Autonomous Card Sorting Robot

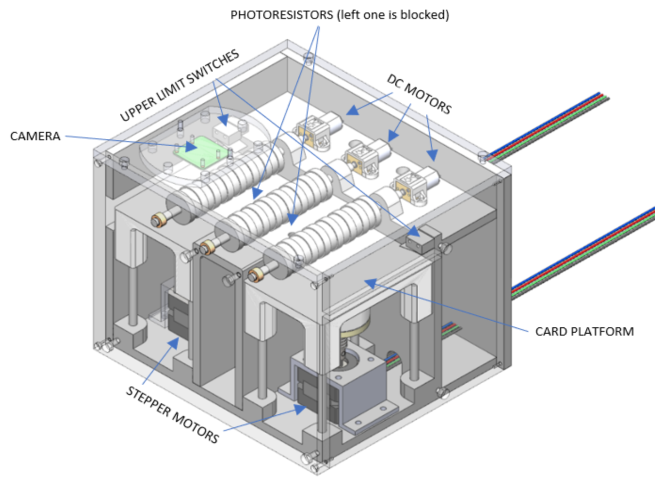
David Guo, Jenny Mei, Kenneth Song, Taewon Kim

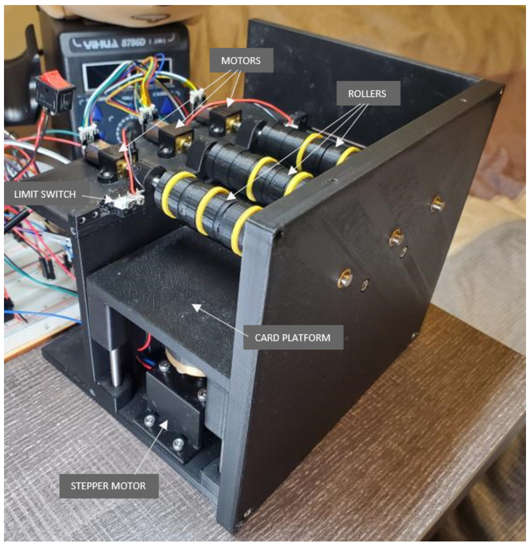
Selected Opportunity

We aim to autonomously manipulate and sort a stack of playing cards.

Device Diagrams

Our CAD of the assembly is included on the left for clarity. We removed the top and right panels on our device for the photo on the right.





Product Description

From a high-level perspective, our machine uses a series of three rollers to move cards between two platforms, one at a time. These platforms are constrained to move vertically and are very precisely actuated via stepper motors and lead screws. Limit switches are placed below each platform to help the program calibrate the linear position of the platforms, and above each platform to locate the top of the card stack. Above the left platform is a camera, which we use alongside computer vision to identify cards and store their information in memory. When the machine is powered on, the first platform is lowered to reveal an entrance slot that allows the user to place cards onto the platform.

Once the “sort” button is pressed, the platforms are raised to apply pressure on the cards, and the rollers are used to transfer cards back and forth. In an initial “sweep”, all cards are transferred from one platform to the other as the camera scans through the deck. Following this, the order in which cards should be ejected is calculated in-memory, and the rollers rearrange and dispense the cards as is needed. Two photoresistors are placed underneath the center platform to detect when a card has passed over to the other platform. Combined with precise control of the platform height, this allows us to ensure that multiple cards are not passed through at once, and maintain an accurate count of the number of cards in each pile.

While our final product successfully achieved the mechanical functionality we desired, the computer vision code proved to be somewhat less reliable. As a result, we chose to highlight the mechanical capabilities of our machine in our final demonstration.

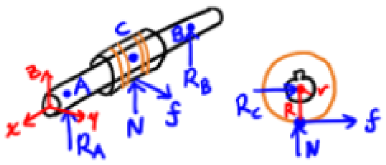
Critical Decisions and Calculations

Platform Height Control: Lead Screw + Stepper Motor Calculations

Each platform is supported by four linear shafts to constrain rotational motion, and to ensure that the stepper motor shaft and the lead screw do not experience any radial loading. The amount of precision we had in the control of the height of the platforms was dependent on the fidelity of the stepper motor and the thread pitch of the lead screw. We chose an M10x2.0 lead screw, meaning a full revolution resulted in 2mm of linear travel. The stepper motor we chose had a step angle of 1.8° which results in 200 steps per revolution of the stepper motor shaft that we could accurately control. This gives us a theoretical minimum linear travel per step of: $l_{step} = \frac{2\,mm}{1\,rev} * \frac{1\,rev}{200\,steps} = 0.01\frac{mm}{step}$, which is more than enough fidelity for our purposes (one card is approximately 0.3 mm thick.

Roller Shaft and DC Motor Radial Force Calculations

Although it would have been ideal to utilize shims and a flexible shaft collar to minimize the effects of manufacturing defects and misalignments between the motor and the roller shafts, we were limited by our project budget to do so. We had to make some non-ideal design choices in our shaft mounting design, and had to forego the inclusion of a flexible shaft collar and use bronze bushings instead of ball bearings.



We performed calculations for the maximum amount of loading we expected to see for the shaft. This was accomplished using a simple statics approach, using the sum of forces in each direction as well as a sum of moments about point B. Assuming the normal and friction forces from the rubber rollers act upon a single point C, we determined that the maximum reaction force would act upon the bronze bushing at point B with a magnitude of 17.7 N.

Manufacturing Methods

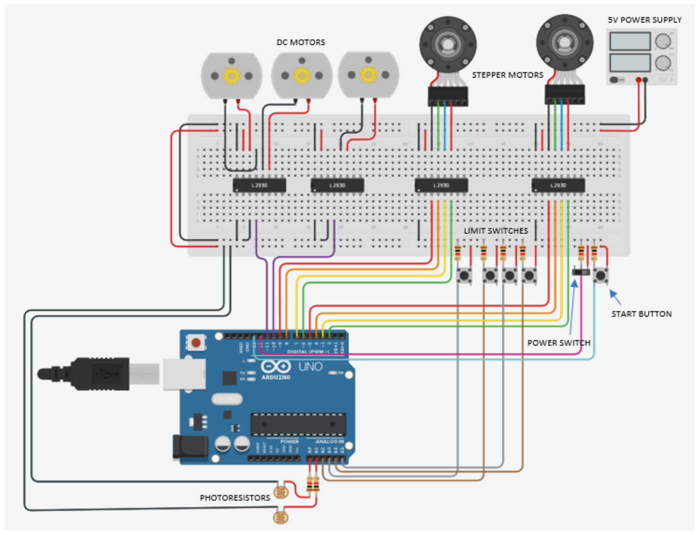
In order to accomplish quick turnarounds and relatively accurate hole positions, we decided to use additive manufacturing to manufacture the housing for the device. We found it relatively easy to 3D print various features such as counterbored holes and push-fit bores for bronze bushings directly into the plates of the housing. By printing slightly undersized holes and manually tapping threads into the plastic material, we could easily assemble various parts of the housing together.

All shaft components of the device had to be machined using more traditional subtractive manufacturing methods. Because we did not expect to see large axial loads during the operation of any rotary elements, we determined that we could reliably use set screws to constrain rotational motion of components. The three roller shafts had an opening bored to allow the DC motor shaft to be inserted, and then a set screw hole was drilled and tapped. Because the lead screw had to be machined with precision to interact with the lead screw nut, we decided to purchase these items off-the-shelf, and then turn down one end of the lead screw and add a set screw hole to constrain it to the stepper motor shaft.

Raspberry Pi

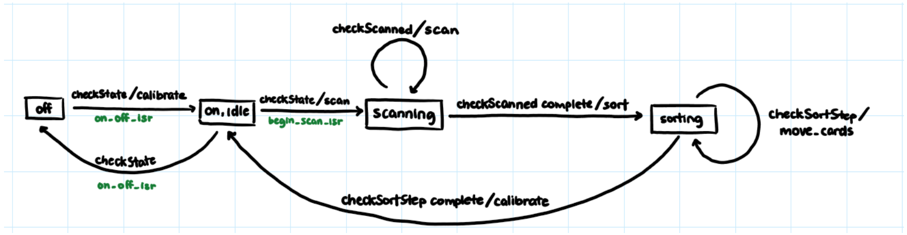
Due to the excessive number of GPIO pins we needed, as well as the need to load in and utilize a computer vision library, our group chose to use a Raspberry Pi to run our machine's logic. As a result, we were asked to write some additional code to demonstrate our understanding of the ESP32--this is addressed in the Appendix.

Circuit Diagram



State Transition Diagram

Since the button presses that turn the machine on and off are handled by interrupts, they modify our state variable (an int) directly--hence, our "event checker" for these is simply the outer "checkState" function, which returns the state variable. The relevant interrupt handler function is written in green for clarity.



Reflection

Our team worked quite well together and all members contributed to the project. This may owe to strong communication within the team and clear delegation of specific deliverables. In spite of our busy schedules, we always found time to discuss/work on the project together--whether by calling or meeting in-person. Time management with long-term projects such as these is always difficult; setting specific deadlines/milestones for the team may have been a good way of ensuring work is on track.

Appendix

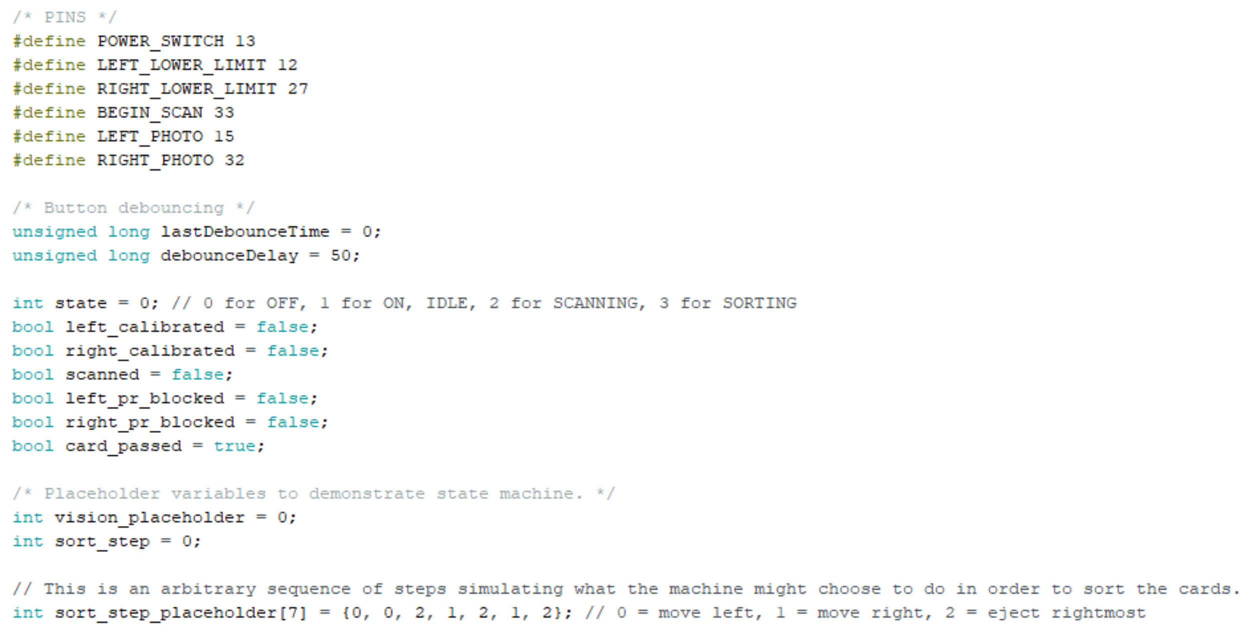
In order to demonstrate our understanding of the Arduino-style coding taught in class, we were instructed to rewrite our state machine on the ESP32 and submit it with this report.

Various simplifications were made to the code in order to account for our inability to actuate motors (due to the limited number of pins) and the lack of computer vision.

Namely, motor control code has been replaced with print statements--examples of what the code might look like are placed in comments.

Cards are not actually scanned or sorted; instead, our “scan_card” function simulates scanning over three cards by incrementing a counter, and our “checkSortStep” function simply returns the next maneuver to perform out of a predetermined sequence of steps (itemized in the sort_step_placeholder array).

Also submitted alongside this report is a video demo of our ESP32-controlled state machine.



```
/* PINS */
#define POWER_SWITCH 13
#define LEFT_LOWER_LIMIT 12
#define RIGHT_LOWER_LIMIT 27
#define BEGIN_SCAN 33
#define LEFT_PHOTO 15
#define RIGHT_PHOTO 32

/* Button debouncing */
unsigned long lastDebounceTime = 0;
unsigned long debounceDelay = 50;

int state = 0; // 0 for OFF, 1 for ON, IDLE, 2 for SCANNING, 3 for SORTING
bool left_calibrated = false;
bool right_calibrated = false;
bool scanned = false;
bool left_pr_blocked = false;
bool right_pr_blocked = false;
bool card_passed = true;

/* Placeholder variables to demonstrate state machine. */
int vision_placeholder = 0;
int sort_step = 0;

// This is an arbitrary sequence of steps simulating what the machine might choose to do in order to sort the cards.
int sort_step_placeholder[7] = {0, 0, 2, 1, 2, 1, 2}; // 0 = move left, 1 = move right, 2 = eject rightmost
```

```
/* INTERRUPT HANDLERS and SETUP */
void IRAM_ATTR on_off_isr() {
  if ((millis() - lastDebounceTime) > debounceDelay) {
    if (state == 0) {
      // Print statements shouldn't be in interrupts, but this is the most convenient way to display the state machine.
      Serial.println("STATE = 1: ON");
      state = 1;
    } else {
      Serial.println("STATE = 0: OFF");
      state = 0;
    }
  }
  lastDebounceTime = millis();
}

void IRAM_ATTR left_min_isr() {
  if (!left_calibrated) {
    Serial.println("LEFT CALIBRATION COMPLETE");
    left_calibrated = true;
  }
}

void IRAM_ATTR right_min_isr() {
  if (!right_calibrated) {
    Serial.println("RIGHT CALIBRATION COMPLETE");
    right_calibrated = true;
  }
}

void IRAM_ATTR begin_scan_isr() {
  if ((millis() - lastDebounceTime) > debounceDelay && state == 1) {
    state = 2;
    Serial.println("STATE = 2: SCANNING CARDS");
  }
  lastDebounceTime = millis();
}

void setup() {
  Serial.begin(115200);
  pinMode(POWER_SWITCH, INPUT);
  attachInterrupt(POWER_SWITCH, on_off_isr, CHANGE);

  pinMode(LEFT_LOWER_LIMIT, INPUT);
  pinMode(RIGHT_LOWER_LIMIT, INPUT);
  attachInterrupt(LEFT_LOWER_LIMIT, left_min_isr, RISING);
  attachInterrupt(RIGHT_LOWER_LIMIT, right_min_isr, RISING);

  pinMode(BEGIN_SCAN, INPUT);
  attachInterrupt(BEGIN_SCAN, begin_scan_isr, RISING);

  pinMode(LEFT_PHOTO, INPUT);
  pinMode(RIGHT_PHOTO, INPUT);

  Serial.println("INITIAL STATE: OFF");
}
```

```
/* EVENT CHECKERS */
int checkState() {
  return state;
}

int checkCalibration() {
  if (!left_calibrated && !right_calibrated) {
    return 0;
  } else if (left_calibrated && !right_calibrated) {
    return 1;
  } else if (right_calibrated && !left_calibrated) {
    return 2;
  } else {
    return 3;
  }
}

int checkScanned() {
  if (!card_passed) {
    return 1;
  } else if (card_passed && !scanned) {
    card_passed = false;
    return 0;
  } else {
    card_passed = false;
    return 2;
  }
}

int checkSortStep() {
  if (sort_step == 7) {
    return 3;
  }
  if (!card_passed) {
    return sort_step_placeholder[sort_step];
  } else {
    card_passed = false;
    sort_step += 1;
    return sort_step_placeholder[sort_step];
  }
}

void check_photoresistor(int side) {
  if (side == 0) { // left side
    int value = analogRead(LEFT_PHOTO);
    if (value < 100) {
      Serial.println("Left photoresistor blocked");
      left_pr_blocked = true;
    } else if (value > 200 && left_pr_blocked) {
      left_pr_blocked = false;
      card_passed = true;
      Serial.println("Card has passed");
    }
  } else if (side == 1) { // right side
    int value = analogRead(RIGHT_PHOTO);
    if (value < 100) {
      Serial.println("Right photoresistor blocked");
      right_pr_blocked = true;
    } else if (value > 200 && right_pr_blocked) {
      right_pr_blocked = false;
      card_passed = true;
      Serial.println("Card has passed");
    }
  }
}
```

```
/* SERVICE ROUTINES */
void calibrate() {
  switch (checkCalibration()) { // 0 for NONE, 1 for LEFT ONLY, 2 for RIGHT ONLY, 3 for BOTH (DONE)
    case 0:
      step_down_left();
      step_down_right();
      break;
    case 1:
      step_down_right();
      break;
    case 2:
      step_down_left();
      break;
    case 3:
      Serial.println("CALIBRATION COMPLETE, IDLING");
      break;
  }
}

void scan() {
  switch (checkScanned()) { // 0 = scan top card, 1 = roll cards without scanning, 2 = complete
    case 0:
      scan_top_card();
      move_cards_right();
      break;
    case 1:
      move_cards_right();
      break;
    case 2:
      Serial.println("SCANNING COMPLETE");
      Serial.println("STATE = 3: SORTING");
      state = 3;
  }
}

void sort() {
  switch (checkSortStep()) {
    case 0: // Move card left
      move_cards_left();
      break;
    case 1: // Move card right
      move_cards_right();
      break;
    case 2: // Eject rightmost card
      eject_card();
      break;
    case 3: // Finished sorting
      left_calibrated = false;
      right_calibrated = false;
      scanned = false;
      state = 1;
      Serial.println("SORTING COMPLETE");
      Serial.println("STATE = 1: IDLE");
      break;
  }
}
```

```
void step_down_left() {
  // Placeholder for left stepper motor
  // left.step(-30);
  Serial.println("Left motor stepping down");
}

void step_down_right() {
  // Placeholder for right stepper motor
  // right.step(-30);
  Serial.println("Right motor stepping down");
}

void step_up_left() {
  // Placeholder for left stepper motor
  // left.step(30);
  Serial.println("Left motor stepping up");
}

void step_up_right() {
  // Placeholder for right stepper motor
  // left.step(30);
  Serial.println("Right motor stepping up");
}

void roll_left(bool centerOnly) {
  if (centerOnly) {
    // digitalWrite(MOTORPIN, HIGH);
    // delay(500);
    // digitalWrite(MOTORPIN, LOW);
    Serial.println("Center roller spinning left");
  } else {
    Serial.println("All rollers spinning left");
  }
}

void roll_right(bool centerOnly) {
  if (centerOnly) {
    Serial.println("Center roller spinning right");
  } else {
    Serial.println("All rollers spinning right");
  }
}

void move_cards_left() {
  Serial.println("Moving card left");
  if (left_pr_blocked) {
    roll_left(true); // Only roll center to prevent grabbing extra cards
  } else {
    step_up_right();
    roll_left(false);
  }
  check_photoresistor(0);
}
```

```
void move_cards_right() {
  Serial.println("Moving card right");
  if (right_pr_blocked) {
    roll_right(true); // Only roll center to prevent grabbing extra cards
  } else {
    step_up_left();
    roll_right(false);
  }
  check_photoresistor(1);
}

void eject_card() {
  Serial.println("\nEJECTING CARD, ROLLING RIGHTMOST MOTORS RIGHT\n");
  sort_step += 1;
}

/* A placeholder for computer vision code; assumes there's only 3 cards in the stack.
   Will "complete" scanning after running 3 times. */
void scan_top_card() {
  Serial.println("\nSCANNING CARD #" + String(vision_placeholder) + "...");
  // Normally, card_found would be set by some call to the computer vision algorithm
  // If computer vision fails to identify a card, card_found = false.
  bool card_found;
  if (vision_placeholder < 3) {
    card_found = true;
    vision_placeholder += 1;
  } else {
    card_found = false;
  }

  if (!card_found) {
    scanned = true;
  }
  delay(1000);
  Serial.println("DONE SCANNING\n");
}


/* MAIN LOOP */
void loop() {
  switch (checkState()) {
    case 0: // OFF: do nothing
      left_calibrated = false;
      right_calibrated = false;
      scanned = false;
      break;
    case 1: // ON, IDLE: Calibrate platforms
      calibrate();
      break;
    case 2: // SCANNING: Scan cards with computer vision
      scan();
      break;
    case 3: // SORTING:
      sort();
      break;
  }
  delay(2000); // So the serial monitor isn't flooded with print statements
}
```