MTE 1A Course Project Final Report

Prepared By Bot Group 8-9

Members:

Alex Tan 21119519

Dylan Zhang 21152510

Joshua Hodson 21118118

Kevin Li 21122937

A Report Prepared For:

MTE 100 and MTE 121

Date of Submission:

December 3, 2024

Table of Contents

Table of Figures

List of Tables

Acknowledgements

The team would like to acknowledge both the MTE 100 and MTE 121 teaching teams; without their support and guidance, this project would never have come to fruition. In particular, the team would like to thank Nathan Menezes for his feedback, advice, and overall positive and uplifting attitude. We would also like to thank the teaching team for allowing us a slight extension of up to ~40 pages of content.

Summary

For their course project, the team developed a card-shuffling and card-dealing robot. When deciding on a focus for the robot, the team reviewed various resources and found inspiration in dealers and casinos; they identified that while dealers are essential in casinos and card games worldwide, they have notable drawbacks, such as the potential for cheating, errors, and unpredictability. To address these issues, the team designed a robot capable of shuffling and dealing cards, effectively replacing human dealers in casinos.

The team aimed for the robot to support a variety of card games, all customizable by the user. The robot was designed to shuffle cards independently and deal with them according to the selected settings; to enhance reliability, it was built to handle cards of different quality and colour, incorporating anti-jamming procedures to minimize disruptions.

Initially, the robot allowed users to remove cards from the center after shuffling for re-shuffling. However, after redesigning, the team decided to simplify the design by removing this feature; instead, they randomized the order in which the robot dealt cards to achieve a similar level of randomness. The robot employed sensors to detect card placement and jams; these motors were used for shuffling, dealing, and rotating the robot, with motor encoders aiding the shuffling and dealing processes. Additionally, a gyro sensor was utilized to manage the robot's rotation.

The robot’s software followed a structured sequence. It began with a “startup” phase, where it prompted users to select the game type and options, then waited for cards to be placed inside. Once the cards were in position, the robot initiated shuffling and dealing procedures, using specific anti-jamming methods to detect and resolve any potential issues. After dealing the cards, it entered the end-of-round procedure; where if additional rounds were required, the robot waited for the next set of cards. Otherwise, it proceeded with the shutdown process.

To ensure the robot met the intended criteria, the team conducted a series of tests; each criterion was evaluated based on the robot's performance and the degree of success achieved. These results allowed the team to determine whether the robot met the project requirements and identify areas for future improvement, given additional time.

1.0 Introduction

Historically, casinos worldwide have relied upon human card dealers [1]. Their deft motions and charisma create a lively energy that nothing else can match, drawing people to their tables. The dealer is a fundamental building block of a successful casino, and without a good dealer, there is no pull to the card table, nothing to entice a person to sit down and play.

However, human dealers inherently bring to the table the human factor. They introduce unpredictability to the operation. Dealers can cheat, make errors, or manipulate the game to shift it in another player's favour [2]. Dealers have hundreds of hours of experience in card handling, so they are very good at manipulating cards. In 2023, two Las Vegas baccarat dealers conspired with two players to help them win thousands of dollars by revealing the gaming cards to the players without anyone else noticing [3]. In 2017, a baccarat dealer in Maryland made a prior agreement with a player to leave a specific portion of cards unshuffled [4]. This way, the player would know the order of the cards for that section and thus know whether to bet on the player's or banker's hand [4]. The scheme resulted in the player earning more than $850,000 in winnings, which was to be split with the dealer as per their agreement [4].

As shown, human dealers have cheated and participated in schemes with players before. Human dealers present risk to not just the player but also the casinos themselves, as their reputation could be significantly lowered if one of their dealers were found to be cheating. This team's goal was to develop an automated robot dealer to solve this problem, used in place of a human one. An automated robot card dealer would eliminate all possibility of cheating or foul play by the dealer. A robot dealer would reduce frustration as it prevents players from wildly accusing the dealer of favouritism, foul play, or unfair treatment, as it is incapable of doing so. The robot is programmed to do only its task, nothing else. The presence of the robot would create an environment where players know that they will get a fair hand and without the fear of the dealer conspiring with others. It would draw players in with its security and guarantee of fair play, improving the casino's reputation and increasing player satisfaction.

2.0 Scope

2.1 Main Functionality

The purpose of this robot is to take a deck of cards, shuffle them, and take user inputs to deal a specified number of cards to a specified number of players in a random fashion. The target users of this robot would be casinos or card game enthusiasts.

2.1.1 Detailed Task List

1. The robot begins with an interactive start menu where the user can input the game they would like to play, either Poker or a Custom Game, then specific settings for their chosen game. There will be specific screens where the user can change the following settings:
    a. The number of players
    b. (Custom Game only) The number of cards per player
    c. (Custom Game only) The number of cards to the dealer. This can be set to 0 if the custom game does not involve a dealer.
    d. The number of rounds to be played
2. After the user inputs all settings, the robot automatically initiates the first round of play, and the round number is displayed on the screen. At the beginning of every round, the robot waits for the two card slots to be occupied with a static number of cards (sensors wait until a stable reading remains to ensure cards are placed correctly and the user's hands are not interfering).
3. The robot will shuffle the deck by ejecting cards from each side alternatively at random timings (the left side will run for a small period of time, then the right side). This will continue until both sensors read the empty value, and both motors will run simultaneously to ensure no cards are left in the slots.
4. According to the user's inputs from the start menu, the robot will deal the specified number of cards to each player and dealer (if applicable) by rotating directly to a player and ejecting a single card at a time.
5. Once the robot has dealt all the necessary cards, it will turn behind the dealing area to eject all remaining cards from the robot, which signifies the end of the round. The screen will update to show the next round, and the robot will once again wait for cards to be placed back in the two shuffling slots.
6. Steps 2-5 will repeat for every round of play.
7. After step 5 of the final round, the robot will prompt the user if they would like to play again. If the user selects yes, the robot will restart from round one. If the user selects no, the robot will display a message, disable all motors and sensors then shut off.

2.2 Measures and Detection

The robot system uses inputs from three sensor types (ultrasonic/infrared, gyro and colour), the motor encoders, and the integrated buttons.

2.2.1 Sensors

The robot design uses ultrasonic/infrared sensors, one for each shuffling slot, to detect whether or not there are cards placed and waiting to be shuffled. These sensors record the empty slot value on startup

and constantly measure the value until cards are placed and the value stays static for a few seconds (this allows for readjusting the cards, waiting until the user's hands are no longer interfering to ensure that the cards are placed properly before starting).

The colour sensor is used in conjunction with the dealing mechanism and simply reads a raw light reflection value to check if a card is being dealt. The raw value allows any type of card (different colour, clear cards, etc.) to work with the robot without any calibration.

The gyro sensor is used to help rotate the entire robot. It is used for absolute turns (turning to a specific degree) or relative turns (turning a specific ammount from current value).

2.2.2 Motor Encoders

The motor encoders are used in the randomness portion of the shuffling process, where motors are run for a random, small period of time to eject cards into the middle. They are also used in the dealing process to deal one card at a time, as this process requires precise timings.

2.2.3 Integrated Buttons

The integrated EV3 buttons are for the user to interact with the start menu. The screen display will tell the user which buttons to press to change specific settings or for selecting options. These buttons allow for customizability through the start menu.

2.3 Reacting to the Environment

The robot has software-based solutions in place to try to prevent the most likely unexpected behavior: the jamming of the cards. Should the cards jam in the shuffling slots or in the dealing slot, the robot is able to recognize this and try to solve the problem. If the jamming is extreme, the robot will deactivate all motors and sensors, allowing the user to intervene and restart.

2.4 Task Completions

This section will overview when the robot detects that a task is completed and how it will proceed to the next step.

2.4.1 Start Menu

The start menu process is complete when all the settings are set (number of players, number of rounds, number of cards per player, etc.). Settings are changed through button inputs from the user and will display the current selections on the screen where the user can cycle through.

2.4.2 Shuffling Completion

The shuffling process is considered complete when both shuffling slots are detected as empty by the ultrasonic sensor. The robot will register the slots as empty when their ultrasonic readings are the same as when the robot started up (value taken when the robot started up with empty slots). To ensure all cards are in the dealing slot, the motors for each slot run for a few additional seconds.

2.4.3 Dealing Completion

While the robot deals, it keeps track of how many cards it has dealt to each player in an array. It will rotate to each player's pile and deal one card at a time, repeating this process until each value of the player cards array is equal to the specified number of cards. When all players have been dealt the specified number of cards, it will turn around to eject the remaining cards. This process signifies the end of the dealing.

2.4.4 Shutdown Process

Once all rounds are complete, the robot will flush out all cards from the robot, rotate back to the starting rotation, and then prompt the user if they would like to play again. If the user accepts, the robot will not shut down and will instead restart the entire process with the same settings. If the user declines, the robot will disable all sensors and motors, display a goodbye message then shut off the robot.

2.5 Scope Changes

1. An early concept allowed the user to shuffle multiple times, but the current design restricts the system to only one round of shuffling, as the cards cannot be easily removed from the dealing slot. It was decided not to implement multiple shuffling as it was better to maintain a simple and compact design. Instead, the shuffling procedure includes some additional random components.
2. The original design also allowed 360 degrees of free rotation; however, due to the design of the rotation mechanism and wiring, the current design is limited to 270 degrees of rotation. This limits the dealing area to about 180 degrees. This 180-degree dealing area was maintained and used in the final design as the intended use is in casinos, where most tables are only 180 degrees around a dealer rather than a round table.
3. Many more games were planned; however, it was decided to focus on just poker, which is considerably different from traditional card games, as well as a custom game mode where all settings for different games can be set by the user. This change was made due to time constraints, and for convenience, as even with other dedicated game modes, there are still many games out there, so a custom game mode would always be required
4. An early design concept included only one motor in charge of the shuffling, which included the use of either gears or a parallel crankshaft. Both of these mechanisms would result in the wheels from both sides spinning in opposite directions, which would send the cards from 2 piles into a middle pile. The issue with these designs was that it was difficult to avoid jamming as it was very likely that cards from each side would enter the middle at the same time as the motors were running simultaneously. The current design has more control over the cards and allows for random shuffling patterns.

5. Another concept concerning the shuffling was to begin with only one shuffling slot, where the user would place the entire deck. The cards would then be shuffled by ejecting to either side of the deck and later combined again by sliding down a channel. This system would be more convenient for the user, however it could lead to many opportunities for cards to jam and could be complicated to combine the cards again once they were separated. The use of a funnel for cards would likely lead to a large system which did not comply with the robot's simple and compact design philosophy.

3.0 Constraints and Criteria

This section will describe the constraints and criteria that guided the design of the robot. Constraints define the limitations/ boundaries under which the robot will operate, and criteria represent task and performance standards the robot must meet to fulfill its purpose.

3.1 Robot Constraints

1. The design allows for a maximum of 8 players. This is to limit memory use on the robot itself and a lack of general space to deal cards as the robot deals in a 180-degree area [5].
2. In the start menu, no unrealistic or impossible values can be entered, such as negative players or numbers of cards.
3. A maximum of 1.5 decks of cards can be used at once, and this is due to the size of the shuffling slots. However, any amount of cards under this amount can be used without any additional setup or calibration.
4. Due to the wiring of the rotational motor, the robot is restricted to 270 degrees of rotation, where 180 degrees is the playing area, and 90 degrees to eject all remaining cards away from the playing area.
5. The tension mechanisms are suited for regular playing cards. This could result in issues with smaller or thinner cards, such as more than one card being dealt at a time.
6. The shuffling speed of the robot cannot perform as fast as possible in efforts to lower the amount of jamming.
7. Dealing distance of cards is limited as a result of the maximum power of the motors. This also means there may not be much difference between a pile for the dealer and for the player.
8. The speed of the dealing is limited as the robot needs to turn to each player for every card it deals and needs to remain precise to ensure distinct piles of cards are formed.

3.1.1 Valuable Constraints

Many of the robot's constraints are due to physical limits of the robots such as only being able to shuffle once per round, the total number of cards it can handle and the rotation of the robot. Since the number of players was limited to 8, the robot was made to work with the restricted 180 degrees of dealing area. Limiting shuffling and dealing speeds allowed the prioritization of the randomness of the shuffling and dealing, which adheres to the main goal of the robot: to provide users with a truly random dealer.

3.2 Criteria

The overall function of the robot is to be able to shuffle any reasonable number of cards, and deal the user specified number of cards to the user specified number of players. The robot should be able to accomplish this task with different types and colours of cards, different settings and in different environments.

3.2.1 Startup

1. Displays a welcome message and prompts the user to press ENTER to proceed.
2. Displays the currently selected game mode (poker or custom game). The screen cycles game modes through the UP button and the selection is confirmed on ENTER.
3. Prompts the user to enter settings for number of rounds, number of players, number of cards per player and/or number of cards for the dealer. Selections confirmed on ENTER.
4. Proceeds to the shuffling once all selections are made, and displays Round 1.

To avoid unexpected events, the robot will not allow for round selections less than 1, below 2 or above 8 players, less than 1 card per player and negative cards for the dealer.

3.2.2 General Operation

1. The robot waits until cards are placed in the shuffling slots. The robot should wait until the sensor values are stable so it does not start while the user's hands are still interfering or adjusting the cards.
2. The robot should begin the shuffling process, where each side will alternate at a random timing and dispense the cards into the dealing slot. The robot additionally shakes during this process to decrease jamming and ensure cards are properly in place.
3. The robot should then deal cards to each player, one card at a time. The robot must successfully deal exactly the specified number of cards to each player and the dealer, in a random order.
4. After dealing, the robot turns behind the playing area and ejects all the remaining cards in the robot and ends the round.
5. The screen updates to the next round and awaits for the cards to be placed in the shuffling slots again. All these steps should be completed during every round of play.

3.2.3 Unexpected Handling

The robot should be able to detect jamming and begin an error procedure. This will also occur when the robot detects that a card has not been dealt properly (if not enough cards were placed in the robot). If the robot is tampered with while turning, it should still turn to the correct spot. Finally, the robot should operate normally with extra cards and with different types, colour and quality of cards.

3.2.4 Error Procedure

1. The robot will display an error message to notify the user that it has detected an error (card jamming or card not dealt properly).
2. Robot stops all motors and rotates back to its starting position.
3. All cards will be ejected from the robot.
4. The robot will wait for the ENTER button to be pressed. During this time, the user can freely intervene as the robot is not operating. Any jammed cards can be safely removed from the robot before proceeding.
5. Once ENTER is pressed, the robot will restart the round, waiting for the cards to be placed in the slots.

3.2.5 Shutdown

1. Once all rounds are completed, the robot should automatically start the shut down procedure and display a message on the screen.
2. The robot will ensure all cards are ejected by activating the shuffling motors and then the dealing motor.
3. All motors and sensors will be deactivated once the robot has rotated back to the starting position.
4. The screen will prompt the user for a replay.
5. Should the user choose YES, the robot will reactivate and begin from Round 1 with all the same settings.
6. Should the user choose NO, the robot will end the program and shut off.

3.3 Changes in Criteria

Most of the criteria are met by the final design of the robot. However, some small details have changed throughout the development of this project. The biggest was the randomness of the shuffling, as the original design had the user shuffle multiple times until they were content. With only one shuffle cycle per round in the final design, the criteria were changed so that the shuffling pattern had to have a considerable degree of randomness and should not repeat the same pattern multiple times in a row. There were also criteria added for the start menu and the unexpected handling procedures, as these were not thought of until testing.

3.4 Valuable Criteria

The general criteria of the robot being to shuffle and deal cards for multiple rounds were the most valuable criteria which guided the project. While many small details changed throughout the development, the general criteria of getting the user's settings and using them to properly shuffle and deal the cards were the tasks the robot absolutely must complete. Some less valuable criteria would be the shaking of the robot while shuffling, but due to the unpredictability of the movement of playing cards, this still needed to be included in the design to limit the occurrences of unexpected events such as jamming. The criteria for the robot to shuffle and deal randomly are important to maintain the purpose of the robot and solve the issue of rigged card games, and were the most valuable criteria in testing.

4.0 Mechanical Design and Implementation

The overall design of the robot is a simple system while being as compact as possible. Using only LEGO and EV3 components, three main mechanisms were created and combined to create the robot. These three mechanisms are the shuffling, dealing and rotating mechanisms. The shuffling and dealing make up the body of the robot, where they are stacked on top of each other and work in cohesion. The frame of the robot has 2 functions, to hold the sensors and motors on the outside, and to serve as a "channel" for the cards on the inside. The entire body is placed on top of a rotational mechanism, which allows it to rotate from a single motor. Simply put, the shuffler takes cards from the sides and funnels them into the center of the bot, and the dealer shoots it out, all while being on a rotating turret.

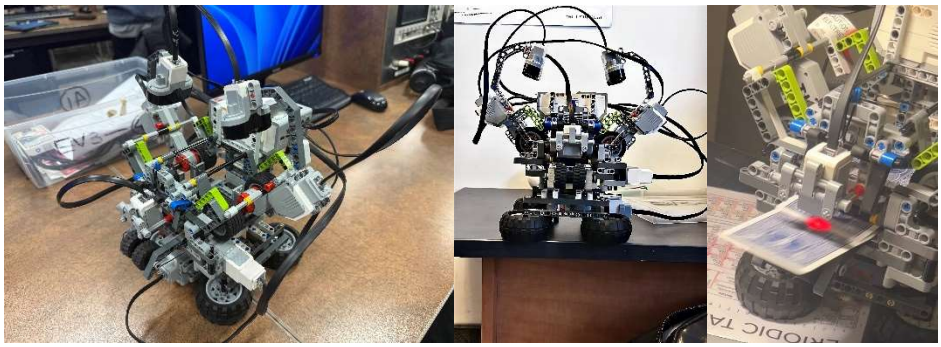


Figure 1: Overall Design of the Robot

4.1 Design Process

The engineering design process was a crucial part of the construction of the robot, allowing a better understanding of the faults in the design and to better implement changes and additions to the robot to meet our criteria. The first step taken was to define the problem the robot was tasked with: shuffling and dealing cards. Criteria had been set for the robot and brainstorming occurred for ways that the robot would be able to meet those criteria.

Following a discussion and considering the requirements/criteria the robot had to meet, a design philosophy: **simple and compact,** was decided. The main objectives were to create a small and portable robot, capable of being extremely easy to set up and use. Keeping in mind the design philosophy, different tasks the robot had to perform were thoroughly examined, along with how different mechanisms would allow the robot to reliably perform each mechanical task without overcomplication. The task list was as follows: shuffle, rotate, deal.

4.2 Individual Components and Mechanisms

4.2.1 Shuffler Design

Many different mechanisms for the shuffler task were considered. Some of these joined two decks which were split by the user and others split and combined an entire deck itself. These two categories had their own pros and cons which can be seen in the following table.

Table 1: Pros and cons of the shuffler designs

Design	Pros	Cons
Single Pile Splitter	- Less Components - Allowed user to insert one pile - Unique	- **Inconsistent and Unreliable** - Complicated funnel system - **Tall and Bulky**
Double Pile Combiner	More Control over Cards Proven Design - **Simpler**	- Wider - More Components - Requires piles to be split before being inserted

Here is one example of a design concept which used a single motor to split up an entire deck into two piles and combined them back together.

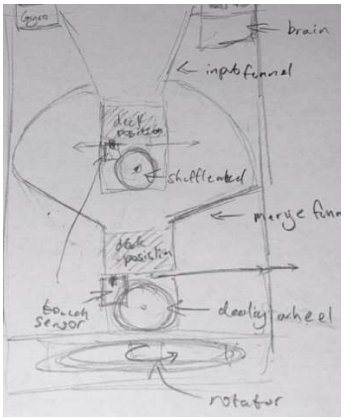


Figure 2: An initial sketch of a single deck insertion shuffler

While this proposed design would allow users to insert a full deck without splitting it, there were many inherent flaws. This design would need to be much taller to incorporate the merge funnel, and to

give space for the cards to fall down the two sides. The proposed bot would be just as wide as the original proposed design as there were card chutes both sides. This design is also significantly less reliable, as there is no way to control the orientation of the cards as they fall down the chute and as they merge into one pile. It was expected that this design would result in a lot of jamming and cards being flipped over. It was decided to simply combine two already split decks into one for the shuffle process.

Featured below is an original design which featured a double crank linkage, which still only used one motor to rotate 2 wheels in opposite directions. This design would have been favorable to save resources and require the use of one less motor.

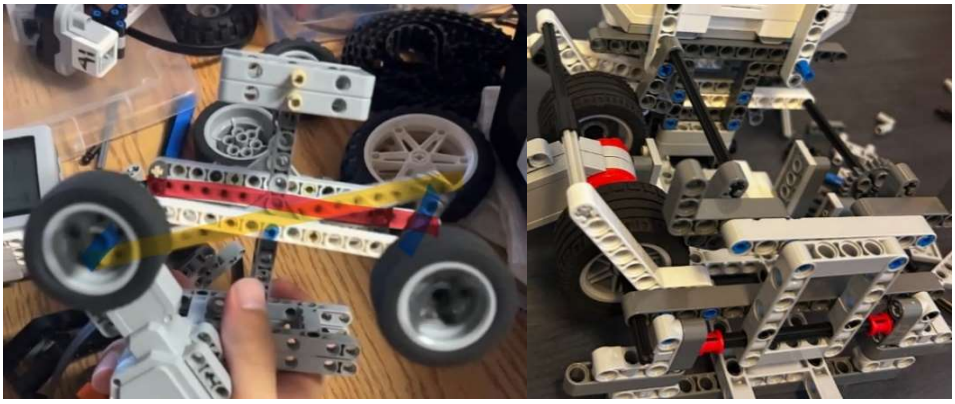


Figure 3: Double Crank Parallel Linkage

The initial testing of this mechanism was promising; the crank was behaving as expected and the max motor rpm was similar to that of a motor under no load. However, issues arose after implementing the other end of the shaft support (from a cantilevered shaft to a fully supported shaft). This addition introduced substantial friction and highlighted initial flaws with the design. A fatal flaw of this design is its incredible need for precision. If the shaft cranks were not parallel, the crank system would go over center and lock itself.

This eventually led to many problems and the design was replaced with the final mechanism, featuring the use of two motors (see Figure 3). This design connects a motor directly to each shaft. Additionally, since wheels were significantly larger than the motor, the motor could be placed in the center of the shaft with concern that it would interfere with the shuffling.

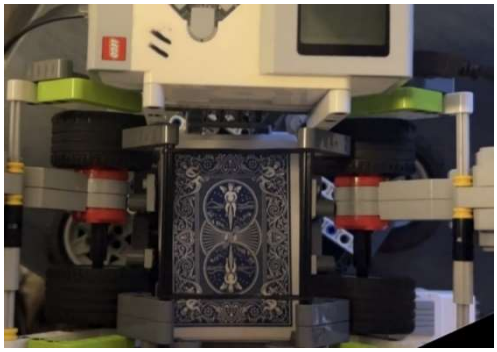


Figure 4:The final shuffler design

Although there was a design for the roller setup, the shuffler still lacked a few sub-components. The first subcomponent was the card holder, formed from L-Pieces on both ends of the cards, which allowed for a perpendicular and parallel component to the card (see Figure 4). This idea was supported by the team's supervising MTE 100 TA, who, in his feedback on the Formal Presentation of the robot, agreed that this was a good solution to the problem [5].

The perpendicular bar ensures the alignment of the cards when they are placed, as the flat surface acts as an "wall" that helps the cards align, while the parallel part of the L-piece, goes the length of the cards and ensures that there is no rotation as they are placed in the holder (see Figure 4).

Figure 5: The cards in the holder

Another subcomponent of the shuffler was the tensioner. The tensioner acted as a "dam" to limit the flow of the cards. Since the cards were angled downwards, the tensioner blocked all but the bottom most card (contacting the wheel) from falling out of the holder. Additionally, the blocker also applied pressure on to the bottom card, ensuring there was enough friction such that only one card is shuffled at a time. The tension design used an L-piece angled against the wheel such that a small triangular funnel

allowed one card out at once, while the length of the L-piece holds the rest of the cards in place (see Figure 5). The card tensioner used elastics to ensure a consistent amount of force being applied.

The final subcomponent of the shuffler was the funnel for the cards, colloquially known as the deck holder. Through testing, the clearance between the two wheels as was made small as possible to limit card movement as it fell through the tunnel. This allowed for a simple deck holder (see Figure 5).

Figure 6: The cards in the holder

Overall, the shuffler was extremely consistent, due in part to the many tight yet functional clearances on both card holders as well as the super tight clearance on the tensioner. Although simple, the shuffler was built with millimeter clearances in mind, which made it extremely consistent and robust.

4.2.2 Dealer Design

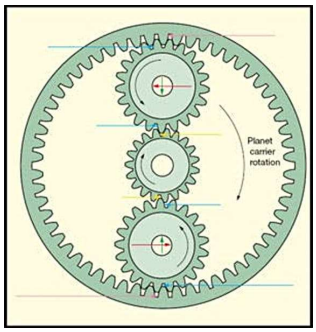
Given that the shuffler was able to place the deck into a container, the next course of action was to implement a card dealer. Intuitively, there were two options: deal from the top, or deal from the bottom, both with its pros and cons. For example, a top dealer using a roller would remove the need of a tensioner, since the cards would be squeezed between a top roller and the bottom plate holding the cards. However, with a top roller design, the roller would need to be able to retract back and forth, to allow the shuffler to drop cards into the deck container, and for the roller to deal the cards.

Table 2: Pros and Cons of Dealer Designs

Design	Pros	Cons
Top Roller Dealer	No separate tensioner needed More human like	Needs to be able to retract in and out for shuffling operations Likely requires a motor to retract in and out
Bottom Roller Dealer	Static roller position More compact	Requires separate tensioner

Based on the Table above, the Top Dealer is not a good option for the robot, with the major reason being its complexity. The possible designs for an automatically retracting top dealer are as follows:

Table 3: Analysis of potential Top Dealer Designs

Two Motor Design	One Motor Design
General Design: - Uses a motor to retract the roller - Uses a motor to power the roller  *Figure 7: A retractable design seen in excavators [6].* Concerns: - Uses two motors	General Design: - Uses planetary gearing and multiple gears to rotate the arm as the roller is spinning  *Figure 8: Planetary Epicyclic Gear Setup [7]* Concerns: - **Extremely complicated** - Requires space in the core of the robot

Examining the potential designs for a Top Dealing mechanism, it was clear that neither are favorable, which led to the selection of the bottom rolling dealer for the robot. This design was comparable to that of a conveyor belt, the cards would be placed on wheels which would spin to eject a card. The other cards on top of the bottom card were held in place with a gate similar to the tensioner design used for the shuffler (see Section 4.2.1), the gate would allow for only one card to be dealt at a time, and the tensioning on it would allow for consistent friction between the card and the roller ensuring it was ejected the correct distance.

The bottom rollers used 24 x 14 Shallow Tread Wheels. These wheels were the smallest available that fit in the gap between the base plate of the deck holder. The large width yet small radius of the wheel let allowed it to fit tightly underneath the plate to be as compact as possible (see Figure 8).

Different wheel sizes were additionally tested previously and had different issues. For example, when it tested with a narrower wheel that protruded more, it was found that the cards would get jammed on the side of the wheel. This resulted in the decision to use a short but wide wheel.

The dealing system used sets of wheels, one at the back of the holder (back wheel), one at the front (middle wheel) and one outside of the holder (front wheel). Even though the middle wheel only protruded up by 3mm, it was enough to angle the card in a way where it is shot angled up instead of angled straight. To counteract this, a wheel was placed in the back, so that the card is parallel to the base

plate when it exited. This simplified the tensioner and ensured that cards do not slide backwards while on top of one another.

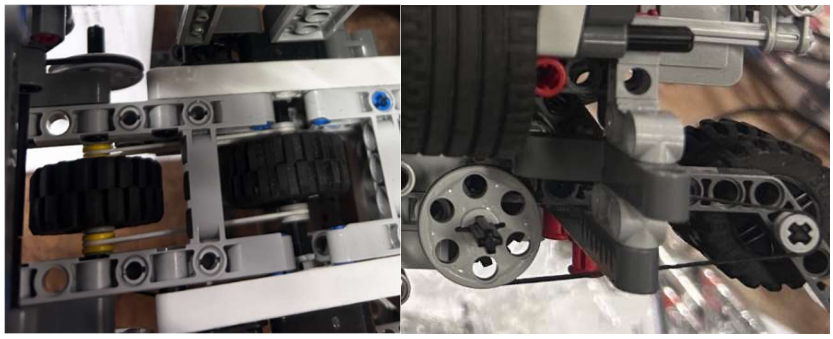


Figure 9: Comprehensive view of the bottom rollers and their connections

The three wheels were connected using a belt drive which is generally lightweight and compact, however, it does not handle well under resistance. As a result, there was often a lag between the wheels when the resistance of the cards overcame the friction on the belt. This was an issue especially with the front wheel which was belted up to increase rpm; since this was the wheel which "shot" the cards out the correct distance, any drop in rpm significantly decreases the accuracy of the deal, for example, if the next card above is pressing down against the back wheel, the front wheel will lose some of its speed and not be able to deal the card the full required distance.

The motor powering the roller mechanism is directly connected to the middle wheel's shaft. This is important, because had the motor been connected to a belt drive with no wheel on it, any slippage might cause the entire dealing sequence to stall out. Additionally, connection directly to the middle wheel as opposed to the back or front, compacted the robot which made it more aesthetically pleasing.

The tensioner was the final component of the dealer. Like the tensioner for the shuffler, the tensioner acted as a gate to limit only one card at a time to pass through while also ensuring the card that goes through is sufficiently compressed against the bottom rollers for consistency. Testing without a tensioner showed that cards, especially near the end of the stack, would shoot out together (e.g two or three at a time). However, due to the limited space in the bot, the tensioner design had to be unique and planned out. A simple triple jointed mechanism (see Figure 9) that looped around the structure of the robot frame was settled upon. This design aligns with the concept of simplicity and compactness. At the end of the tensioner, free spinning wheels were used which decreased friction from the tensioner while ensuring compression against the bottom spinning wheels. Additionally, this design allowed for the elastic used to tension the gate to be located further from the center of the bot for ease of access.

One issue faced with the tensioner was that the elastic had to be constantly swapped for new ones, and occasionally multiple cards or no cards at all would shoot out of the robot. It was believed that this is

attributed to the fact that since the multiple joints are used, the tension of the elastic doesn't proportionally translate to the compression of the roller onto the cards.

Finally, one last area for improvement for the dealer in general was the dealing speed; there is a clear drop-off in speed as cards are being dealt. This is due to there being more cards on top, so the bottom card is under more compression against the bottom roller (due to the mass of the other cards), which increased friction and thus made it shoot out faster. With less cards in the pile, the cards may slip against the bottom roller resulting in lag or a slight pause between the rollers starting up and the card leaving the chamber and being dealt.

4.2.3 Rotator Design

The final major component of the robot was the rotator. The goal of the rotator was to allow the robot to deal to all players in a circle. When different designs were examined for the rotating mechanism, two main options were considered. The first of the two was to use wheels to spin the robot, the other was to use a turret to rotate the robot on the spot. Below is a table summarizing the design considerations for the rotating mechanism.

Table 4: Summary of potential Rotating Mechanisms

Wheel Spinner	Turret Rotator
General Design Wheels across from another spinning in opposite directions to rotate  *Figure 10: The wheel setup would be similar to that of the robot in this image [8]*	General Design Uses a central rotating bar to spin the top component about the bottom base  *Figure 11: An example of a robot mounted on a turret [9]*

Table 5: Pros and Cons of the Rotator Designs

Design	Pros	Cons
Wheels	Can easily spin infinitely on the spot	No stable base Can drift away from original position

Turret	Stable Base Aesthetically Pleasing	Difficulty wiring May lack the torque necessary

Considering both the pros and cons and the necessary applications of the robot, the turret was evidently the best fit for the design. Over the course of the design, three different options for the turret mechanism were considered, all of which are very similar to one another.

Table 6: Summary of different turret designs

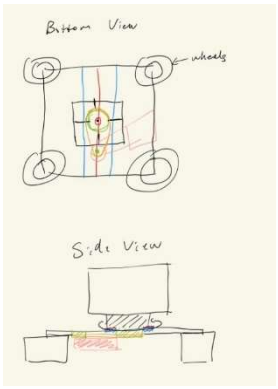
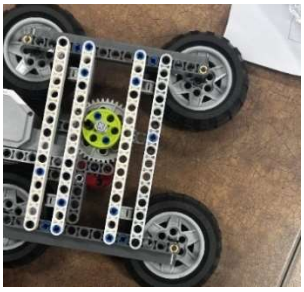
Design 1	Design 2	Design 3
Direct Power Transfer	Belt Drive Power Transfer	Geared Power Transfer
The robot is attached directly to the motor, which is mounted on the base  *Figure 12: Direct power base*	The rotator motor is shifted away from the center and a belt drive is installed to it from the original position  *Figure 13: Concept sketch of the belt drive rotator*	The rotating element of the robot is connected to a gear which is geared down to increase torque.  *Figure 14: Geared power transfer base*

Table 7: Tested pros and cons of different turret designs

Design Iteration	Pros	Cons
Direct Power Transfer	Simple	Undershot/Overshot Turns Lacked the Torque to implement a consistent PID Needed to be underpowered to reach a reasonable speed
Belt Drive		Belt did not have enough friction to turn the robot

Geared Power Transfer	More torque Manageable gear ratio	More Complex

The direct Power Transfer was first implemented for the robot, however due to the cons stated in Table 7, the robot had malfunctioned repetitively during testing because of undershooting and overshooting turns. This resulted in failure to stay within the rotation tolerance which led to the gyro reading being incorrect. These issues were resolved after the design was switched to the Gear Power Transfer, featured in the final version of the robot.

4.2.4 Sensor Mounts

The robot used three types of sensors: Ultrasonic sensors/Infrared sensors to detect cards in the shuffling slots, the colour sensor to detect cards being dealt and the gyro sensor to detect rotation of the robot.

The ultrasonic/infrared sensors were mounted onto the motors of the shuffling mechanism, allowing them to hang above each respective shuffling slot and point down. It was aimed to have the sensor perpendicular to the cards while they were placed in the slots. If set up perpendicular to the cards, the sensor would provide a more accurate reading as the waves will not deflect in other directions and will simply bounce back to the sensor. When the robot starts up, the sensor will read the empty value of the shuffling slots and repeatedly compare the current value to that initial value, which allowed it to recognize when cards were placed in the slots.

Figure 15: Ultrasonic sensor hanging above shuffling slot, pointing down

The gyro sensor was simply mounted directly to the brain as there is no need for it being exposed. Since it is mounted to the main body of the robot (it will rotate with the robot), it would have an accurate reading of the current rotation of the robot.

The colour sensor was mounted directly above the front wheel of the dealing mechanism to read the raw light value, detecting whether there is a card being dealt. This sensor was mounted as close as possible to the cards to ensure an accurate reading at all times. Should the sensor be placed further, it

could have resulted in other movements of the robot being mistakenly read as a card passing through as the sensor was only reading raw values.

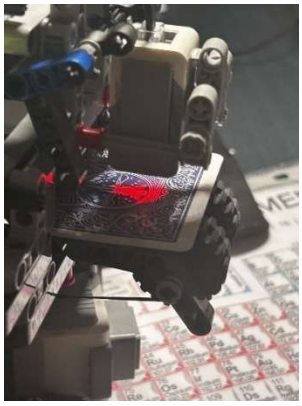


Figure 16: Color sensor, above front wheel of dealing mechanism

4.3 Overall assembly

As previously stated, the design of the robot prioritized being simple and compact. Each individual mechanism works cohesively with the others and as such the overall assembly is quite simple.

4.3.1 Integration of Mechanisms

The main challenge of integrating the mechanisms was ensuring they do not interfere; however, the robot was designed to have each mechanism set up the next one.

Shuffler Integration: Positioned at the top of the robot, takes the cards and feeds them into a single pile, deck holder, which is the starting position of the dealing mechanism. Shuffling slots were placed above and to the side of the dealing system to avoid interference. Since the motors and wheels for shuffling had to be placed below the slots, it was made sure they would be out of the way of the dealing slot. This also allowed for the cards to fall into the slot rather than just sliding in which would decrease chances of jamming and allow for more cards to be in the robot at a time.

Dealing Integration: The entire dealing mechanism sits right below the shuffling mechanism, which allows these 2 systems to work seamlessly. This way, once the cards are done shuffling, they are ready to be dealt without any other inputs or processes. As previously mentioned, the motors and wheels from the shuffler are placed on the sides of the dealing slot, so the two mechanisms can be combined and remain compact. The bottom of the dealing slot is where the wheels and rollers for dealing are placed, which allows for dealing right when the slot is occupied with cards. The motor is placed to the side of the dealing slot, and just under the wheels of the shuffling system.

Rotator Integration: Since the shuffler and dealing mechanisms are essentially the body of the robot, they are placed and attached on top of the entire rotating mechanism. The rotating mechanism thus stays static and rotates the body of the robot, allowing for the other shuffler and dealer to rotate together. Two pegs at the bottom of the frame of the body attach to the gear portion on the rotator, which is placed

around the center of gravity of the body. This allows the entire body to be rotated from one point which is the rotational motor. The motor itself is placed under the supporting frame and held above the ground due to the wheels.

4.4 Mechanical Design Changes

Many changes were made to the robot throughout the development process of the robot. These changes were implemented to address design challenges, improve functionality, and enhance the overall performance of the robot.

4.4.1 Multiple Card Shuffling Cycles

An original design allowed the user to take out the deck of cards and replace them to shuffle as many times as they would like prior to beginning the dealing process. However due to the tensioning mechanism for dealing, once the cards have been shuffled once, it is rather difficult for the cards to be removed from the slot. As a result, the final design only allows for 1 shuffle cycle per round, but the software increased the randomness of the shuffling. The team decided this would overall benefit the design of the robot as it allowed it to be more compact and the inclusion of the tension system was necessary for accurate single card dealing.

4.4.2 Robot Rotation Range

The original plan included 360 degrees rotation, which would allow for the robot to be placed in the middle of a table and deal to players sitting around it. The current design features 270 degrees rotation due to the wiring and the design of the turret rotator. Since the motor was attached to the rotator itself, the wire would not rotate along with the robot. As a result, if the robot would rotate more than 270, the wire would get caught around the robot to prevent further rotation. A solution to this issue would be to mount the motor onto the robot itself rather than the rotator. Instead, the current design was kept and the software was changed to 2025-03-03accommodate for 180 degrees of dealing area, and the last 90 degrees to eject the cards away from the dealing area.

4.4.3 Previously Mentioned Changes

Refer to Sections 4.2.1 for changes to the Shuffling Mechanism and 4.2.3 for changes to the Rotator Mechanism.

5.0 Software Design and Implementation

With the goal of flexibility and consistency in mind for the program, the program utilized in-depth error handling processes, parallel/background processing threads, PID feedback loops, and sensor data adjustments. Additionally, in-depth pointer and memory management were used to improve the program's overall memory usage and to improve the program efficiency.

5.1 Program Breakdown

The program can be broken down into multiple phases: configuration, user interaction and setup, control functions, game mechanics, error handling and utility, and program shutdown phase.

5.1.1 Configuration

The configuration functions set up the robot's hardware by configuring motor behaviors and sensor modes. This foundational setup ensures that all subsequent operations have the necessary hardware parameters correctly established. Additionally, it defined motor ports, sensor ports as well as key global constants such as the CPU tick speed, default sensor values, and the sensor error scale values.

5.1.2 User Interaction and Setup

User interaction and setup are handled primarily through the startMenu function, which served as the gateway for configuring game options such as the number of players, rounds, and cards. This function leveraged helper functions such as getRoundCount, getPlayerCount, getCardCount, and getDealerCount to gather specific game settings from the user, ensuring that the game is tailored to user preferences before it begins. Once the settings are obtained, the getAngleArr function calculates the angles corresponding to each player's position to facilitating accurate card distribution.

5.1.3 Game Mechanics and Control

The core game mechanics are orchestrated by the playRound function, which manages the sequence of actions required for each game round. Within this function, dealPlayers and dealDealer were invoked to distribute cards to players and the dealer, based on previously configured settings. The game mechanic functions rely on control functions such as shuffle, dealSingle, ejectAll, turnAbsolute, jiggle, and waitLoad to perform specific tasks like shuffling the cards, dealing individual cards, ejecting remaining cards, and ensuring that the cards are properly loaded and stabilized before proceeding.

5.1.4 Error Handling and Utility

Error handling and utility are integral to the program's reliability. The pause function is employed to manage any errors that occur during game operations by halting all motor activities and awaiting user intervention to reset the system. Utility functions like clearDisplay manage the display, ensuring that only relevant information is shown to the user.

5.1.5 Shutdown Procedure

The shutdown function handles the termination of the program by stopping all motors, ejecting any remaining cards, and prompting the user to decide whether to start a new game or exit. This is called upon after indicators from game mechanic functions show that the game has ended.

5.1.6 Main Task

The main task serves as the starting point, coordinating the overall flow of the program. It first calls the config function to initialize the system and then enters a game loop that continues based on user

input. Within this loop, main invokes startMenu to gather game settings, calculates player angles using getAngleArr, and manages multiple game rounds by calling playRound for each iteration. After completing the rounds, main calls shutdown to determine whether to continue playing or end the program.

5.1.7 Overall Modularity

Overall, the program's modularity ensures that each function has a clear and distinct responsibility, promoting maintainability and scalability which is important to any program. Initialization sets the stage, user interaction gathers necessary settings, core game functions handle the main operations, utility and error handling support smooth and reliable performance, and the main task orchestrates the entire workflow. Concurrent tasks like the jiggle task enhance reliability without interfering with primary functions.

5.2 Task list and task list changes

5.2.1 Startup

First the robot must display the start menu interface. Using EV3 buttons, the player can then specify the game mode, number of players and rounds, number of cards for each player, and for the dealer. For startup, the robot's complete task list is:

1. Displays welcome message
2. Displays currently selected game mode
    a. Cycles game modes using UP button
    b. ENTER button confirms game mode selection
    c. Both Poker and Custom game modes can be selected
3. Depending on game mode, robot prompts for number of rounds, number of players, number of cards per player, and/or number of cards for dealer.
    a. ENTER button proceeds to the next prompt
4. When all selections are made, the shuffling and dealing operations are started.

The robot can also handle unexpected results. There were limitations as to what the robot accepted as valid inputs, and the robot must deal with incorrect inputs well. The limitations are the robot does not allow for less than 1 round, below 2 or above 8 players, less than 1 card per player, and negative cards for dealer.

Previously, the team had more game modes that could be selected in the task list. However, after some changes done to the team's "custom" game mode, the team decided that the only game modes would be poker and custom, as custom was very customizable. Another change that was made was the player count, as previously the team designed the robot to be free spinning, allowing to deal to a wider

range of angles, and thus more players. But after the robot was built, the team decided to limit player count more, due to this design limitation.

5.2.2 Operation and Reaction

The robot must automatically shuffle and deals cards to each player as per the users selections given in the startup menu. The robot must be able to:

1. Waits for cards to be placed in shuffling slots
    a. Waits until cards are at rest before starting
2. Cards from both sides are shuffled (without jamming) into middle
    a. The robot “shakes” while shuffling to decrease chances of jamming
3. Robot deals cards to each player, one card at a time
4. Robot deals correct amount of cards to each player
5. Robot deals cards in random fashion
6. Robot then deals to the dealer position
7. After dealing is complete, robot ejects all remaining cards behind the dealing area
8. Robot will wait until cards are loaded back into shuffling slots (beginning of next round)
9. Repeats all steps every round (# of rounds is set by player during startup)
    a. Displays the current round number

The robot can also handle unexpected results. The robot is expected to solve issues when they arrive, and also detect what the problem is, and if there is a problem. The robot should also be able to unction with a variety of different starting variables. It should be able to:

1. Detects jamming and sorts it out through the error procedure
2. Detect if card is not dealt properly (e.g not enough cards to deal) and starts error procedure
3. Robot operates even with extra cards
4. Robot operates with different types (colour) and quality of cards
5. Robot turns to the correct spot even after physically being turned

Previously, the team designed the robot in a way where a user could take the cards out of the robot in order to shuffle the cards again. However, once construction was complete, the team decided that this would be too complicated to achieve. Instead, the robot will simply shuffle once then deal, but it would deal in a randomized manner.

5.2.3 Error Procedure and Shutdown

The robot must have a defined error procedure when a jam or other issue is detected. For the error procedure, the robot must:

1. Print error message
2. Stop all motors and turns to start position

3. Eject all cards
4. Wait for ENTER button to be clicked
5. Restart the round

The robot must also have a clear shutdown procedure, that allows the user to play again. For the shutdown procedure, the robot must:

1. Empties all cards remaining in the robot
2. Turns off motors and deactivates sensors
3. Prompts for a replay
    a. If yes, successfully restarts the code. If no, ends the program

The team didn't make any changes to this task list from when we first had it. Because it was fairly basic and does not directly require any mechanical operation for it to work, this task list was not changed.

5.2.4 Software Changes and Trade-offs

A major change made to the software was splitting up a rather large startMenu function into a handful of smaller functions. This approach allowed for more efficient and organized code and helped the troubleshooting process as there were many struggles with the start menu. A few functions mentioned in the software presentation were simply written in other functions instead of being their own. The dedicated poker function was scrapped and replaced with an array which held data for the dealing pattern for poker. This was decided to be the only pre-programmed game as it is rather different than other traditional card cards. The code was resultingly more clean and did not require a complex procedure, but simply fed an array to the general dealing function. A general dealing function was created to free up the main function and would work with different inputs depending on the game mode chosen and the settings.

5.3 Functions

5.3.1 void config() – Member 1

The config function is run at the start of the main function; it assigns the motor states to the motors and configures the mode of each sensor. All motors except the bottom rotating motor is assigned to Coast mode, which allows to free spin while being unpowered. The rotator motor, however, is set to brake mode to hold its position while dealing (e.g. shaking the table won't rotate the bot), and to also avoid overshoots on turn. This code also calibrates the gyroscope sensor and resets the heading value to 0.

5.3.2 int getScaledGyroDegrees() – Member 1

This function is called when the program accesses the gyro readings. This function is used as opposed to directly accessing the gyroscopic readings as a method to adjust for sensor errors. These comes in two forms: scale and drift errors [10]. This function is called when the program accesses the gyro readings. This function combats scale errors by applying a scale error factor to all readings from the gyroscope. The scale error factor (in this robot's case 0.99) is found by rotating the bot by hand and

comparing the expected degrees of rotation to the degrees of rotation indicated by the sensor. The difference is applied to all future readings to improve accuracy.

5.3.3 void turnAbsolute (int targetAngle) – Member 1

This function turns our robot to an absolute heading relative to the original heading. For example, if the targetAngle were set to -90, the robot would turn to 90 degrees counterclockwise relative to its starting orientation. Likewise, calling turnToAngle for a targetAngle of 0 would return the robot to face its original position. This function implements Proportional-Integral-Derivative (PID) control (see Appendix B for full calculations and explanation of PID Theory) to quickly yet accurately turn to the correct position. A PID controller is a type of closed-loop control system—also known as a feedback loop, which means it updates in real time, making it useful when unexpected scenarios occur [11]. The PID controller adjusts the power of the rotator motor in real-time as it is turning. This allows it to adjust to varying environments such as different friction levels, motors overheating etc. The control loop can predict an undershoot/overshoot and adjust the power of the motor accordingly. This allows for a reaction to changing and unique environments and handle unexpected errors.

5.3.4 bool dealSingle(bool isClose) – Member 3

This function makes the robot deal a singular card either close to the robot or far away from the robot. The robot will need to deal close to the robot depending on whether it is dealing to a dealer or to a player, since the dealer is likely sitting closer to the robot than the other players. The robot returns either true or false depending on whether the card was able to be dealt, using the color sensor (set to reflection mode) mounted directly above it. The return is used to indicate whether an error has occurred and is passed into wrapper functions.

5.3.5 void ejectAll() – Member 4

This function turns on the dealer motor and keeps it on as long as the colour sensor detects at least one card leaving the robot per second. Once no more cards are being ejected, the motor is turned off and the function ends. The purpose of this function is to remove all cards inside the chamber, whether for error handling or between rounds.

5.3.6 void waitLoad() – Member 2

This function is run to check for the cards being placed into their respective containers. It does so by using the readings of the distance sensors (ultrasonic and infrared) to determine the height of the card stack. It also ensures that the height of the stack is stable for a minimum duration. This means that the cards are stationary and is no longer being adjusted by the user

5.3.7 bool shuffle() – Member 2 & 3

The shuffle function, as the name suggests, shuffles the cards. It does so by randomizing the amount of time each motor is on for which means that the cards are essentially being placed into the pile

in a random fashion. The team referred to [12] when writing the code to randomize a value in the program. The robot can detect when the cards are finished being shuffled by using the infrared and ultrasonic sensor. Comparing the values to the known reading of when the card slots are empty allow the robot to track the progress of the shuffling. Additionally, if for any reason the shuffling process stalls out or jams, by using sensors to track the progress of the shuffling, it allows the bot to detect and return the information to other functions. This is used so that if any card jams while being shuffled, it can exit and start the error handling process. If one side finishes their half of the cards before the other side, the function automatically switches and only shuffles the side that still has cards left in order to save time.

5.3.8 task jiggle() – Member 4

The jiggle task splits the processor into a separate thread, which allows it to control the motors parallel to the main task. This task shakes the bot slightly left and right, and is used in the error handling process and during the shuffle function. This decreases the chance cards get jammed and is also aesthetically pleasing.

5.3.9 void getAngleArr(int* angleArray, int numPlayers) – Member 1 & 2

This function receives a pointer referencing an array as well as the number of players in each game. The function assigns each player an angle between -90 to 90 degrees, all even spaced out. The function is optimized to spread all the players across evenly to improve overall user experience. The values are altered directly into the input array. By passing in an array and altering directly, this is significantly more memory efficient long-term than creating a static array and passing back the pointer to that array. This angle array is later used to determine how the bot should locate and deal to each player.

5.3.10 void clearDisplay() – Member 3

Clears every single line on the display, allowing new text to be displayed properly.

5.3.11 void pause() – Member 4

This is the error handling procedure. It stops all motors and resets the robot to its original position. The robot then proceeds to eject all cards inside, and then waits for user input. This gives the user the chance to fix any error that may have occurred, such as not putting in enough cards, or the shuffler jamming. After the error has been manually fixed, the user is prompted to click the enter bot to resume the robot functions.

5.3.12 bool dealPlayers(int* settings, int* angles), bool dealDealer(int* settings) – Member 3

This procedure deals all the cards to the player depending on the settings that the user inputted. The robot creates an array which keeps track of the total number of cards each player has been dealt. For each card the robot has to deal, it randomly selects a player that still needs more cards and deals to that player. Again, the team referred to [12] when writing the code to randomize a value in the program. If at

any point, a card cannot be dealt either because there are no cards left in the chamber or because the card dealer jammed, the function will return false, which will initiate the error fixing function. Similarly the dealDealer function only deals the cards to the dealer position depending on the initial settings.

5.3.13 bool playRound(int* settings, int* angles) – Member 2

This function is a wrapper for all the individual steps needed to play out a full round of cards. It waits for the cards to be input into the robot, calling upon the waitLoad function, then shuffles the card with the shuffle function. It proceeds to deal the players and then deal the dealer. Afterwards, it ejects any remaining extra cards onto the side and returns to the original position. This function is set as a Boolean order to detect errors in any of the processes. If any of the subprocesses such as shuffle or deal, return an error this indicates that the rest of the round cannot be played out. As a result, this function returns false immediately and exits so that the error handling process can start, in order to ensure a smooth user experience even if unexpected things go wrong.

5.3.14 bool shutDown() – Member 4

This process is called at the end of a game when all rounds have been successfully completed. This process stops all motors and sensors and then resets the position of the robot while ejecting all the cards. The user is prompted to either play again or to stop playing. If the user selects to play again the function returns true, and whole game process is restarted. If the user selects false, this returns false back to the main function which is able to safely and properly end the code on the robot.

5.3.15 int getRoundCount(), int get PlayerCount(), int getCardCount(), and int getDealerCount() – Member 2

These functions are utilized in the startMenu() function (refer to 5.3.17 for details on function), and they all function nearly the same. The function makes the robot screen display the current number of the parameter you are inputting. The user can change the input values with the UP and DOWN buttons, with each button press and release making the robot display update to show the current value of the parameter the user is setting. For all the functions, the user-set values have lower limits, with a few functions defining upper limits as well for the values. The functions utilize if checks to see if the user is trying to increase or decrease the value beyond the defined limits. The robot waits for an ENTER button input and release from the player before ending the function and returning the integer value that the user set for that specific game parameter.

5.3.16 void startMenu(int* settings[]) – Member 1

The startMenu function is a user interface that allows the player to input their desired game settings/parameters. The function takes in a settings array that the settings values are stored inside (refer to 5.4.1 for more details on settings array). No return value is needed as we are altering the index values of an array, and arrays are pointers and thus any changes we make to their values are saved.

The start menu utilizes a Boolean initially set to false to check whether the game settings have been set. The robot first displays a welcome screen and message and waits for the user to press and release the ENTER button. It then moves to the next screen, which prompts the player to cycle between either poker or custom game using the UP button and pressing ENTER to select the game. The presence of both poker and custom game is to demonstrate how the team can make pre-set games for the player to choose from as well. For pre-set games, certain index values for the settings array are pre-defined, while still allowing for some player input, such as how many players there are. To set game parameters, the user is prompted through multiple screens to declare how many players/cards/rounds/dealer cards there are by calling the getRoundCount(), getPlayerCount(), getCardCount(), and getDealerCount() functions (refer to 5.3.16 for details on these functions). These functions each return an integer value that is inputted into the respective index values of the settings array. Once the user has inputted all their game settings, the Boolean from earlier is set to true, and the program exits the start menu, clears the robot display screen, and the rest of the code is run. Refer to Figure 17 for a visual representation.

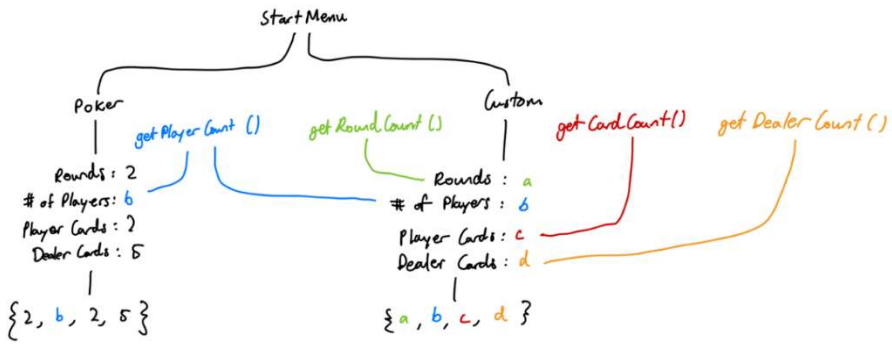


Figure 17: Visual representation of start menu and it's related get functions

5.3.17 task main()

The main function first calls upon the configuration function to ensure all the motors and sensors are properly setup and usable. It enters a loop that continues for as long as the player wants to play cards. Each loop represents a game that the user plays. Inside the loop, the startMenu function is called to get the desired game settings from the players. It proceeds to play out the correct number of rounds, while detecting if any errors occurred in the robot. If it an error is detected it calls the error handling function to fix the error before restarting the round and continuing. Once all rounds have been played the main function calls the shutdown function to determine its next course of action. If the user decides they would like to play again the loop restarts, otherwise the main function can safely end all tasks, and exit the program.

5.4 Data Storage

5.4.1 Settings Parameters Storage (Settings Array)

To allow players to create their own games, the player would be able to input their own game parameters and have the robot deal for those parameters. To store the inputted game parameters in the robot's memory, an array was utilized to keep track of them, with each parameter being stored at a different index of the array. An array was used over individual variables because the data in it is stored in memory, removing the need to use pass by reference. Arrays removed the need to input multiple game settings variables into a function, and instead just one array could be implemented into the function and then find the parameter values being looked for locally. Arrays are also memory efficient, as they are stored in memory together, one after another, and thus the robot doesn't have to expend as much memory to keep track of the random positions of individual variables.

For the array itself, the size of the array was made to be four. The 0-index represented the number of rounds, the 1-index represented the number of players, 2-index represented the number of cards per player, and 3-index represented the number of cards for the dealer (see Figure 18).

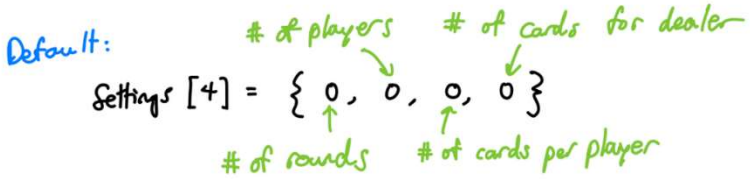


Figure 18: Visual representation of settings array

It was determined that these four parameters were sufficient to create a large variety of games and that more would be either redundant, rarely used, or beyond the scope of what the group would be able to implement with their current knowledge. As discussed previously; to use the settings array, it was inputted into the functions that needed a settings parameter. The code would then retrieve the desired index values from the array as needed (see Figure 18 for a visual representation).

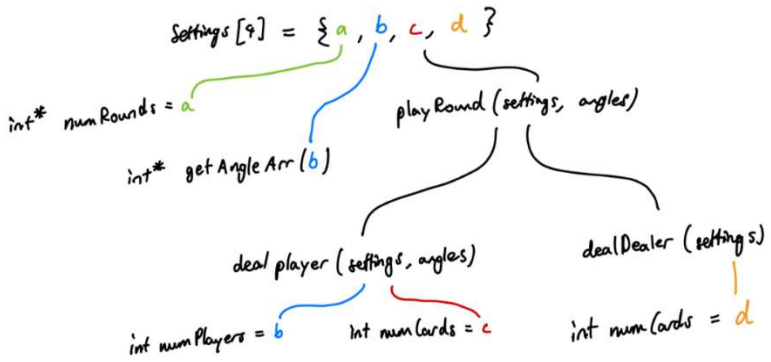


Figure 19: Visual representation of settings array being used in code

5.4.2 Player Angle Position Storage

To keep track of player location, an array was utilized (see Figure 19). As shown in 5.4.9, the getAngleArr() function takes in the number of players and divides the robot's turn angle among the players. It then assigns each player an absolute turning angle which is stored inside the angles array to the player's corresponding index. The angle values stored inside the array could then be called upon at any time by different functions by passing the array into the function before getting the index values for any player it wanted. The array also allowed for 0's as index values, as those were simply considered to be empty players.

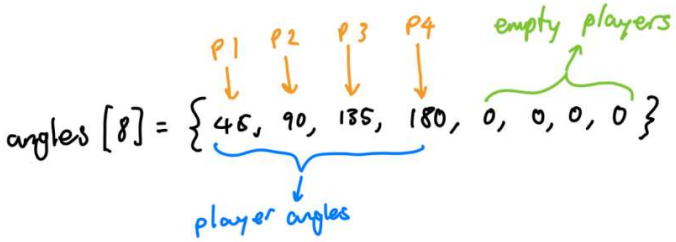


Figure 20: Visual representation of angles array

5.4.3 Player Card Count Storage

As the robot deals to players randomly, we must keep track of how many cards each player currently has. To do so, we utilized another array, with each index number corresponding to a player and the index value corresponding to how many cards they have. Similar to the player angles array, 0 values are also allowed in the cards array, with 0's being considered empty players. The cards array is utilized in the dealPlayers() function (refer to 5.3.12 for details on this function), in which a cards are dealt to players randomly.

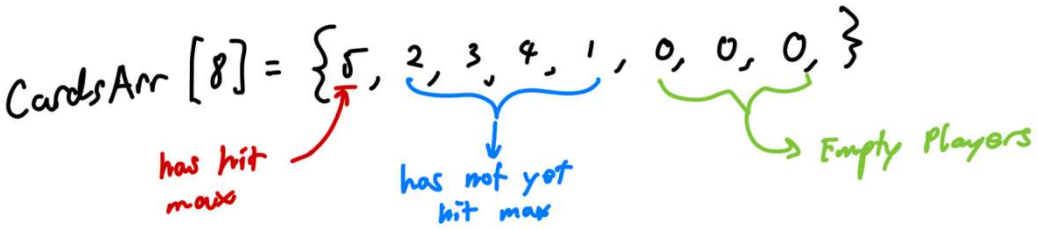


Figure 21: Visual representation of cards array

5.5 Testing

This section details the testing of all functions and the main program to evaluate the robot's performance and ensure it meets the criteria and constraints. Each function was tested under various conditions to verify its consistency and accuracy. Different functions were tested individually and cohesively in the main program to debug and troubleshoot any errors. A general test for every function was to try multiple times with different inputs/settings, then test the whole program simultaneously. Below is a detailed table which contains all the criteria the functions need to accomplish, and the testing procedures used.

5.5.1 Testing Individual Functions

Table 8: Individual functions testing and criteria

Function	Testing Procedures	Criteria
startMenu() and subfunctions	• Test multiple times with different inputs (game mode, players, cards per player, etc.) • Test with same inputs to ensure consistency • Separated into many sub functions while troubleshooting	• Displays all text properly • Buttons change displayed values on screen • Settings array values are changed and set correctly
shuffleDeck()	• Test multiple times with different number of cards (too many, very few) • Test multiple times with different types of cards (colours, thickness, condition)	• Detects that both sides have cards placed • Cards do not jam often (any jamming should be detected) • Motors stop automatically after cards are done shuffling
turnAbsolute()	• Test multiple times with different angles	• All turns are within 2 degrees of error

	• Test multiple times with unrealistic angles to test unexpected handling	• Stays within 270 degrees constraint
ejectAll()	• Test multiple times with a different number of cards • Test multiple times with different types of cards	• All cards are successfully ejected every time • Motor stops after all cards are ejected every time
waitLoad()	• Test with different types and number of cards • Test with different conditions (constantly adjusting cards, leaving hand under sensor)	• Displays a test message to the screen once a stable reading is detected (during tests only) • Only recognizes the cards once they are definitely in place (cards not being moved, no hands under sensor)
shuffle()	• Test with different number and types of cards • Test with and without jiggle task	• Jamming in 2/10 tests or less • Random shuffling pattern observed • Motors flush out any remaining cards after slots are empty
dealSingle()	• Test with different number and types of cards • Test with different motor power (close and far)	• Only one card is dealt at a time • Distinct difference between close and far • Cards land in 5-inch diameter circle of error
deal() (player and dealer)	• Test with different values for settings (amount of cards, number of players, dealers)	• Correct number of players get correct number of cards • Distinct piles of cards • Random dealing order • Works with and without dealer
playRound()	• Test with different settings and game modes • Test the same settings twice to compare differences	• Works with multiple rounds • Performs dealing successfully at least 8/10 times • Different dealing patterns for poker and custom
shutDown()	• Test multiple times with replay and without	• Automatically called after last round • User is prompted for replay

		• Stops all motors, sensors and tasks if shutdown is confirmed

Other functions not listed here only behaved one way, such as the jiggle task. These functions were simply tested alone and were considered complete once they demonstrated completion of the criteria multiple times. Since it was a matter of "it works" or "it doesn't work," there was no need to test these functions in different conditions or with different settings. An example is clearDisplay(), where text could be displayed on the screen then see if the display is cleared once the function is called. The screen could also be used as a "console" to test functions which return a value, as the screen would display the value after the function is called.

5.5.2 Main Function Testing

The testing of the main function focused on evaluating the combined tasks of all the functions. As such, the entire program was tested many times to ensure that the logic was correct, and the robot performed first the start menu, then the shuffling, then finally the dealing and repeated for the desired number of rounds. Testing of the main function all required many tests with different settings and game modes, making sure different detection cases work, such as the jamming errors. Since the code was broken into many functions, the main function was left clean and was easier to troubleshoot.

5.6 Issues Faced

5.6.1 Start Menu Bug

At one point, the settings value that the user inputted refused to increase even despite the UP button being pressed. In addition, the value was able to go below the minimum value. After diagnosing the `getRoundCount()`, `getPlayerCount()`, `getCardCount()`, and `getDealerCount()` functions, the author of the start menu function and it's corresponding get functions realized that they forgot to make sure the program only progresses upon button release instead of just button pressed. Originally, the program progressed upon button press, but didn't wait for a release. Thus the program would immediately record multiple button presses for one button press, thus it would skip over the part that says to increase the value or the checks to whether or not the value could be decreased. Regarding the problem with the value going below 0, it was also additionally found that the if conditions were written incorrectly as well. After both these issues were fixed, the start menu values were changing properly and consistently.

5.6.2 Frequent Jamming While Shuffling

Initially, the robot had trouble shuffling smoothly and reliably, often jamming and the cards getting stuck in the middle of shuffling. The issue was eventually found to be that our motor speeds were set to be running too quickly. This is because the fast the motors ran, the harder they were to stop and the more likely they were to overshoot and cause jams. To resolve this, the team incrementally reduced the

speed of the motors and tested the robot's shuffling capabilities just until the robot could shuffle properly and smoothly on a consistent basis.

6.0 Verification list and results

6.1 Overall Function

1. Must be able to deal with both a hard, smooth surface and a rougher, softer carpet

This was achieved by designing the dealing and spinning mechanisms to operate elevated from the ground. Only the supports make contact with the surface, and the dealing mechanism propels cards through the air, making the surface material irrelevant.

2. Must be able to shuffle and deal two different decks with different colours and conditions

This was confirmed through extensive testing with various decks, including old and new, red and blue, and mixed decks. The robot performed effectively across all scenarios, with occasional errors unrelated to card condition, colour, or quantity.

6.2 Start Up

1. Players can consistently set game settings to their preferences in all test cases

This was achieved as all valid inputs in the startup menu were correctly recorded and applied during testing. There are sufficient game settings available for players to create nearly any desired game.

2. Start menu displays current game mode selection correctly

The robot consistently displayed the selected game mode, which matched the mode used in the shuffling and dealing process. This alignment worked in all test cases.

3. Both Poker and Custom game modes can be selected

Both "Poker" and "Custom" game modes were easily selectable and functional in all test cases.

4. Start menu buttons display each parameter setting screen correctly

The start menu buttons accurately adjusted or set each parameter as needed. Users could easily change values and game modes, and all test cases were successful.

5. Start menu doesn't fail if you set parameters at their max and mins

The robot handled maximum and minimum parameter limits correctly by doing nothing when an invalid input was attempted. It maintained the correct value at the limit without exceeding it, working as expected in all test cases.

6. Different start menu selections change the behaviour of dealing

Start menu selections were properly stored and used during dealing. The robot consistently dealt cards according to the selected settings in all test cases.

7. Deals correct number of cards to each player every round

This was not met consistently. Errors included the robot sometimes dealing two cards simultaneously or failing to release any cards due to early motor reversal. Adjustments to the tensioning tool and motor timing are needed to improve reliability.

8. Robot automatically ejects remaining cards after each round

After each round, the robot successfully emptied all remaining cards onto a pile behind it, ensuring the card chute was clear for the next round in all test cases.

9. Robot automatically starts next round once cards are detected in shuffling slots

The robot began the next round after detecting cards in the shuffling slots. It waited for 2 seconds of no movement to confirm placement, then started shuffling. This worked in all test cases.

6.3 Shuffling

1. Shuffling must be completed in under 30 seconds

The robot consistently completed shuffling in under 30 seconds in all test cases, regardless of the number or type of cards, unless a jam occurred. In case of a jam, the robot detected it and attempted to resolve it.

2. If the cards aren't being dealt at the correct speed, the robot should know there is a jam

When cards remained in the shuffling compartments beyond the 30-second mark, the robot correctly identified a jam. It attempted to resolve the jam by reversing motor directions and moving them back and forth. This was successful in all test cases where jams were created, though persistent jams required human intervention.

3. Card detection in shuffling slots waits for sensor values to remain stable for 2 seconds

In all test cases, the robot ensured sensor values were stable for 2 seconds before initiating shuffling. This mitigated potential interference.

4. Doesn’t repeat the same shuffling pattern of a deck twice in a row

The robot shuffled the cards randomly in all test cases, with timing variations ensuring no repeated patterns. While theoretically possible, identical shuffles are highly unlikely due to program variability.

6.4 Dealing

1. Less than 2.5 seconds between the dealing of single cards

In all test cases, the robot maintained a maximum interval of 2.5 seconds between single card deals, as this matches the time needed for a 180-degree spin to reach the furthest player. Occasionally, low battery caused slight delays, but battery life is beyond the team's control.

2. Deals correct number of cards per round

This was not always achieved due to occasional jams, timing issues, or improper spacing. These issues can be addressed by adjusting the spacer and updating timing configurations. Jams were resolved either manually or automatically.

3. Deals correct number of cards per player

This constraint was also inconsistent. Timing issues sometimes caused the robot to incorrectly register a card as dealt, resulting in some players receiving the wrong number of cards. Fixes include spacer adjustments and timing updates.

4. Deals to correct number of players

The robot consistently attempted to deal to all players and the dealer based on input selections, succeeding in all test cases.

5. At most two misdeals per round

In all test cases, misdeals were kept to a maximum of two per round. Misdeals were easily resolved by discarding or retrieving extra cards from the discard pile.

6. Deals to each player in distinct piles with even spacing in between dealt card piles

The robot successfully calculated distances between players and dealt cards into distinct piles with even spacing. Distances and the number of players were determined by user inputs.

7. Dealt cards are within a 5-inch diameter margin of error away from robot

In all test cases, dealt cards landed within a 5-inch error margin due to the motor speed controlling the cards' ejection. Minor fluctuations, such as sliding on different surfaces, were observed but stayed within the margin. Discarded cards landed closer to the robot.

8. Robot repeats procedure for correct number of rounds

The robot correctly repeated the dealing and shuffling process for the inputted number of rounds. Other than occasional jams, it functioned as expected for each round.

9. Doesn’t repeat the same dealing pattern twice in a row

The robot never repeated the same dealing pattern twice during testing. While repetition is theoretically possible due to randomization, it is highly unlikely. Each card's dealing pattern is calculated independently, allowing for occasional coincidences.

6.5 Shutdown

1. Ensures cards are ejected

After completing all rounds, the robot enters the shutdown process and ensures all remaining cards are ejected. It briefly activates all motors to detect and remove cards. Once empty, it starts the next phase.

2. Shuts down all motors

In all test cases, the robot correctly turned off all motors once it confirmed there were no remaining cards, stopping any movement.

3. Returns robot to 0 degrees

After shutting down the motors, the robot returned to its default position of 0 degrees (facing straight forward) to continue the shutdown process. This was successful in all test cases.

4. Robot displays a shutdown prompt after the last round

The robot displayed a prompt asking whether to shut down the program or restart it. This worked correctly in all test cases.

5. Continues playing if "continue playing" is selected

Selecting this option restarted the program, prompting for new starting values. This functionality worked as expected in all test cases.

6. Stops program if shutdown is selected

If the shutdown option was chosen, the robot successfully stopped all tasks and ended the program in all test cases.

7.0 Project Plan

This section presents the project plan, which outlines the timeline and tasks during the development of the robot. It will explore how the team approached each deliverable and deadline, as well as how the work was divided between us.

7.1 Project Management

Prior to the beginning of the project, the group decided to have meetings on Mondays to work on the next task and/or discuss different aspects. The team agreed to work individually on most of their own tasks and come together once they are all complete to combine the work and have it ready for submission or presentations.

7.2 Task Management

The original schedule for the work is as follows, where everyone agreed to work together for most tasks. As such there is not any deliberate work distribution. There is also an estimated time frame to complete each task.

7.2.1 Original Timeline

Physical Formal Presentation → November 7

- Finish CAD model by November 4 [approximately 5 hours]
- General presentation work throughout the approaching days [approximately 2 hours]
- Request any necessary extra parts

Software Presentation → November 11

- Finish software by November 12 [approximately 3 hours]

- Prepare presentation by November 12 [approximately 2 hours]

Final Demonstration → November 21

- 3D print any parts necessary by November 2-3
- Assembly begins as soon as possible (~ november 5)
- Finish robot by November 17 [many hours]
- Testing and troubleshooting november 17 and 18

Final report → December 3

- Finish draft by November 29 [5-8 hours?]
- Review and finishing touches November 29 to December 2

Many changes were made from this original draft as plans had changed. For the physical presentation, rather than a CAD model of the robot, which would be quite time consuming, a nearly finished version of the robot was demonstrated using the default system software to demonstrate the functionality of the motors. The software presentation was misunderstood and instead of finishing the code by the deadline, it was simply needed to design the software, which resulted in less time needed for that deliverable. For the final demonstration, it was decided to use only LEGO to construct the robot, so there was no need for modelling or 3D print any extra components. Due to these differences in the tasks, an updated breakdown of the project is as follows.

7.2.2 Revised Timeline

Table 9: Actual timeline with divided workload

Task	Individual Member Tasks	Deadline
Project Proposal	All members to work on this together during team meeting	October 11, 2024 (MTE100 LEARN)
Informal Presentation	Member 1: Project Reasoning Section Member 2: Robot Tasks Member 3: Robot Requirements Member 4: Scheduling, feedback and questions	October 31, 2024 (MTE100 Tutorial)
Physical Formal Presentation	Member 1: Recap and Design Problems Member 2: Physical System Progress Member 3: Motors and Inputs Member 4: Constraints and Design Changes	November 7, 2024 (MTE100 Tutorial)

Software Formal Presentation	Member 1: Robot Summary and Task List Member 2: General Operation and Sensors Member 3: Data Organization and Function List Member 4: Unexpected Behavior and Testing In addition to these tasks, every member created at least 2 non-trivial functions for the robot.	November 11, 2024 (MTE121 Lab)
Final Demonstration	Member 1: Creating Constraints/Task List Member 2: Creating Testing Procedures Member 3/4: Robot Testing	November 21, 2024 (MTE100 Tutorial)
Final Report	Member 1: Introduction, Software, Formatting, Conclusion Member 2: Software and Mechanical Member 3: Scope, Constraints/Criteria, Project Plan, Mechanical Member 4: Software and Verification All members work on reviewing and the recommendation section.	December 3, 2024 (MTE121 and MTE100 Crowdmark)

Prior to the deadline for every deliverable, the team aimed to finish the work around a day in advance so discussion or reviewing aspects or presentation preparation could occur before it was due. In terms of the physical development of the robot, this was done at a member's dorm where the other members would come over to assist with building the robot. The software portion had each member write their own function then a group effort to combine code and write the main function. For some tasks, the team split them further among members to split up big functions into smaller functions for troubleshooting.

7.3 Revisions

Some tasks were moved around to accommodate members who were unable to make it to meetings or completing their tasks for any reason. Generally the project plan was upheld throughout the project and the team incorporated extra meetings on top of the Monday meetings to make sure all the tasks were completed on time.

7.3.1 Timing Differences

Some tasks took longer than expected to complete, such as the software prior to the final demonstration. The troubleshooting took especially long as combining code from four people was

underestimated in the original plan. As a result, instead of finishing the code a few days in advance, more work had to be done to complete it just before the presentation. Most of the other sections saw task being complete around one day late as there was other work to complete at times.

8.0 Conclusions

To solve the drawbacks of using human card dealers, such as their unpredictability and ability to cheat/conspire with players or rig games, an automated robot card dealer and shuffler was built. Throughout the development of the robot, the team encountered significant challenges and failures that shaped the final design. These challenges highlighted the goal to achieve a balance between functionality, simplicity, and consistency.

The primary and most important mechanical design features of the project were the shuffling, dealing, rotation, and dealing mechanisms. These features also presented challenging design obstacles and demonstrates times where the team had to redesign. For the shuffler, the original goal was to be able to take out the deck after it was shuffled, then reinsert it into the shuffling mechanism again. However, this design proved overly complex due to the difficulty of removing cards from the dealing slot. To address this, the robot was limited to a single shuffle per round and software-driven randomness was introduced to ensure effective shuffling. Another notable failure involved the rotation mechanism, as the original design aimed for a full 360-degree rotation. The team was not able to accomplish this due to wiring constraints, and thus the robot was limited to only 270-degrees of rotation. Despite reducing functionality, the change improved reliability and fit within intended use cases (i.e. semi-circular casino tables). The team attempted to stay aligned with their original design philosophy of compactness and reliability. For example, a simpler bottom-dealing roller system was chosen despite lacking in sophistication and a slower motor speed was used in order to minimize the risk of jamming and maintain precision. The goals that couldn’t be achieved in this design process could be goals for future expansions of this project.

The complex software design, through the usage of PID control, error handling, and modular functions, allowed for smooth operation despite mechanical and environmental challenges. Additionally, inclusions of functions such as jiggle() implemented parallel process that mitigated the frequency of task failures and increasing consistency. To successfully implement these software design choices, it required careful and clever memory management and debugging.

During testing, the robot was successfully able to meet a large majority of the constraints and criteria. However, the robot was unable to perform tasks related to dealing to the standards the team set in their criteria. The robot sometimes dealt two cards simultaneously or no cards at all due to timings in motor reversal. The correct number of cards per round were not dealt as well, as a result. These errors in

the robot's operation highlight possible areas for improvement in spacer sizes, timing of motors and program, and tensioning of cards.

The failures and subsequent design changes encountered throughout the project highlighted the importance of considering multiple solutions to the same problem, adaptability, and reliability over pushing the limit as far as possible. From this project, it is evident that despite following an initial vision for a project, one should stay flexible and be willing to change

9.0 Recommendations

This section outlines any changes that would be made to the design if there had more time to work on it. These changes are both mechanical and software and are mostly ideas rather than successful designs.

9.1 Mechanical Recommendations

There are a few different changes that could be made to the mechanical portion of the robot. These changes are mostly to increase the overall reliability of the robot, rather than the overall function.

1. The tension tool used for both the card shuffler and the card dealer could be adjusted slightly, especially when dealing. This is a problem because sometimes too many cards come out of the dealing mechanism, but the robot thinks it is one card. This leads to too many cards being dealt.
2. Making the robot free-spin would be very helpful for a variety of reasons, such as allowing for a higher degree of rotation, no risk of getting stuck or twisted, and more player capacity. This could be done by attaching the motor to the robot itself rather than the base that it spins on.

9.2 Software Recommendations

There are some changes that could be made to the software portion of the robot. Like mechanical, these changes are to increase the reliability of the robot, not changing the function overall.

1. The timing of the dealing mechanism could be updated, as occasionally the robot would attempt to deal a card, but the dealing motor would retract too quickly, leading to no cards being dealt. If this was updated, a timing that had a much higher success rate could be found
2. We could code the robot with some more complex games. This would make operation of the robot better and more intuitive for the user, while making it faster overall. If these games are preloaded, the user would not need to make the game manually

References

[1] M. Rahmsdorf, "As AI Remakes the Casino Environment, Can the “Live Dealer” Survive?," 27 October 2021. [Online]. Available: https://www.linkedin.com/pulse/ai-remakes-casino-environment-can-live-dealer-survive-mridula-saini/. [Accessed 30 11 2024].

[2] S. Wong, "Dealer Cheating Methods," 13 August 2020. [Online]. Available: https://bj21.com/articles/casino-employees/dealer-cheating-methods. [Accessed 30 11 2024].

[3] D. Charns, "2 Las Vegas casino dealers accused of cheating," 8 November 2023. [Online]. Available: https://www.8newsnow.com/investigators/2-las-vegas-casino-dealers-accused-of-cheating/. [Accessed 30 11 2024].

[4] Federal Bureau of Investigation, "Casino Cheaters Caught," 16 January 2020. [Online]. Available: https://www.fbi.gov/news/stories/casino-cheaters-caught-011620. [Accessed 30 11 2024].

[5] N. Menezes, Interviewee, *Course Project Formal Presentation Feedback.* [Interview]. 5 November 2024.

[6] Off Highway Research, "Equipment Coverage of Excavators," [Online]. Available: https://offhighwayresearch.com/Equipment-Coverage/Excavators.

[7] Machine Design, "Basics of Planetary Gears," [Online]. Available: https://www.machinedesign.com/mechanical-motion-systems/article/21834331/planetary-gears-the-basics.

[8] VEX ROBOTICS, "Selecting a Drive Train," [Online]. Available: https://kb.vex.com/hc/en-us/articles/360035591572-Selecting-a-VEX-IQ-Drivetrain.

[9] M. M. S. M. R. D. Daniel Romero, "Robotic Turret Project," California Polytechnic, [Online]. Available: https://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?params=/context/mesp/article/1110/&path_info=Robotic_Turret_Project_Report.pdf.

[10] S. E. S. G. Mojtaba Fazelinia, "Stochastic analysis of drift error of gyroscope in the single-axis attitude determination," *Measurement,* vol. 237, 2024.

[11] University of Michigan Ann-Harbor, "Introduction to Control PID," [Online]. Available: https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID.

[12] Carnegie Mellon University, "Reserved Words," ROBOTC, [Online]. Available: https://www.robotc.net/teaching_rc_cortex_v2/lesson/media_files/hp_res_words.pdf. [Accessed 30 November 2024].

Appendices

Appendix A: Robot Code

```
#pragma config(Sensor, S1, leftDistance, sensorEV3_Ultrasonic)
#pragma config(Sensor, S2, rightDistance, sensorEV3_IRSensor)
#pragma config(Sensor, S3, color, sensorEV3_Color)
#pragma config(Sensor, S4, gyro, sensorEV3_Gyro, modeEV3Gyro_RateAndAngle)
#pragma config(Motor, motorA, rotator, tmotorEV3_Large, openLoop,
      reversed, encoder)
#pragma config(Motor, motorB, rightShuffler, tmotorEV3_Large, openLoop,
      reversed, encoder)
#pragma config(Motor, motorC, dealer, tmotorEV3_Medium,PIDControl, encoder)
#pragma config(Motor, motorD, leftShuffler, tmotorEV3_Large, openLoop,
      reversed, encoder)

// definition for motor port naming
#define leftDistance S1
#define rightDistance S2
#define color S3
#define gyro S4
#define rotator motorA
#define rightShuffler motorB
#define dealer motorC
#define leftShuffler motorD

// public values
const int LSDEFAULTDIS = 6; //Ultrasonic Sensor, picks up the center of the
      holder
const int RSDEFAULTDIS = 20; //Infrared Sensor
const int COLOURDEFAULTREAD = 14; //Reflected Light Mode

const int TICKSPEED = 20;
const int MAX_PLAYERS = 8;

// control functions
void turnAbsolute(int targetAngle);
void ejectAll();
void waitLoad();

bool shuffle();
bool dealSingle(bool isClose);

task jiggle();

// other functions
```

```
int getScaledGyroDegrees();
void getAngleArr(int* angleArray, int numPlayers);
void clearDisplay();
void config();
void pause();

// game functions
bool dealPlayers(int* settings, int* angles);
bool dealDealer(int* settings);
bool playRound(int* settings, int* angles);

// start menu functions
void startMenu(int * settings);
int getDealerCount();
int getCardCount();
int getPlayerCount();
int getRoundCount();

//-----------------------------------------------------------------------------

// control functions

// returns adjusted gyro value for improved performance accuracy
int getScaledGyroDegrees(){
    const float GYRODRIFTSCALE = 0.99; //Tuning value
    // function adjusts for sensor error through a proportional scale factor
    // scale factor is estimated by turning the robot physically multiple
            times and determining the percent error

    return getGyroDegrees(gyro) / GYRODRIFTSCALE;
}

// PID based turn function
void turnAbsolute(int targetAngle){
    const int TURN_TOLERANCE = 2;

    int error = 0;
    int lastError = 0;
    int integral = 0;
    int derivative = 0;

    int motorPower = 0;

    const float Kp = 0.50; //tuned value
    const float Ki = 0.03; //tuned value
    const float Kd = 0.1; //tuned value
```

```
    const float maxPower = 50;
    const float minPower = 10; //min. power that results in consistent motion

    while(true){
        int currentAngle = getScaledGyroDegrees();
        error = targetAngle - currentAngle;

        integral += error;
        derivative = error - lastError;

        motorPower = Kp * error + Ki * integral + Kd * derivative;

        //cap power at the max power
        if(motorPower > 100) motorPower = maxPower;
        if(motorPower < -100) motorPower = -maxPower;

        //ensure motor achieves at least the min power
        if(motorPower < 10 && motorPower > 0) motorPower = minPower;
        if(motorPower > -10 && motorPower < 0) motorPower = -minPower;

        motor[rotator] = motorPower;

        if(abs(error) <= TURN_TOLERANCE){
            motor[rotator] = 0;
            break;
        }

        lastError = error;
        wait1Msec(TICKSPEED);
    }
}

bool shuffle(){

    // ammount of time each side of the shuffler is on for
    int rsTime = 0;
    int lsTime = 0;

    // the min/max ammount of time each side can be on for
    // ensures proper randomization
    int minDelay = 1000;
    int maxDelay = 2000;

    // the ammount of cycles where no progress has been made
    // used to keep track of jams
```

```
int stall = 0;

// the distance from the cards to  ultrasonic sensor
// keeps track of the progress (decreasing distance indicates cards are
        leaving the holder and being shuffled)
int prevLSDistance = 0;
int prevRSDistance = 0;

bool isEmpty = false;
// begins the jiggle task which shakes the bot decrease rate of jams, and
        for aesthetic purposes
startTask(jiggle);

while (!isEmpty){

    rsTime = rand() % (maxDelay - minDelay + 1) + minDelay;
    lsTime = rand() % (maxDelay - minDelay + 1) + minDelay;

    // if the sensor read that there are no more cards left in at least
        one chamber it will flush out the other chamber to speed up the
        process
    isEmpty =!(abs (SensorValue[leftDistance] - LSDEFAULTDIS) > 1 && abs
        (SensorValue[rightDistance] - RSDEFAULTDIS) > 4);

    if (isEmpty){
        // if right side is empty and left side is not empty, it will
              turn on only the left side until it is also empty
        if (abs (SensorValue[leftDistance] - LSDEFAULTDIS) > 1){
            motor[leftShuffler] = 100;
            while (abs (SensorValue[leftDistance] - LSDEFAULTDIS) > 1){}
        }
        else {
            motor[rightShuffler] = 100;
            while (abs(SensorValue[rightDistance] - RSDEFAULTDIS) > 4){}
        }
    }
    else{
        motor[rightShuffler] = 100;
        wait1Msec(rsTime);
        motor[rightShuffler] = -30;
        wait1Msec(600);
        motor[leftShuffler] = 100;
        motor[rightShuffler] = 0;

        motor[leftShuffler] = 100;
        wait1Msec(lsTime);
```

```
            motor[leftShuffler] = -30;
            wait1Msec(600);
            motor[rightShuffler] = 100;
            motor[leftShuffler] = 0;
        }

        //checks if any progress was made on the shuffler
        if (abs(SensorValue [leftDistance] - prevLSDistance) <= 1 &&
                  abs(SensorValue [rightDistance] - prevRSDistance) <= 2){
            stall++;
            } else {
            stall = 0;
        }
        // if the shuffler hasnt made progress in 5 cycles, it will return
            false and start the jam fixing process
        if (stall >= 5 ) {
            stopTask(jiggle);
            motor[dealer] = 0;
            motor[rotator] = 0;
            motor[rightShuffler] = 0;
            motor[leftShuffler] = 0;
            return false;
        }

        prevLSDistance = SensorValue[leftDistance];
        prevRSDistance = SensorValue[rightDistance];
    }

    stopTask(jiggle);
    motor[dealer] = 0;
    motor[rotator] = 0;

    // runs the shuffler for 2.5 sec to remove any strnded cards
    motor[rightShuffler] = 100;
    motor[leftShuffler] = 100;
    wait1Msec(2500);
    motor[rightShuffler] = 0;
    motor[leftShuffler] = 0;

    return true;
}

// waits for the cards to be placed in the shuffler chamber, and settled in
      before allowing anything else to happen
void waitLoad(){
```

```
    int stableTime = 0;

    const int minMilliseconds = 2000;

    int prevRSDistance = 0;
    int prevLSDistance = 0;

    // checks if the card has been stable for at least 2000 millisecond
    while (stableTime < minMilliseconds){
        if (abs (SensorValue[leftDistance] - LSDEFAULTDIS) <= 1 || abs
                  (SensorValue[rightDistance] - RSDEFAULTDIS) <= 4 ) {
            // if no cards are in the chamber keep the resetTime at 0 seconds
            stableTime = 0;
        }
        else if(SensorValue[leftDistance] == prevLSDistance &&
            SensorValue[rightDistance] == prevRSDistance)
        {
            // if cards are in the chamber AND the sensors read a stable
                  value (meaining the users hands are in the machine anymore)
            // increases the stable time
            stableTime += TICKSPEED;
        }
        else {
            // if cards are in chamber but hands are still in the machine,
                  the robot must conitnue waiting
            stableTime = 0;
        }

        prevLSDistance = SensorValue[leftDistance];
        prevRSDistance = SensorValue[rightDistance];
        wait1Msec(TICKSPEED);
    }
}

// deals a single card
bool dealSingle(bool isClose)
{
    // cards dealt closer to the bot are at lower speeds and need a longer
            delay between due to the decreased velocity
    int speed = (isClose ? 40 : 100);
    int waitTime = (isClose ? 600 : 250);

    motor[dealer] = speed;

    // for 20 ticks of robot tick speed checks if a card exited the dealer
```

```
    for(int i=0; i<50;)
    {
        if(abs(SensorValue[color] - COLOURDEFAULTREAD) <1)
        {
            wait1Msec(TICKSPEED);
            i++;
        }
        else
        {
            wait1Msec(waitTime);
            motor[dealer] = - speed;
            wait1Msec(waitTime);
            motor[dealer] = 0;
            return true;
        }
    }

    // if no card exited the dealer when it was supposed to this returns
            false and thus begins the error fixing process
    motor[dealer] = 0;
    return false;
}

// turns the dealer on until no card exits (shoots the card out until no
      cards are left)
void ejectAll(){
    motor[dealer] = 40;
    for (int i = 0; i < 50;){
        if (abs(SensorValue[color] - COLOURDEFAULTREAD) < 1) i++;
        else i = 0;

        wait1Msec(TICKSPEED);
    }
    motor[dealer] = 0;
}

// begins a separate thread that shakes the robot back and forth until it is
      stopped
// stops automatically after 10 seconds if not stopped separately
task jiggle(){
    for (int i = 0; i < 10; i++){
        motor[rotator] = 80;
        wait1Msec(200);
        motor[rotator] = -80;
        wait1Msec(200);
        motor[rotator] = 0;
```

```
        wait1Msec(600);
    }
}

// ------------------------------------------------------
// other fucntions

// error handling function
void pause ()
{
    displayTextLine(4, "Robot has encountered an error.");
    displayTextLine(5, "Press ENTER to restart round.");

    // stops all motor motion and turns back to start pos
    stopTask(jiggle);

    turnAbsolute(0);
    motor[dealer] = 0;
    motor[rotator] = 0;
    motor[leftShuffler] = 0;
    motor[rightShuffler] = 0;

    ejectAll();

    // allows user to fix any error and waits for enter button to continue
    while(!getButtonPress(buttonEnter)) {}
    while(getButtonPress(buttonEnter)) {}

    clearDisplay();
}

// creates a STATIC array in memory which tracks the heading of each player
// returns the pointer to the array so it can be accessed by functions
void getAngleArr(int* angleArray, int numPlayers)
{
    int spacing = 180 /(numPlayers - 1);
    for(int i = 0; i < numPlayers; i++)
    {
        angleArray[i]= -90 + i * spacing;
    }
    return;
}

// process for playing the round
// returns false if the round couldnt be played out (e.g card got jammed)
bool playRound(int* settings, int* angles){
```

```
    waitLoad();
    // if shuffler encountered an error, ends round and begins error process
    if (!shuffle()) return false;
    wait1Msec(1000);

    // if dealer encountered error, stops the round and starts error process
    if (!dealPlayers(settings, angles)) return false;
    wait1Msec(1000);

    if (!dealDealer(settings)) return false;
    wait1Msec(1000);

    // turns to the side to eject all the cards as to not get in anyone's way
    turnAbsolute(135);
    ejectAll();
    turnAbsolute(0);
    return true;
}

// deals the cards given the settings and the array of player headings which
      are passed in as pointers
bool dealPlayers (int* settings, int* angles){
    int numPlayers = settings[1];
    int numCards = settings[2];
    int totalCards = numPlayers * numCards;

    int cardsArr[8] = {0,0,0,0,0,0,0,0};
    int cardsDealt = 0;

    turnAbsolute(0);
    while(cardsDealt < totalCards){
        // picks a random player to deal to
        int n = random(numPlayers - 1);
        // if player doesnt have the max number of cards deal them a card
        if (cardsArr[n] < numCards){
            turnAbsolute(angles[n]);

            // if there was an error dealing, return the dealPlayer function
                  as false
            if(!dealSingle(false)) return false;

            cardsArr[n]++;
            cardsDealt++;
        }
    }
    return true;
```

```
}

bool dealDealer (int* settings){
    turnAbsolute(0);

    int numCards = settings[3];

    for (int i = 0; i < numCards; i++){

    // deals short to the dealer position
    // if there was error dealing return the dealDealer function as false, so
            the error handling process can start
        if (!dealSingle(true)) return false;
    }

    return true;
}

void config(){

    // limited resistance from these motors means coast can help prevent them
            from overheating
    motorType[leftShuffler] = motorCoast;
    motorType[rightShuffler] = motorCoast;
    motorType[dealer] = motorCoast;

    motorType[rotator] = motorBrake;

    SensorType [rightDistance] = sensorEV3_IRSensor;
    wait1Msec (50);
    SensorMode [rightDistance] = modeEV3IR_Calibration;
    wait1Msec (50);
    SensorMode [rightDistance] = modeEV3IR_Proximity;
    wait1Msec (50);

    SensorType [leftDistance] = sensorEV3_Ultrasonic ;
    wait1Msec (50);
    SensorMode [leftDistance] = modeEV3Ultrasonic_Cm;
    wait1Msec (50);

    SensorType [color] = sensorEV3_Color ;
    wait1Msec (50);
    SensorMode[color] = modeEV3Color_Reflected;
    wait1Msec (50);

    SensorMode[gyro] = modeEV3Gyro_Calibration;
```

```
    wait1Msec (50);
    SensorMode[gyro] = modeEV3Gyro_RateAndAngle;
    wait1Msec (50);
    resetGyro(gyro);
}

bool shutDown()
{
    // stop all motors
    motor[rightShuffler] = 0;
    motor[leftShuffler] = 0;
    motor[rotator] = 0;
    motor[dealer] = 0;

    // turns back to start and ejects all cards
    turnAbsolute(0);
    ejectAll();

    // displays a thank you message
    displayTextLine(5, "Thank you for playing!");

    displayTextLine(10, "Would you like to play again?");
    displayTextLine(11, "Up for yes, Down for no");

    while(!getButtonPress(buttonUp) && !getButtonPress(buttonDown)) {}
    if(getButtonPress(buttonUp))
    {
        while(getButtonPress(buttonUp)){}
        // returns false to indicate that the program will NOT shutdown
        return false;
    }
    else
    {
        while(getButtonPress(buttonDown)){}
        //returns true to indicate that the program will shutdown
        return true;
    }
}

void clearDisplay ()
{
    for (int i = 0; i<15; i++)
    {
        displayTextLine(i, "");
    }
}
```

```
//-------------------------------------------------------------------------
// start menu
int getRoundCount ()
{
    // default number of rounds is 1
    int roundCount = 1;

    displayTextLine(4, "UP and DOWN to set # rounds.");
    displayTextLine(5, "Press ENTER to proceed.");

    while(!getButtonPress(buttonEnter))
    {
        displayTextLine(6, "%d", roundCount);

        while(!getButtonPress(buttonUp) && (!getButtonPress(buttonDown)))
        {
            if (getButtonPress(buttonEnter))
            {
                while(getButtonPress(buttonEnter)) {}
                return roundCount;
            }
        }
        if (getButtonPress(buttonUp))
        {
            while(getButtonPress(buttonUp)) {}
            roundCount++;
        }
        else if (getButtonPress(buttonDown))
        {
            while(getButtonPress(buttonDown)) {}
            // min number of rounds is 1
            if (roundCount > 1)
            {
                roundCount--;
            }
        }
        while(getButtonPress(buttonUp) || (getButtonPress(buttonDown))) {}
        wait1Msec(TICKSPEED);
    }
    return roundCount;
}

int getPlayerCount ()
{
    // default num players is 2
```

```
    int playerCount = 2;

    displayTextLine(4, "UP and DOWN to set # players.");
    displayTextLine(5, "Press ENTER to proceed.");

    while (!getButtonPress(buttonEnter))
    {
        displayTextLine(6, "%d", playerCount);

        while(!getButtonPress(buttonUp) && (!getButtonPress(buttonDown)))
        {
            if (getButtonPress(buttonEnter))
            {
                while(getButtonPress(buttonEnter)) {}
                return playerCount;
            }
        }
        if (getButtonPress(buttonUp))
        {
            // max number of player is 8
            while(getButtonPress(buttonUp)) {}
            if (playerCount < MAX_PLAYERS)
            {
                playerCount++;
            }
        }
        else if (getButtonPress(buttonDown))
        {
            // min players is 2
            while(getButtonPress(buttonDown)) {}
            if (playerCount > 2)
            {
                playerCount--;
            }
        }
        wait1Msec(TICKSPEED);
    }
    return playerCount;
}

int getCardCount ()
{
    int cardCount = 1;

    displayTextLine(4, "UP and DOWN to set # cards/player.");
    displayTextLine(5, "Press ENTER to proceed.");
```

```
    while (!getButtonPress(buttonEnter))
    {

        displayTextLine(6, "%d", cardCount);

        while(!getButtonPress(buttonUp) && (!getButtonPress(buttonDown)))
        {
            if (getButtonPress(buttonEnter))
            {
                while(getButtonPress(buttonEnter)) {}
                return cardCount;
            }
        }
        if (getButtonPress(buttonUp))
        {
            while(getButtonPress(buttonUp)) {}
            cardCount++;
        }
        else
        {
            while(getButtonPress(buttonDown)) {}

            // min number of cards is at least 1
            if (cardCount > 1)
            {
                cardCount--;
            }
        }
        wait1Msec(TICKSPEED);
    }
    return cardCount;
}

int getDealerCount ()
{
    int dealerCount = 0;

    displayTextLine(4, "UP and DOWN to set # dealer cards.");
    displayTextLine(5, "Press ENTER to proceed.");

    while (!getButtonPress(buttonEnter))
    {

        displayTextLine(6, "%d", dealerCount);
```

```
        while(!getButtonPress(buttonUp) && (!getButtonPress(buttonDown)))
        {
            if (getButtonPress(buttonEnter))
            {
                while(getButtonPress(buttonEnter)) {}
                return dealerCount;
            }
        }
        if (getButtonPress(buttonUp))
        {
            while(getButtonPress(buttonUp)) {}
            dealerCount++;
        }
        else
        {
            while(getButtonPress(buttonDown)) {}
            // dealers cannot be dealt a negative number of cards
            if (dealerCount > 0)
            {
                dealerCount--;
            }
        }
        wait1Msec(TICKSPEED);
    }
    return dealerCount;
}

// returns the game array data as a pointer using a static array
void startMenu(int *settings)
{
    bool selection = false;

    displayTextLine(4, "Hello, esteemed guest!");
    displayTextLine(5, "This is a card dealer robot.");
    displayTextLine(6, "Press ENTER to begin");

    while(!getButtonPress(buttonEnter)) {}
    while(getButtonPress(buttonEnter)) {}

    while (!selection)
    {
        displayTextLine(5, "Selected Game: Poker.");
        displayTextLine(6, "Press ENTER button to play.");
        displayTextLine(7, "Cycle to next option using UP.");

        while(!getButtonPress(buttonEnter) && !getButtonPress(buttonUp)) {}
```

```
        if (getButtonPress(buttonEnter))
        {
            while(getButtonPress(buttonEnter)) {}
            settings[0] = 2;
            settings[2] = 2;
            settings[3] = 5;
            settings[1] = getPlayerCount();

            clearDisplay();

            return;
        }
        while(getButtonPress(buttonUp)) {}

        displayTextLine(5, "Selected Game: Custom.");
        displayTextLine(6, "Press ENTER button to play.");
        displayTextLine(7, "Cycle to next option using UP.");
        while(!getButtonPress(buttonEnter) && !getButtonPress(buttonUp)) {}
        if (getButtonPress(buttonEnter))
        {
            while(getButtonPress(buttonEnter)) {}

            settings[0] = getRoundCount();
            wait1Msec(TICKSPEED);

            settings[1] = getPlayerCount();
            wait1Msec(TICKSPEED);

            settings[2] = getCardCount();
            wait1Msec(TICKSPEED);

            settings[3] = getDealerCount();
            wait1Msec(TICKSPEED);

            selection = true;
        }
        while(getButtonPress(buttonUp)) {}
    }

    clearDisplay();

    return;
}

task main()
{
```

```
    config();

    bool playing = true;
    while(playing)
    {
        int settings[4] = {0,0,0,0};
        // passes the array in as a pointer and the values are directly
            changed
        // this is the improves memory management
        startMenu(settings);

        int angles[MAX_PLAYERS] = {0,0,0,0,0,0,0,0};
        getAngleArr(angles, settings[1]);

        int numRounds = settings[0];

        for (int i = 0; i < numRounds;){
        displayTextLine (5 ,"Round: %d", i+1);
            // passes in the arrays to the play round function
            if(playRound(settings, angles)){
                i++;
            }
            // if the round wasnt played properly, activate the error mode
                  (pause) and dont count the round
            else pause();
        }

        // if not shutting downn, keep playing, else set playing to false
        playing = (shutDown() ? false : true);
        clearDisplay();
    }
    // stops all task ONLY AFTER RUNNING A SEPERATE shut down function.
    stopAllTasks();
}
```

Appendix B: Explanation of PID Control Theory

PID stands for Proportional, Integral and Derivative, the three key components of this control system. PID is classified as a closed feedback loop, meaning it uses constantly updating data to adjust the output of motors, which results in a higher level of accuracy. The backbone of the PID controller is the error value which is the difference between the set point (SP) which is also known as the target and the current process variable (PV) also known as the current position.

PID essentially applies a weighted combination of the proportional, integrated and derived terms, and adjusts the controller system to bring the PV closer to the SP, in other words to minimize the error.

The benefits of PID include adjustment to uncertain environments (changes in friction, physical resistance, overheating motors etc.), which help build a robust and adaptable program, which is able function properly even in a variety of different environments.

Calculations:

Error (e(t))

The error value is the difference between the current state and desired state, and is calculated as:

$$e(t) = SP - PV$$

Where SP is the target value, and PV is current value.

Proportional (P)

The proportional control action is calculated as:

$$P = Kp \times e(t)$$

Where Kp is a tuning parameter and e(t) represents the error at time t.

Integral (I)

The integral term addresses the steady state error, by summing up past errors over time. It is calculated as:

$$I = K_i \cdot \int_0^t e(\tau)\, d\tau$$

Where Ki is the integral tuning parameter and, the integral is the sum of all errors over time. In implementation, the integral is a carried summation of the error variable.

Derivative (D)

The derivative term predicts future system behavior by evaluating the rate of change of the error:

$$D = K_d \cdot \frac{de(t)}{dt}$$

Where Kd is the derivative gain, and the derivative represent the rate of change. In implementation the derivative is estimated by subtracting the current error by the previous error.

PID Control Law

The final control signal, u(t) is the sum of the proportional, integral and derivative contributions:

$$u(t) = P + I + D$$

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau)\, d\tau + K_d \cdot \frac{de(t)}{dt}$$