loading into its own function isolates mechanical motion that could otherwise interfere with the detection or sorting logic.

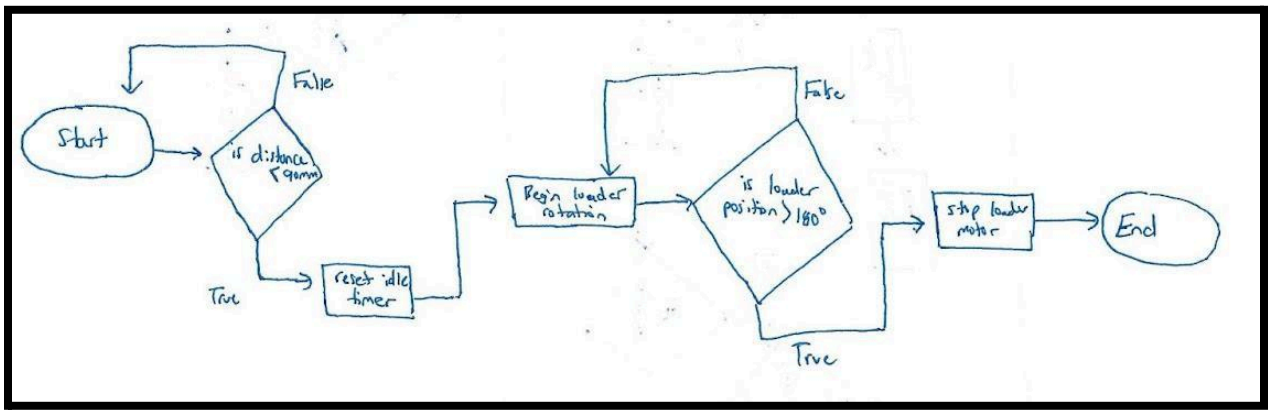

Figure 6.3

6.1.4 Conveyor Cycle Management

This is the core of the sorting mechanism: a 12-second cycle during which the conveyor runs, sensor readings are collected, and the final sorting decisions are made.

This block handles:

- Starting and maintaining the conveyor motion
- Running an internal cycle timer
- Measuring brightness values from the optical sensor
- Identifying when the cap enters and leaves the field of view
- Determining material based on peak brightness
- Ensuring timing remains exact even if paused

The conveyor cycle needs to be isolated so that timing, detection, and safety controls remained stable and predictable. Containing the cycle logic in one place also made debugging the optical sensor and timing interactions much simpler.

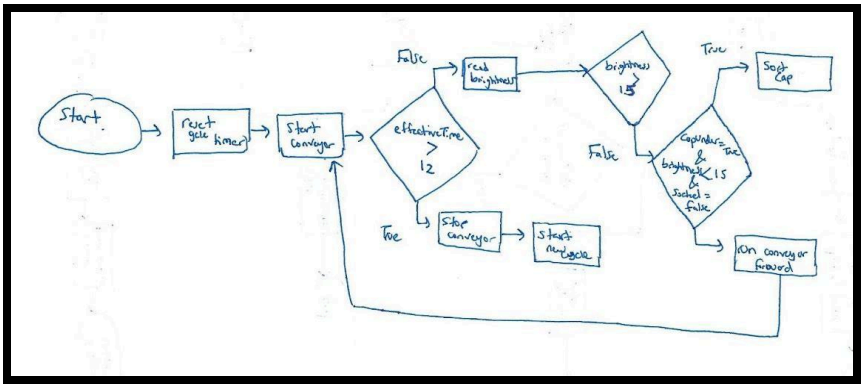

Figure 6.4

6.1.5 Sorting Decision and Sorting Arm Control

Once the peak brightness for the cap has been recorded, the sorting block handles moving the sorting arm:

- Applying thresholds to classify plastic vs. metal
- Moving the arm to +10° for plastic or -10° for metal
- Pausing and resuming motion
- Tracking arm direction for later resetting
- Ensuring the sorting action triggers exactly once per cycle

The sorting arm requires precise and discrete movement. Including this logic inside the conveyor loop would have introduced conflicts between conveyor timing and arm timing. Isolating it into its own section ensures consistent classification and reduces the chance of accidental double-sorting.

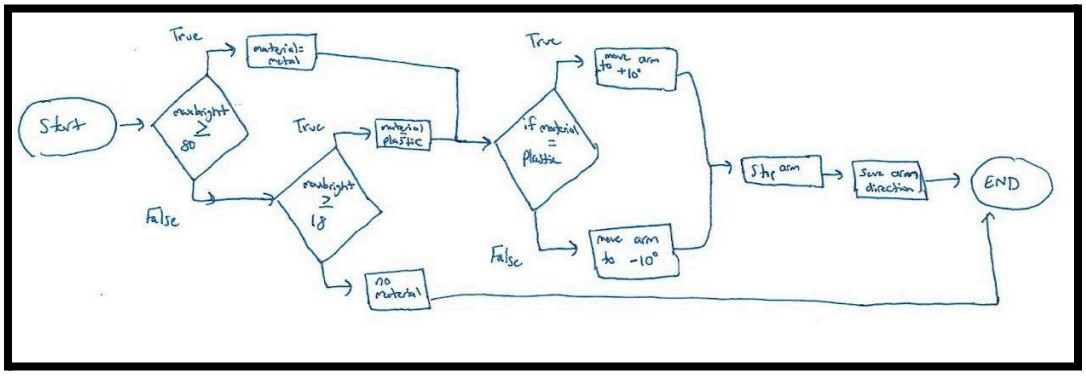

Figure 6.5

6.1.6 Post-Cycle Reset and Arm Re-Centering

After the 12-second cycle ends, the program ensures the system is ready for the next cap:

- Stopping the conveyor
- Returning the sorting arm to 0°
- Displaying diagnostic information (brightness peak, distance reading)
- Allowing the idle timer to restart

The robot must be able to repeat cycles reliably. Resetting mechanisms after each cap ensures repeatability and prevents errors from accumulating over multiple cycles.

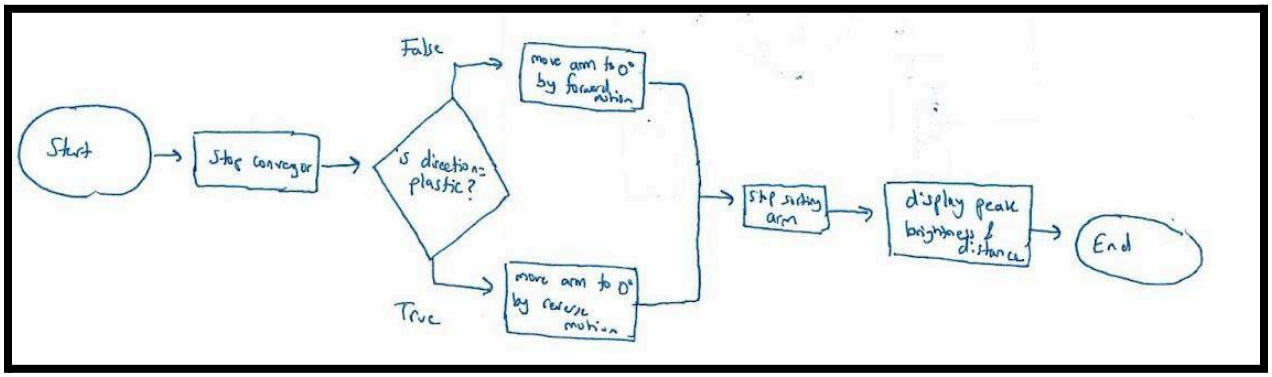

Figure 6.6

6.2 Task List

This is the final checklist of tasks the robot needed to do in the final demo. Throughout the course of this project, the ta\sk checklist changed slightly due to the team testing hardware designs and constraints of the VEX IQ components. The team originally planned on sorting more than 1 category of plastic and metal through the optical sensors, but when tested, found the brightness of the sensor was limited to detecting only 1 type of material at a time. Once the team determined this limitation, the plan change to sorting only between plastic caps vs. metal. By doing this, the team still met the original goal while improving the ability to accurately implement this part of the project.

Another major change to the robot's operation was how caps would be released from the feeder. In the initial task list, the feeder would release multiple caps at a time onto the conveyor belt. In

practice, this caused many jams, inconsistent spacing, and made it difficult to guarantee safe and predictable behaviour of the robot. Releasing only 1 cap at a time greatly reduced the risk of cap jams, improved trouble-shooting, and allowed better tracking of the cap throughout the entire process.

The final task list below reflects these changes and defines how the program behaves from startup to shutdown:

- Startup behaviour: After the program starts, absolutely nothing happens until a cap is detected in the feeder. This ensures the robot never moves unexpectedly and only begins work when the system has material to process.
- Single-cap processing: Only one bottle cap is loaded at a time, and the conveyor never carries two caps simultaneously. This guarantees that the optical sensor and sorting arm always receive clean, isolated readings.
- Sorting accuracy requirement: The sorting arm must correctly classify and sort at least 80% of all caps into the correct bins.
- Sorting arm reset: After a sorting action is completed, the arm must always return to its calibrated 0 degree position so that every cycle starts from a consistent baseline.
- Emergency stop behaviour: Pressing the ESTOP button pauses the entire program, and pressing it again resumes operation. This guarantees safe interruption without terminating the program.
- Automatic shutdown: If no caps are detected for 30 seconds, the robot shuts down on its own. This prevents unnecessary motor use and removes any need for human intervention.
- Loading mechanism accuracy: The loading arm must spin, collect, and dispense caps with at least 80 percent accuracy, ensuring the rest of the cycle receives consistent input.

These tasks form an overarching goal of how the robot should behave during the final demonstration and guide how the team structured the software and testing procedures.

6.3 Functions

Name	Parameters	Return Type	Description
checkPause() Author: Dhyey	bool &estopPaused, bool &lastButtonState, timer &cycleTimer, double &pausedOffset, double &pauseStart, bool cycleActive	void	This function is imperative in allowing an emergency stop to exist. Rather than terminating the program entirely, the Emergency Stop feature allows users to safely pause and resume their program at a later time. The Emergency Stop feature responds by allowing users to toggle between a paused or active state each time the Brain Button is pressed. The Emergency Stop feature will stop all motors immediately when the program is in a paused state, will show "E STOP ACTIVATED" on the brain screen, and once you exit the paused state, will continue the program exactly from where you left off. Additionally, if the robot is doing a sorting cycle during a pause state, the Emergency Stop feature will keep track of the amount of time that the robot spends in a pause state so that the 12 seconds of timing do not affect the conveyor cycle. As a result, even if the robot pauses in the middle of an operation, the conveyor cycle will remain accurate.
uWait() Author: Prerak	int ms, bool &estopPaused, bool &lastButtonState, timer &cycleTimer, double &pausedOffset, double &pauseStart, bool cycleActive	void	This special wait function replaces the normal blocking wait times with a series of tiny waits (10ms). Instead of freezing the program, it performs a series of small waits and checks the emergency stop button after each small wait. If the robot has been paused, its time will no longer be moving forward; if it is running normally, it will continue its normal time for the wait. In the sorting process, any duration during which the robot was paused will be subtracted from the total sorting time. As a result, the conveyor will always receive 12 seconds of continuous real movement for sorting. This makes the program completely responsive and prevents the freezing of the robot while it is processing important tasks.
configureAllSensors() Author: Nitish	bool &estopPaused, bool &lastButtonState, timer &cycleTimer, double &pausedOffset, double &pauseStart	void	This function prepares all physical components of the system, including calibrating the Inertial Sensor for two seconds, resetting the positions of the conveyor, loader, and arm, and turning on the LED in the Optical Sensor. It also clears the Brain display and establishes the font for all messages sent by the system, ensuring each run of the robot begins with an identical and predictable combination of physical characteristics, which are critical for hosting a safe operation.

capLoaded() Author: Tanay	none	bool	The sensor reads the distance and determines if there is a cap present in the loading mechanism. It looks for distance readings less than 90mm as an indication of a cap being present; if not, then no action will be taken. The function triggers the loading process to begin with the detection of one cap only when the distance reading is less than 90 mm (millimeters). It returns true if a cap is present, or false if it isn't.
loadCap() Author: Dhyey	bool &estopPaused, bool &lastButtonState, timer &cycleTimer, double &pausedOffset, double &pauseStart	void	This function moves the loading arm motor at 15 percent power for each 180 degree cycle. During each cycle, it will monitor for emergency stops. If the emergency stop time occurs, the loader motor will stop where it is and automatically start back where it stopped, allowing the loader to operate in a controlled manner. The loader loads caps at only one time, so this method prevents jams and allows for consistent timing for the sorting operation.
sortCap() Author: Tanay	int material, bool &estopPaused, bool &lastButtonState, timer &cycleTimer, double &pausedOffset, double &pauseStart	int	This function sorts bottle caps after they are detected by the optical sensor. After the cap is assigned by the optical sensor to its respective category, the sorting arm moves to the proper position. The sorting arm will moves at 40% speed. The sorting arm rotates 10 degrees clockwise if it is sorting a plastic (1) cap, and rotates 10 degrees counterclockwise if it is sorting a metal (-1) cap. If the system is paused, the sorting arm will immediately stop movement. After completion of the sorting function, the function output returns the direction in which the sorting arm should return.
resetSortingArm() Author: Nitish	int direction, bool &estopPaused, bool &lastButtonState, timer &cycleTimer, double &pausedOffset, double &pauseStart	void	This function returns the sorting arm back to its original position (0 degrees). It moves at a speed of 20%. If the arm is located at +10 degrees for example, it will rotate in the reverse direction until it reaches 0 degrees. Conversely, if the arm is at -10 degrees, it will rotate in the forward direction until it reaches 0 degrees. Similar to every other function that allows movement, it will respond to a pause command at any time during the movement process by immediately halting movement and resuming when commanded to do so.
main() Author: Prerak	none	int	The main function is a complete controller that performs the following functions: Turning on the system and checking to see whether any caps are present; loading any detected caps that are present; starting a conveyor belt cycle for 12 seconds; using optical sensors to track which

			the brightness of each bottle cap; distinguishing between Plastic Cap (brightness≥18), Metal Cap (brightness ≥80), or no Cap; moving the arm to sort the identified caps. This function also ensures that one cap is kept on the conveyor at any given time and resets the arm position to 0 degrees after the task has been completed. The function also provides a 30-second Automatic Shut Off feature that shuts down the system if no caps are captured within 30 seconds. Lastly, the function continually monitors the Emergency Stop button and, when pressed, allows the system to safely stop and restart from that point.

Table 6.1

6.4 Data Storage

The program uses simple and predictable storage strategies so that all information remains easy to follow and safe to modify. The following types of data are stored and updated throughout the software:

6.4.1 Sensor Readings

- Distance sensor values are stored as temporary double values when needed.
- Brightness readings from the optical sensor are stored in an int called brightnessNow, and the peak value of each cycle is stored as maxBrightness.
- These values are not kept long-term; they are recalculated each cycle to ensure accuracy.

6.4.2 System State Flags

Several bool values track the robot's state:

- estopPaused keeps track of whether the system is currently paused.
- lastButtonState stores the previous reading of the Brain button to detect press events.
- capUnder and sortedAlready track whether the cap has passed the sensor and whether sorting has already occurred.
- programRunning, cycleRunning, and loadRunning are used to control different stages of execution.

All of these are local to main or passed by reference so that the emergency stop system always works consistently without the use of global variables.

6.4.3 Timing Values

The program manages time using:

- idleTimer to measure inactivity.
- cycleTimer to control the 12-second conveyor cycle.
- pausedOffset to subtract paused time from the conveyor's cycle.
- pauseStart to record the moment the emergency stop was activated.

All timer-related values are stored as double when needed so they can be adjusted precisely while the ESTOP is active.

6.4.4 Movement and Sorting Data

- The direction of the sorting arm (int direction) is stored so that the system knows how to reset the arm.
- The classification of the cap (material) is stored briefly inside the cycle loop.

Because the project forbids global variables, all state information is passed through function parameters by reference:

- This allows each function to modify shared program state safely.
- It ensures the ESTOP, timers, and cycle logic all stay synchronized.
- No hidden data or side effects are created.

6.4.5 Parameter Passing

Because the project forbids global variables, all state information is passed through function parameters by reference:

- This allows each function to modify shared program state safely.
- It ensures the ESTOP, timers, and cycle logic all stay synchronized.
- No hidden data or side effects are created.

6.4.6 No Dynamic Memory

All variables are statically allocated as simple C++ primitives. No arrays, pointers, or dynamic memory are required. This keeps the design predictable and avoids memory issues during long demo runs.

6.5 Software Design Choices

6.5.1 Pause-and-Resume ESTOP Instead of a Hard Stop

The emergency stop was designed so that pressing the button pauses the entire system and pressing it again resumes operation from the exact point it left off. A full shutdown stop would require restarting the entire program every time, which is unsafe during testing and makes troubleshooting harder. The pause-and-resume design keeps the robot safe while maintaining progress and preventing sudden resets.

Trade-off:

This required developing additional logic to prevent timers from advancing during pauses and ensuring every function cooperatively checks the pause state, making the code more complex. The benefit is a predictable, safe, and user-friendly safety system.

6.5.2 Peak Brightness Classification Instead of Real-Time Thresholding

The optical sensor produces highly variable readings due to angle changes, cap reflectivity, and motion. Instead of classifying based on a single brightness reading, the software continuously samples values while the cap is under the sensor and stores the highest (peak) brightness.

Trade-off:

The system must gather multiple samples, track the highest value, and wait until the cap leaves the sensor to classify it. This increases computational logic slightly but dramatically improves accuracy and ensures reliable identification of plastic versus metal.

6.5.3 One Cap at a Time Instead of Continuous Loading

The original concept involved releasing multiple caps onto the conveyor at once. Through testing, this approach caused jams, overlapping readings, and safety risks. The final design loads and processes only one cap at a time.

Trade-off:
This reduces overall throughput, but it ensures predictable sensor behavior, prevents mechanical collisions, simplifies error detection, and increases safety. It also makes debugging significantly easier because problems can be isolated to a single cap.

6.5.4 Time-Based Conveyor Cycle Instead of Position Tracking

The conveyor runs for a fixed 12-second cycle rather than using encoders or additional sensors to track cap position. The optical sensor alone cannot reliably determine the cap's exact location after detection because the conveyor speed, friction, and cap orientation vary.

Trade-off:
A fixed-time cycle must be long enough to handle the slowest case, which slightly reduces efficiency. In return, the system becomes much simpler, more predictable, and easier to test with consistent results.

6.6 Testing

Test Category	What Was Tested	Why This Test Was Run	Expected Behavior	How Correctness Was Verified
Sensor Calibration & Input Readings	Distance sensor readings and optical brightness readings	Ensure the robot reacts properly to its environment and can identify when a cap is present	Distance sensor correctly detects when a cap is within 90 mm; optical sensor detects relative brightness differences	Manually placed bottle caps under sensors and observed live Brain screen outputs until consistent values were produced

Threshold Discovery & Material Identification	Determining reflectivity ranges for bottle caps (plastic vs metal)	Establish real-world threshold values before coding logic	Plastic produces medium brightness, metal produces high brightness, conveyor belt produces low/black values	Repeatedly placed caps under sensor, wrote down brightness values printed on the Brain screen, and validated that the program's classification matched those readings
Loader & Conveyor Motion Tests	Loader arm rotation to dispense a cap; conveyor belt movement during sorting cycle	Ensure bottle caps load properly and move under the optical sensor smoothly	Loader stops around 180°, one cap released at a time; conveyor runs for correct duration	Ran multiple cycles with actual bottle caps and visually monitored positions and timing
Sorting Arm Functionality	Sorting motion to +10° (plastic) or –10° (metal), then resetting to 0°	Confirm correct bin sorting and return to starting position	Sorting arm rotates only once per cap, reaches the correct angle, and returns to 0°	Used markings on arm to verify angles and watched multiple full cycles for reliability
Pause-and-Resume ESTOP Testing	Pressing the Brain button during all stages of program execution	Ensure safety system can pause the entire robot and resume seamlessly	All motors stop immediately; program continues from the same point when resumed	Pressed ESTOP repeatedly during loading, scanning, sorting, and resetting, checking that program state was preserved
Full System Cycle Testing	Integration test of startup, loading, scanning, sorting, and shutdown	Validate correct behavior when all components run together	Robot waits for cap, loads one cap, sorts it, resets arm, and shuts down after 30 seconds of inactivity	Ran complete runs with several caps and confirmed consistent performance and proper shutdown timing

Table 6.2

6.7 Significant Issues

The modular structure of the program was beneficial when developing and testing most components of the program because most pieces of code were fairly simple to create and test. However, one of the most challenging aspects was how to implement an emergency stop for pausing the operation of the motors, not just stopping the motors but also pausing all the logical processing within the program, including timer functions, loading functions, and conveyor cycle operations. This traditional method of halting the timers was insufficient because they continued to run even while halted. To solve this problem, the team introduced a pause state, along with two variables (pausedOffset and pauseStart) which are passed by reference to all functions in the program, enabling them to properly account for time that has passed while they are in a paused state so that they can continue execution at the exact same point they would have reached had they not been paused. Another problem was to ensure that the conveyor system only sorted one cap at a time, and did not advance through multiple cycles if there were no caps. To assure this, the team coupled the distance sensors reading closely with the programs' conditions for starting and for each cycle so that when there is a detection of a cap, the program shall only continue processing when the cap is truly detected. These solutions allow the finished program to operate reliably and within reasonable expectations while being compliant with the functional and safety specifications of this project.

7. Verification:

To ensure that the constraint of only using VEX and Lego parts without damaging them was met, the group used creative strategies to overcome challenges. One such strategy was making bigger parts out of smaller parts when supplies were running low. For example, for the distance sensor mount, three connectors were used to make a vertical column since there were not any more pieces available that would be usable for that purpose. Another strategy used by the group was anything that could damage VEX or Lego components that do not directly come in contact with them. For instance, when hot-glueing the feed track to the scaffolding, the group wrapped the support beam with tape so the glue does not melt the pieces.

Another constraint the group had to meet was working with the approved sensors and functions. To ensure this constraint was met the group used the fundamentals of how functions and sensors

worked to achieve their goals. In terms of the physical build, to detect whether the bottle cap was made of plastic or metal, the group used the way the optical sensor detects proximity to determine the material. The group harnessed the fact that the optical sensor measures the intensity of reflected infrared light to determine proximity to determine the material of the bottle cap, since metal would reflect more light than plastic [3]. In terms of only using functions that are allowed, the group once again broke down the functionality of functions to achieve their goal. For example, instead of overloading the motor.stop() function to brake the motors, they first set the stopping mode to brake and then used the non-overloaded stop function.

To meet the constraint of keeping the speed safe yet efficient, the group conducted testing with several combinations of motor speed to determine the most ideal speed. The ideal speed can be defined as the speed of a motor which results in safe operations while not wasting time or energy. They also reduced motor speeds to address the fact that VEX motor encoders and sensors have inherent delays and inaccuracies.

Meeting the constraint of being able to address ambient light was crucial because the robot's performance and accuracy depended on this. To meet this constraint the optical sensor was outfitted with an umbrella that would block ambient light, so that the reflectivity readings remain consistent regardless of ambient light positioning or intensity.

8. Project Plan

The team created a weekly project schedule that aligned with MTE 100 and MTE 121 goals. The team organized regular Thursday work sessions, alongside online coordination and meetings before deadlines. These meetings were used to plan concepts, and discuss mechanical prototypes, sensor integration, software, and the testing process. This schedule provided the team with structure and kept the project moving fluidly.

The project roles were organized using a team-oriented model rather than strict separated roles. All members contributed to construction, wiring, and testing, and each member wrote at least one non-trivial C++ function. To increase efficiency, each software function did have a primary

lead, but then all members collaborated to put together the functions into one functioning program. The primary areas of focus for each team member are summarized in Table 8.1.

The actual process mostly followed the original plan: concept selection and kit sign-out early on, then mechanical prototyping throughout October, sensor configuration and coding through early November, and finally testing and debugging before the final demo day. The main changes happened due to testing during the mechanical building process which disclosed necessary feeder redesigning and sensor-angle adjustments. The full week-by-week schedule is provided in Appendix B – Project Schedule. The project showed that mechanical testing and sensor calibration require more time than expected. In the future the team knows that starting mechanical experimentation and integration earlier would likely reduce the pressure of having both software and mechanical aspects working by the deadline.

Primary Lead	Mechanical	Software
Dhyey	Mounted and assembled storage facility for the unsorted bottle caps. Built feeding motor to collect bottle caps from storage.	Implemented bottle cap detection using distance sensor to begin the robot's task. Developed the loading mechanism for using the feeding motor to put bottle cap on conveyor.
Prerak	Mounting optical sensor into the optimal position. Assembling the conveyor belt to align all of the components.	Collecting and implementing values of reflectivity to calibrate optical sensor. Developing an emergency stop button to pause/resume robot software.

Tanay	Assembled the sorting motor and its facilities along the end of the conveyor belt. Created storage container for sorted bottle caps	Ensured the sorting arm is moving to the correct positions depending on values of reflectivity read by the optical sensor. Implemented a timer that runs the conveyor built for an appropriate amount of time when the distance sensor detects bottle caps.
Nitish	Mounted the Brain for optimal wiring and easy access of emergency stop button. Connected all sub-components into one fluent robot.	Developed startup function to calibrate all sensors and motors in the necessary ways. Designed the feature to shut down the robot when no bottle caps are detected for 30 seconds by distance sensor.

Table 8.1

9. Conclusions

The goal of this project was to design and build an autonomous system capable of sorting metal and plastic bottle caps using VEX IQ hardware. This addresses the broader problem of inefficient and inaccurate sorting of recycling in large-scale facilities. The robot displays how reflectivity-based sensing and mechatronic mechanisms can sort recycling efficiently without a need for manual labour.

The final prototype successfully met the majority of the project's constraints and criteria. The robot consistently remained idle until bottle caps were detected, sorted only one cap at a time, and achieved sorting and loading accuracies upwards of 80%, meeting the task criteria established for the final demo. The system also behaved predictably with usage of the emergency stop and it shut down automatically after no bottle caps were detected for 30 seconds. Constraints such as sensor delay, external light, and bottle cap alignment were addressed through mechanical tuning and software adjustments, allowing the robot to operate reliably throughout all of the team's testing.

Mechanically, the design had a 3D printed feeding track with a feeding mechanism, a conveyor system, and a funneling system that transported caps safely and minimized jams. The addition of a small shade above the optical sensor improved reflectivity readings in different environments by reducing the impact of external light sources. The software centered around non-trivial functions that handled cap detection, movement of the conveyor, optical sensor readings, sorting-arm positioning, and safe shutdown.

Overall, the project showed that a fully autonomous, reflectivity-based bottle cap sorter can be implemented effectively using simple hardware and well-structured software. While there are opportunities for both improved efficiency and effectiveness, the final design successfully met the primary criteria and serves as a strong foundation for future development.

10. Recommendations

The prototype met its main goal of sorting metal and plastic bottle caps, but implementing several recommendations could increase reliability and make the design more applicable to a real-life recycling facility. The recommendations below focus on mechanical changes to the sensors, feeding system and funneling, and software changes to improve efficiency and functionality.

10.1 Mechanical design

Improve funneling:

Adjusting the funneling and guide walls so that caps cannot bounce out, land beside the conveyor, or get stuck at the edges. A better path for bottle caps would give more consistent alignment and reduce jams.

Redesign the feeder to reduce errors:

Adjust the feeders throat width, shape, and drop height so the motor does not occasionally fling caps in unpredictable directions. A more controlled feeding system would ensure each cap lands on the conveyor in a repeatable way.

Make the feeder adjustable for different cap sizes:
Creating a universal style for guide walls and a different feeding style so that the robot can handle bottle caps with different diameters and shapes would allow for more applications of the sorting machine. It would make the system more representative of a real mixed recycling system.

Distance sensor placement for more accurate readings:
Due to a lack of resources and longer wires the distance sensor was not able to be placed parallel with the feeding container, leading to misreadings of distances at times. Fixing this would allow more accurate data on whether bottle caps are in the feeding container or not.

10.2 Software Design

Include multi-cap handling for higher efficiency:
Modify the program so multiple caps can be in the system at once, rather than a very linear program where one action happens at a time. Allowing overlapping operations would increase efficiency in a practical setting.

Replace purely time-based movement with sensor-based control:
Instead of running the conveyor on a timer, using a distance sensor to track how far along the track the bottle cap is would increase efficiency. The conveyor can turn off as soon as the distance sensor detects that the current bottle cap is sorted.

11. Backmatter

11.1 References

[1] Association of Plastic Recyclers, “Caps On FAQ”, 2021, Accessed date: November 29, 2025 [Online]. Available: **https://plasticsrecycling.org/images/library/APR-Caps-On-FAQ.pdf**

[2] R. L. Smith, S. Takkellapati and R. C. Riegerix, “Recycling of Plastics in the United States: Plastic Material Flows and Polyethylene Terephthalate (PET) Recycling Processes” in *ACS sustainable chemistry & engineering,* 2022, vol 10, issue 6, p. 2084-2096. Accessed date: November 29, 2025, DOI https://doi.org/10.1021/acssuschemeng.1c06845.

[3] VEX. Using the IQ Optical Sensor. [Online] Available: https://kb.vex.com/hc/en-us/articles/

Appendix A: C++ Code

```
#pragma region VEXcode Generated Robot Configuration
#include "vex.h"
using namespace vex;

brain Brain;

inertial BrainInertial = inertial();
optical Optical6 = optical(PORT6);
motor Conveyor = motor(PORT5, true);
motor SortingArm = motor(PORT4, false);
motor Loader = motor(PORT1, false);
distance Distance2 = distance(PORT2);

void vexcodeInit() {}
#pragma endregion


// ======================================================
//  PAUSE / E-STOP TOGGLE SYSTEM
// ======================================================
// This checks whether the brain button is pressed,
// toggles pause on/off, freezes motors, and adjusts timing.
//
void checkPause(bool &estopPaused,
bool &lastButtonState,
                timer &cycleTimer,
                double &pausedOffset,
                double &pauseStart,
                bool cycleActive)
{
    bool buttonNow = Brain.buttonCheck.pressing();

    // When button is pressed for the first moment (rising edge), toggle pause
    if (buttonNow && !lastButtonState)
    {
        estopPaused = !estopPaused;

        if (estopPaused)
        {
            // Freeze all movement immediately
            Conveyor.stop();
            Loader.stop();
            SortingArm.stop();

            // Record pause start time if we're inside a conveyor cycle
            if (cycleActive)
            {
                pauseStart = cycleTimer.value();
            }

            Brain.Screen.clearScreen();
            Brain.Screen.setCursor(1, 1);
            Brain.Screen.print("E STOP ACTIVATED");
        }
        else
        {
            // When resuming during a cycle, subtract paused time
            if (cycleActive)
            {
                double now = cycleTimer.value();
                pausedOffset += (now - pauseStart);
            }
        }
    }

    lastButtonState = buttonNow;
}


// ======================================================
//  PAUSE-AWARE WAIT
```

```
// ====================================================
// Waits without counting paused time.
//
void uWait(int ms,
           bool &estopPaused,
           bool &lastButtonState,
           timer &cycleTimer,
           double &pausedOffset,
           double &pauseStart,
           bool cycleActive)
{
    int waited = 0;
    const int step = 10;

    while (waited < ms)
    {
        checkPause(estopPaused, lastButtonState, cycleTimer, pausedOffset, pauseStart, cycleActive);

        if (estopPaused)
        {
            // Do not progress time during pause — stay responsive
            wait(20, msec);
        }
        else
        {
            wait(step, msec);
            waited += step;
        }
    }
}


// ====================================================
// SENSOR + MOTOR INITIALIZATION
// ====================================================
// Calibrates sensors and puts all motors into known state.
//
void configureAllSensors(bool &estopPaused,
                         bool &lastButtonState,
                         timer &cycleTimer,
                         double &pausedOffset,
                         double &pauseStart)
{
    BrainInertial.calibrate();
    uWait(2000, estopPaused, lastButtonState, cycleTimer, pausedOffset, pauseStart, false);

    Conveyor.setPosition(0, turns);
    SortingArm.setPosition(0, degrees);
    Loader.setPosition(0, turns);
    Optical6.setLight(ledState::on);

    Brain.Screen.clearScreen();
    Brain.Screen.setFont(mono15);
}


// ====================================================
// CAP LOADED?  (distance < 90 mm)
// ====================================================
bool capLoaded()
{
    double distVal = Distance2.objectDistance(mm);
    return distVal < 90.0;
}


// ====================================================
// LOAD CAP INTO CONVEYOR
// ====================================================
// Spins the loader until 180 degrees, with pause support.
//
void loadCap(bool &estopPaused,
             bool &lastButtonState,
```

```
             timer &cycleTimer,
             double &pausedOffset,
             double &pauseStart)
{
    Loader.setVelocity(15, percent);
    Loader.setStopping(brake);
    Loader.setPosition(0, degrees);

    bool loadRunning = true;

    while (loadRunning == true)
    {
        checkPause(estopPaused, lastButtonState, cycleTimer, pausedOffset, pauseStart, false);

        if (estopPaused)
        {
            wait(20, msec);
        }
        else
        {
            double pos = Loader.position(degrees);

            if (pos > 180)
            {
                loadRunning = false;
            }
            else
            {
                Loader.spin(forward);
            }
        }
    }

    Loader.stop();
}


// ======================================================
// SORT CAP BY MAX BRIGHTNESS
// ======================================================
// material = 1  → plastic  (move to +10°)
// material = -1 → metal    (move to -10°)
//
int sortCap(int material,
            bool &estopPaused,
            bool &lastButtonState,
            timer &cycleTimer,
            double &pausedOffset,
            double &pauseStart)
{
    SortingArm.setVelocity(40, percent);
    SortingArm.setStopping(brake);

    if (material == 1)
    {
        bool sorting = true;

        while (sorting == true)
        {
            checkPause(estopPaused, lastButtonState, cycleTimer, pausedOffset, pauseStart, false);

            if (estopPaused)
            {
                wait(20, msec);
            }
            else
            {
                double pos = SortingArm.position(degrees);

                if (pos >= 10)
                {
                    sorting = false;
                }
```

```
                else
                {
                    SortingArm.spin(forward);
                }
            }
        }

        SortingArm.stop();
        return 1;
    }

    if (material == -1)
    {
        bool sorting = true;

        while (sorting == true)
        {
            checkPause(estopPaused, lastButtonState, cycleTimer, pausedOffset, pauseStart, false);

            if (estopPaused)
            {
                wait(20, msec);
            }
            else
            {
                double pos = SortingArm.position(degrees);

                if (pos <= -10)
                {
                    sorting = false;
                }
                else
                {
                    SortingArm.spin(reverse);
                }
            }
        }

        SortingArm.stop();
        return -1;
    }

    return 0;
}


// ======================================================
// RESET SORTING ARM TO 0°
// ======================================================
void resetSortingArm(int direction,
                     bool &estopPaused,
                     bool &lastButtonState,
                     timer &cycleTimer,
                     double &pausedOffset,
                     double &pauseStart)
{
    SortingArm.setVelocity(20, percent);
    SortingArm.setStopping(brake);

    if (direction == 1)
    {
        bool resetting = true;

        while (resetting == true)
        {
            checkPause(estopPaused, lastButtonState, cycleTimer, pausedOffset, pauseStart, false);

            if (estopPaused)
            {
                wait(20, msec);
            }
            else
            {
```

```
                double pos = SortingArm.position(degrees);

                if (pos <= 0)
                {
                    resetting = false;
                }
                else
                {
                    SortingArm.spin(reverse);
                }
            }
        }
    }
    else if (direction == -1)
    {
        bool resetting = true;

        while (resetting == true)
        {
            checkPause(estopPaused, lastButtonState, cycleTimer, pausedOffset, pauseStart, false);

            if (estopPaused)
            {
                wait(20, msec);
            }
            else
            {
                double pos = SortingArm.position(degrees);

                if (pos >= 0)
                {
                    resetting = false;
                }
                else
                {
                    SortingArm.spin(forward);
                }
            }
        }
    }

    SortingArm.stop();
}


// ======================================================
// MAIN PROGRAM
// ======================================================
int main()
{
    vexcodeInit();

    bool estopPaused = false;
    bool lastButtonState = false;

    timer idleTimer;
    timer cycleTimer;

    double pausedOffset = 0.0;
    double pauseStart = 0.0;

    configureAllSensors(estopPaused, lastButtonState, cycleTimer, pausedOffset, pauseStart);

    const double CYCLE_TIME = 12.0;
    const int BLACK_THRESHOLD = 15;
    const int PLASTIC_THRESHOLD = 18;
    const int METAL_THRESHOLD = 80;

    bool programRunning = true;

    while (programRunning == true)
    {
        checkPause(estopPaused, lastButtonState, cycleTimer, pausedOffset, pauseStart, false);
```

```
double idleVal = idleTimer.value();

if (idleVal >= 30)
{
    Brain.Screen.clearScreen();
    Brain.Screen.setCursor(1, 1);
    Brain.Screen.print("No caps detected.");
    Brain.Screen.setCursor(2, 1);
    Brain.Screen.print("Shutting down.");
    wait(5, seconds);

    programRunning = false;
}
else
{
    bool capIsHere = capLoaded();

    if (capIsHere == false)
    {
        Brain.Screen.clearScreen();
        Brain.Screen.setCursor(1, 1);
        double distNow = Distance2.objectDistance(mm);
        Brain.Screen.print("Distance: %.1f mm", distNow);
        uWait(20, estopPaused, lastButtonState, cycleTimer, pausedOffset, pauseStart, false);
    }
    else
    {
        idleTimer.reset();

        loadCap(estopPaused, lastButtonState, cycleTimer, pausedOffset, pauseStart);

        cycleTimer.reset();
        pausedOffset = 0.0;
        pauseStart = 0.0;

        Conveyor.setVelocity(20, percent);
        Conveyor.setStopping(brake);

        int maxBrightness = 0;
        bool capUnder = false;
        bool sortedAlready = false;
        int direction = 0;

        bool cycleRunning = true;

        while (cycleRunning == true)
        {
            checkPause(estopPaused, lastButtonState, cycleTimer, pausedOffset, pauseStart, true);

            if (estopPaused)
            {
                wait(20, msec);
            }
            else
            {
                double effectiveTime = cycleTimer.value() - pausedOffset;

                if (effectiveTime >= CYCLE_TIME)
                {
                    cycleRunning = false;
                }
                else
                {
                    int bNow = Optical6.brightness();

                    if (bNow > BLACK_THRESHOLD)
                    {
                        capUnder = true;

                        if (bNow > maxBrightness)
                        {
                            maxBrightness = bNow;
```

```
                        }
                    }

                    bool leaving = (capUnder == true &&
                                    bNow <= BLACK_THRESHOLD &&
                                    sortedAlready == false);

                    if (leaving == true)
                    {
                        int material = 0;

                        if (maxBrightness >= METAL_THRESHOLD)
                        {
                            material = -1;
                        }
                        else if (maxBrightness >= PLASTIC_THRESHOLD)
                        {
                            material = 1;
                        }

                        direction = sortCap(material,
                                            estopPaused,
                                            lastButtonState,
                                            cycleTimer,
                                            pausedOffset,
                                            pauseStart);

                        sortedAlready = true;
                    }

                    Conveyor.spin(forward);
                    uWait(20, estopPaused, lastButtonState, cycleTimer, pausedOffset, pauseStart, true);
                }
            }
        }

        Conveyor.stop();
        resetSortingArm(direction,
                        estopPaused,
                        lastButtonState,
                        cycleTimer,
                        pausedOffset,
                        pauseStart);

        Brain.Screen.clearScreen();
        Brain.Screen.setCursor(1, 1);
        Brain.Screen.print("Peak Bright: %d", maxBrightness);
        Brain.Screen.setCursor(2, 1);
        double dnow = Distance2.objectDistance(mm);
        Brain.Screen.print("Dist: %.1f mm", dnow);

        uWait(200, estopPaused, lastButtonState, cycleTimer, pausedOffset, pauseStart, false);
    }
  }
}

  Brain.programStop();
  return 0;
}
```

Appendix B: Project Schedule

Week / Dates	Start / Focus / Tasks	Meeting Plan
Sept 23 – 29	Team formation; idea brainstorming; finalize concept (recycling sorter).	Thursday, Sept 25 – Intro meeting (CMH)

Sept 30 – Oct 6	Draft and submit project idea.	Online coordination.
Oct 7 – 13	Sign out VEX kit; begin conveyor prototype; outline mechanical layout.	Thursday, Oct 9 – Design review (CMH)
Oct 14 – 20	Build feeder system; assemble main structure.	Thursday, Oct 16 – Build session (CMH)
Oct 21 – 27	Integrate distance and optical sensors; adjust mechanical design as needed.	Thursday, Oct 23 – Hardware session (SLC)
Oct 28 – Nov 3	Prepare for formal presentation; finalize mechanical design; order remaining parts.	Slide work completed online.
Nov 4 – 10	Begin coding; implement subsystem functions; initial integration.	Thursday, Nov 6 – Coding session (CMH)
Nov 11 – 17	Software design review; refine logic; verify sensor thresholds.	Monday, Nov 10 – Debug session (online)
Nov 18 – 24	Full-system test run; tuning; prepare demo; finalize integration.	Thursday, Nov 20 – Final build/test session (CMH)
Nov 25 – Dec 1	Return parts; record final video; complete and polish final report.	Thursday, Nov 27 – Wrap-up meeting (CMH)
Dec 2	Submit final report.	Online submission and confirmation.

Table 11.1 - Timeline of Project