Autonomous French Press: ‘Brewt Force’

Jacob Kuczynski, Jacob Lopez, Eduardo Diaz

OPPORTUNITY

Our goal through this project was to automate a home activity in order to make people’s lives easier. A french press is a simple appliance that is easy to operate, however it takes time to use, and can be imprecise depending on the operator. This is especially true since you need to wait 5 minutes between adding hot water and pressing the plunger. Therefore, we built an automated french press with the goal of making the coffee making process more consistent and convenient. In addition, utilizing a french press may prove difficult for some users with disabilities. For added accessibility, we designed the french press to be simplistic and require minimal input from the user to operate.

HIGH LEVEL STRATEGY

Our final design of the project uses a rack and pinion system driven by a brushed DC motor in order to press the plunger rod down. We retrofitted the plunger rod so that it is connected to the gear rack and designed a custom lid to hold the transmission system and electronics. In order to operate the machine, the process is as follows:

1. The system first begins in the IDLE state in which all processes are currently waiting. Once the user would like to begin the coffee-making process, they would indicate beginning through the use of a button.
2. The system then enters the MOVING state in which through open-loop control, the transmission line continuously actuates the movement of the plunger upwards. While still in this state of movement, the user would press a limit-switch button, indicating they would like to input the boiling water and coffee grounds.
3. The system then enters the WAIT state in which the coffee grounds and boiling water would be added. This is also the time for the user to stir the grounds for proper mixture. The user would have specified how long they would like their coffee to sit prior to beginning the entire process.
4. Finally, the system enters the DOWN state in which the transmission line actuates the plunger downwards. During this state, the controller is continuously detecting any sort of change in the load cell. When it does, the system then returns to IDLE state.
5. At this point, the coffee is ready to be served!

This differs from our original design which used two motors, one on each side of the plunger that drove rubber wheels to move the plunger rod via friction. We changed this design after the machine shop staff voiced concerns that there may not be enough friction between wheels and the plunger rod. Our original design also included a thermocouple and water heater feedback loop that maintains the water temperature, however due to time constraints, we focused on the main functionality of the pressing motion of the press. Given time and adequate resources, the aforementioned would be implemented. In addition, the load cell would be faceted in the proper location, at the bottom of the pot, detecting a change in weight through a proper cantilever setup. As it stands, the current model improperly pushes on the middle of the load cell.

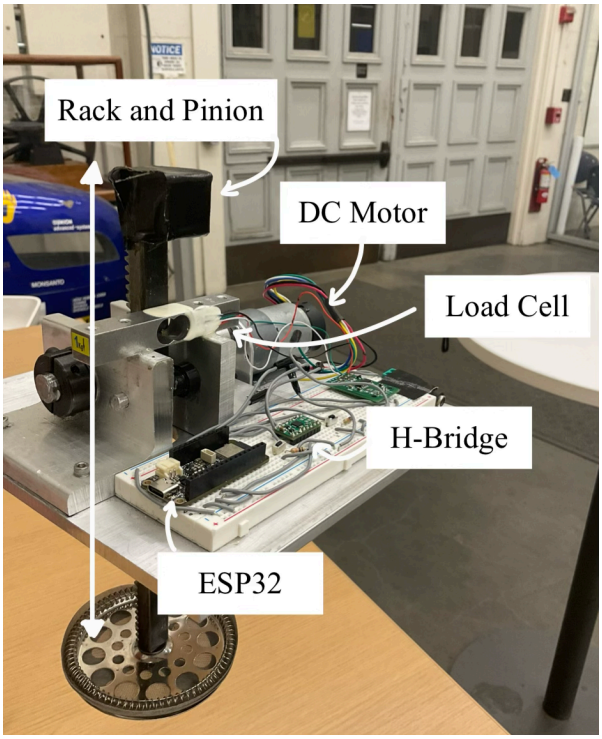


Figure 1: High-Level Electronic Overview (White line describes the 1 D.O.F. motion)

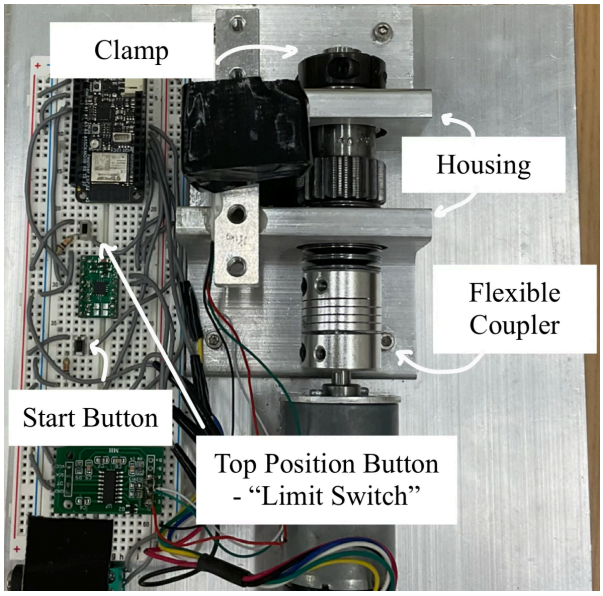


Figure 2: Top-Down Overview of Transmission Line, Buttons, and Housing.

DESIGN DECISIONS

One of the first concerns when designing and selecting components was the required force it takes to actuate the plunger/rack itself. For calculating the torque, we neglect the friction force between the disk & pot as its minimal. For calculating the required torque:

$T_{required} = F * r * sin\theta = 4.064 \frac{kg}{cm}$ where the calculated force is simply the weight of the rod: $F = 3.2\ kgf$. Radius of the gear rack is : $r = 0.5\ in.\ or\ 1.27cm.$ Angle of $\theta = 90°$

As the Pololu 37D Metal Gearmotor has a maximum operating torque of $8.4\ \frac{kg}{cm}$, we were fine when it came to producing enough torque for actuation. As for the safety factor, we calculated a value of 2.07, ensuring proper safety of handling our equipment:

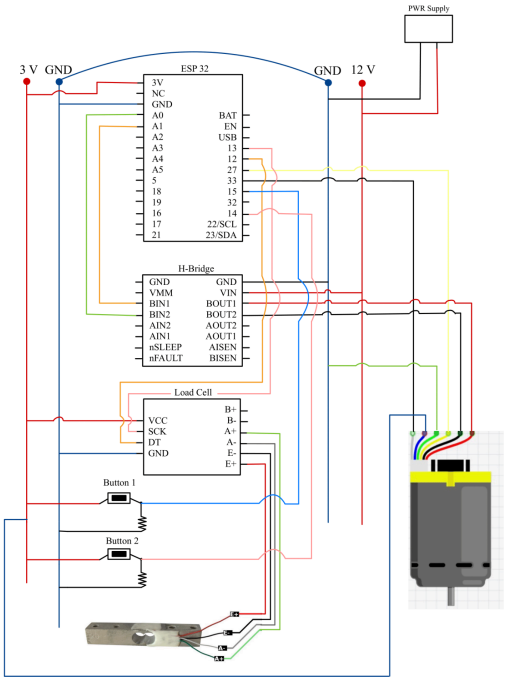
$$FOS = \frac{T_{motor}}{T_{required}} = 2.07 \text{ where } T_{motor} = 8.4\frac{kg}{cm}, T_{motor} = 4.064\frac{kg}{cm}$$

Finally, we wanted to ensure that the bearing load rating for commonly used bearings (including the ones utilized within our product) did not exceed the minimum of 500 lb (in the case there was ever a load applied to the bearing itself):

$$L_{bearing\ rating} > L_{applied} = 500\ lb\ >\ 7.05\ lb \text{ where } L_{bearing\ rating} = 500\ lb,$$
$$L_{applied} = 3.2\ kg\ *\ 2.204\ \frac{lb}{kg} = 7.05\ lb$$

In addition, to protect the motor from supporting any kind of radial load, a flexible shaft coupling was incorporated into the transmission system. We also inserted belleville washers and shims between the gear, bearings, and collar to protect from friction between these components. The transmission housing is made from L brackets and is attached to the custom lid with screws.

CIRCUIT DIAGRAM/STATE TRANSITION DIAGRAM



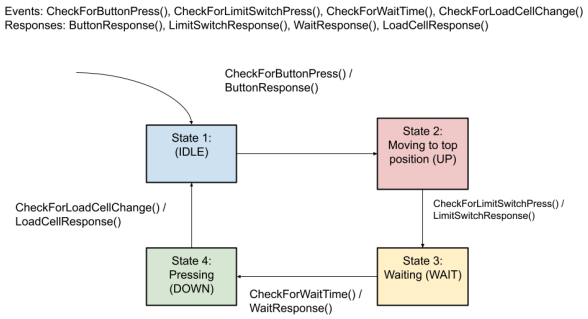


Figure 4: State Transition Diagram (highlighting the 4 states of the machine. Starting in the "IDLE" state, the states flow clockwise, first into the "UP" state, then to the "WAIT" state, then to the "DOWN" state, and finally back to the "IDLE" state.)

Figure 3: Circuit Diagram

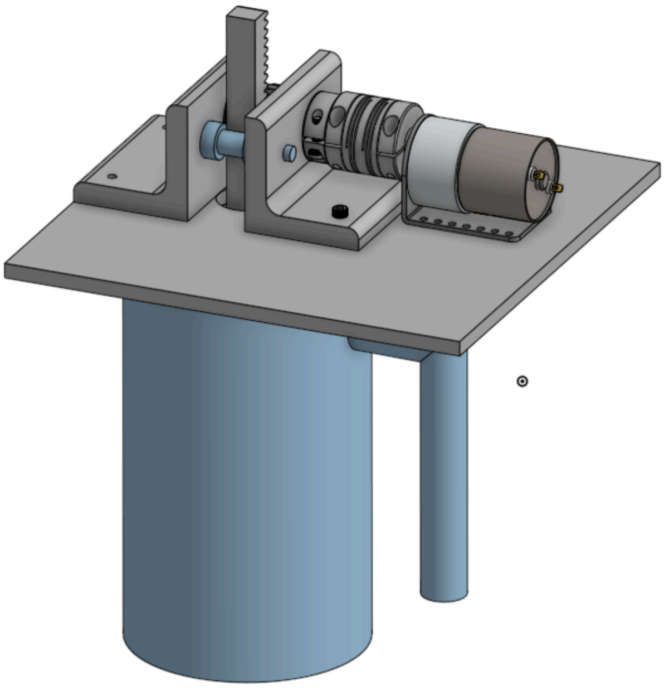
REFLECTION

Our group learned several lessons throughout the process of designing and building this project related to what went well and what we wish we'd done differently. First, we would have implemented the load call at the bottom of the pot, properly using it. Also, we realized how crucial it is to fully constrain the moving parts of the machine. At first, we didn't have the gear rack supported in the back and on the sides so we decided to add the 3D printed roller. However, we didn't make the wheel adjustable leading to the system having some wiggle room. On the other hand, some things that went well include having clean wiring, a robust transmission system, and well written code. This was done through team collaboration, allowing each of us to play to our strengths and contribute in ways that add the most value to the final product.

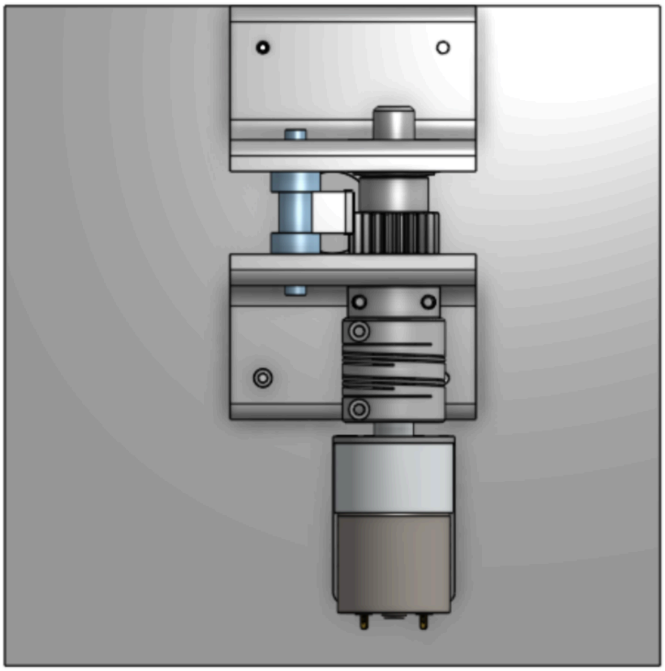
APPENDIX A: BILL OF MATERIALS

Description	Part Number	Unit Price	Quantity	Tax	Total Price
Stainless Steel French Press		$24.99	1	$2.56	$27.55
Inch Rectangular Gear Rack	5174T1	$30.31	1	$3.11	$33.42
Inch Gear	5172T12	$35.24	1	$3.61	$38.85
Cold Rolled Aluminum Sheet 8"x8"x1/4"	9246K11	$21.43	1	$2.20	$23.63
Aluminum Rod	8974K22	$1.51	1	$0.15	$1.66
Rotary Shaft 0.5"	1346K17	$10.81	1	$1.11	$11.92
Shaft Collars 2 Pack For 0.5" Shaft	6436K14	$6.22	2	$1.28	$13.72
Shielded Ball Bearings	60355K291	$8.04	2	$1.65	$17.73
Shims 0.5" 5 pack	97022A132	$6.97	1	$0.71	$7.68
Belleville Washers 12 pack for 0.5" Shaft	9712K74	$10.70	1	$1.10	$11.80
Aluminum 90 Degree Angle 1'x2" 3/8" thick	8982K38	$20.80	1	$2.13	$22.93
Flexible Shaft Coupling		$14.99	1	$1.54	$16.53
6-32 Stainless Steel Screws 5/8" (100 pack)	92196A150	$8.83	1	$0.91	$9.74
6-32 Stainless Steel Nuts (100 pack)	91841A007	$4.33	1	$0.44	$4.77
37D Metal Gearmotor	4752	$0.00	1	$0.00	$0.00
Polou Stamped Aluminum L-Bracket	1084	$0.00	1	$0.00	$0.00
Load Cell		$0.00	1	$0.00	$0.00
ESP32 Feather V2	5400	$0.00	1	$0.00	$0.00
				Total:	$241.92

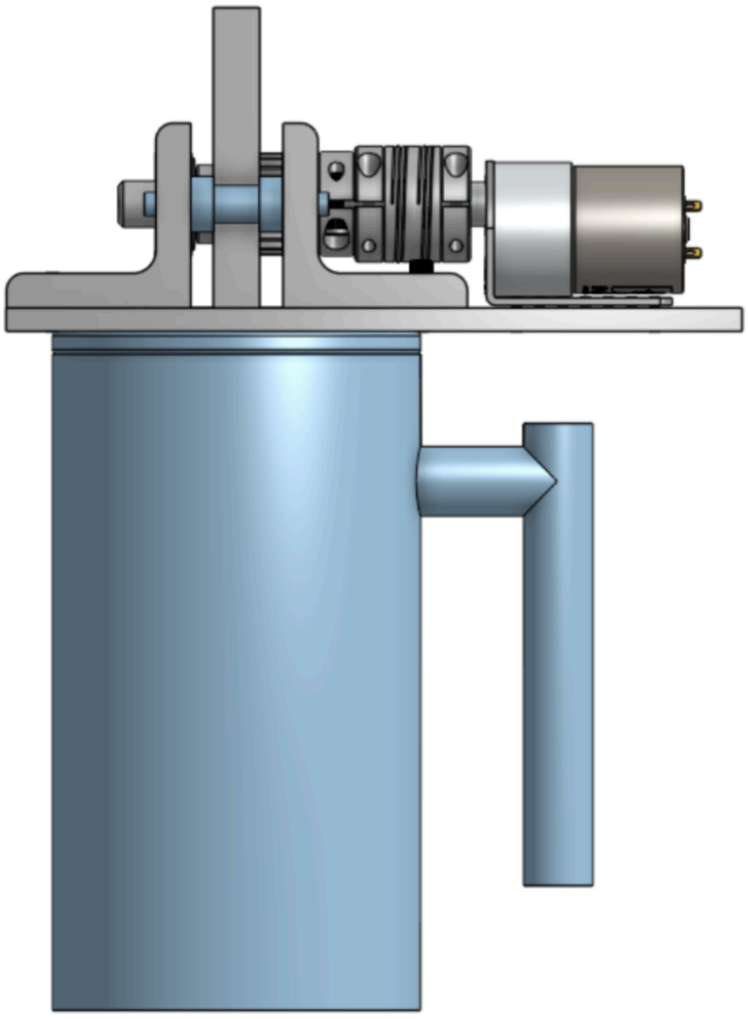
APPENDIX B: CAD



Isometric View



Top View



Side View

APPENDIX C: CODE-ANNOTATED

FinalCode.ino

```
#include <Arduino.h>
#include <ESP32Encoder.h>
#include "HX711.h" //This library can be obtained here http://librarymanager/All#Avia_HX711

#define ENC_1 27 // Motor OUT A
#define ENC_2 33 // Motor OUT B
#define BTN 14  // For the button
#define BTN2 15 // For the limit switch (currently a button, but changing when it comes in)
#define BIN_1 25 // Motor Driver (Purple line) Bin 1
#define BIN_2 26 // Motor Driver (Orange line) Bin 2
#define LOADCELL_DOUT_PIN  12 // Load Cell DT (Orange Line)
#define LOADCELL_SCK_PIN  13 // Load Cell SCK (Purple Line)

ESP32Encoder encoder;
HX711 scale;

enum states {IDLE, UP, WAIT, DOWN};
enum states state = IDLE; // The first state
enum states previousState = (states) - 1;

// Debouncing booleans for main button
volatile bool buttonFlag = false;
volatile bool debouncingFlag = false;
bool isButtonOn = false;

// Debouncing variables for the limit switch
volatile bool limitSwitchFlag = false;
volatile bool limitDebouncingFlag = false;
bool isLimitSwitchOn = false;

// Variable for motor control
const int freq = 5000;
const int resolution = 8;
const int MAX_PWM_VOLTAGE = 255;
const int NOM_PWM_VOLTAGE = 150;

// WAIT start time
unsigned long waitStartTime = 0;
unsigned long waitTime = 5000; // 5 seconds

// Load cell variables
float initialLoadCellValue = 0.0;
```

FinalCode.ino

```
float loadCellThreshold = 50.0; // We need to adjust this based on the sensitivity

// Create the timer for button debouncing
hw_timer_t * timer0 = NULL;
portMUX_TYPE timerMux0 = portMUX_INITIALIZER_UNLOCKED;

// Create the timer for limit switch debouncing
hw_timer_t * timer1 = NULL;
portMUX_TYPE timerMux1 = portMUX_INITIALIZER_UNLOCKED;

// ISR for main button press
void IRAM_ATTR isr() {
  buttonFlag = true;
}

// ISR for limit switch press
void IRAM_ATTR limitSwitchISR() {
  limitSwitchFlag = true;
}

// ISR for timer (debouncing) for main button
void IRAM_ATTR onTime0() {
  portENTER_CRITICAL_ISR(&timerMux0);
  debouncingFlag = false;
  portEXIT_CRITICAL_ISR(&timerMux0);
  timerStop(timer0);
  buttonFlag = false;
}

// ISR for timer (debouncing) for limit switch
void IRAM_ATTR onTime1() {
  portENTER_CRITICAL_ISR(&timerMux1);
  limitDebouncingFlag = false;
  portEXIT_CRITICAL_ISR(&timerMux1);
  timerStop(timer1);
  limitSwitchFlag = false;
}

void setup() {
  Serial.begin(115200); // Baud rate
  pinMode(BTN, INPUT); // Set the BTN as the input (main button)
  attachInterrupt(BTN, isr, RISING); // Now triggers interupt when you click
```

FinalCode.ino

```
85
86    pinMode(BTN2, INPUT); // Limit switch
87    attachInterrupt(BTN2, limitSwitchISR, RISING); // Now triggers interupt when you click
88
89    // Timer setup for button debouncing
90    timer0 = timerBegin(1000000); // 1 MHz
91    timerAttachInterrupt(timer0, &onTime0); // Connects timer to interrupt
92    timerAlarm(timer0, 50000, true, 0); // Sets how long the timer runs for
93    timerStop(timer0); // We just made sure the timer is properly on
94
95    // Timer setup for limit switch debouncing
96    timer1 = timerBegin(1000000); // 1 MHz
97    timerAttachInterrupt(timer1, &onTime1); // Connects timer to interrupt
98    timerAlarm(timer1, 50000, true, 0); // Sets how long the timer runs for
99    timerStop(timer1); // We just made sure the timer is properly on
100
101   // Initialize the encoders
102   ESP32Encoder::useInternalWeakPullResistors = puType::up;
103   encoder.attachHalfQuad(ENC_1, ENC_2);
104   encoder.setCount(0);
105
106   // Attach the motor drivers
107   ledcAttach(BIN_1, freq, resolution);
108   ledcAttach(BIN_2, freq, resolution);
109
110   // Initialize the loac cell
111   scale.begin(LOADCELL_DOUT_PIN, LOADCELL_SCK_PIN);
112   scale.set_scale(-7050.0); // Set the calibration factor
113   scale.tare(); // Resets the scale to 0
114   initialLoadCellValue = scale.get_units(10); // Used for the actual function
115
116   Serial.println("Setup complete, current state: IDLE");
117 }
```

FinalCode.ino

```
119 void loop() {
120   if (state != previousState) {
121     Serial.print("Entered ");
122     switch (state) {
123       case IDLE:
124         Serial.println("IDLE");
125         break;
126       case UP:
127         Serial.println("UP");
128         break;
129       case WAIT:
130         Serial.println("WAIT");
131         break;
132       case DOWN:
133         Serial.println("DOWN");
134         break;
135     }
136   previousState = state; // Update previous state
137 }
138
139 switch (state) {
140   case IDLE:
141     MotorStop(); // Continuously runs (not a service function)
142     if (CheckForButtonPress()) { // EVENT CHECKER: Check that the button has actually been pressed
143       if (!isButtonOn) { // Checks that isButtonOn is false (off) but we pressed it (here for the debouncing)
144         ButtonResponse(); // SERVICE FUNCTION: Actually perform what happens when the button is pressed
145       }
146       isButtonOn = !isButtonOn; // Changed the flag to say that its on
147     }
148     break;
149
150   case UP: // Motor is continuously moving up!
151     MotorUp(); // Continuously runs (not a service function)
152     if (CheckForLimitSwitchPress()) { // EVENT CHECKER: Checks for the limit switch
153       if (!isLimitSwitchOn) { // Checks that isLimitSwitchOn is false (off) but we pressed it (here for the debouncing)
154         LimitSwitchResponse(); // SERVICE FUNCTION: Actually performs what happens when the limit switch is activated
155       }
156       isLimitSwitchOn = !isLimitSwitchOn; // Changed the flag to say its on
157     }
```

Tells us what state we are currently in

State 1: IDLE

State 2: UP

EVENT CHECKER!

SERVICE FUNCTION!

FinalCode.ino

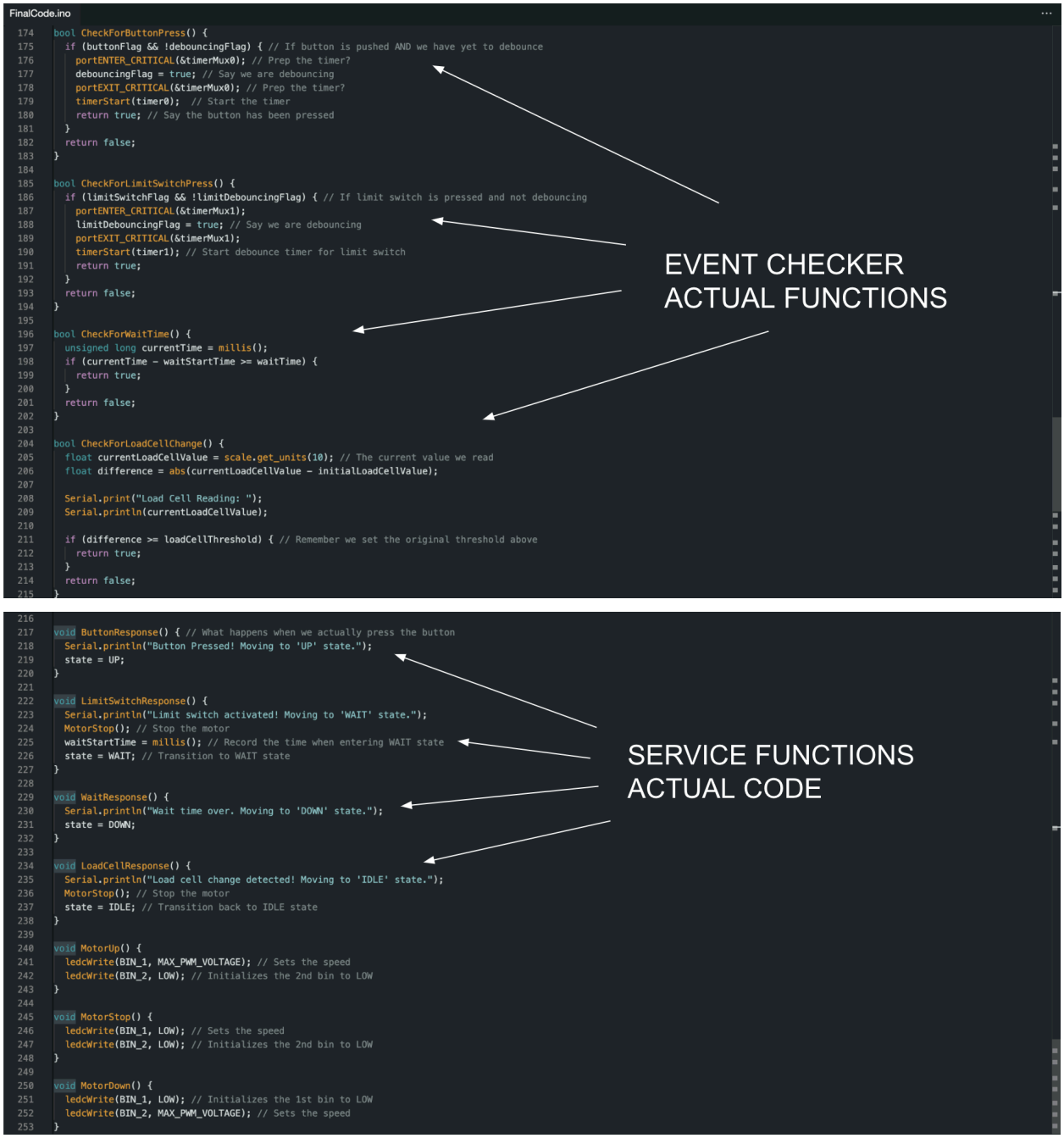
```
158     break;
159
160   case WAIT:
161     MotorStop(); // Continuously runs (not a service function)
162     if (CheckForWaitTime()) { // EVENT CHECKER: Check if the designated number of seconds have passed
163       WaitResponse(); // SERVICE FUNCTION: Transition to the next state in this response
164     }
165     break;
166
167   case DOWN:
168     MotorDown(); // Continuously runs (not a service function)
169     if (CheckForLoadCellChange()) { // EVENT CHECKER: Check if there has been an actual change
170       LoadCellResponse(); // SERVICE FUNCTION: Transitions the state to IDLE
171     }
172     break;
173 }
174 }
```

State 3: WAIT

State 4: DOWN

EVENT CHECKER!

SERVICE FUNCTION!




```
bool CheckForButtonPress() {
  if (buttonFlag && !debouncingFlag) { // If button is pushed AND we have yet to debounce
    portENTER_CRITICAL(&timerMux0); // Prep the timer?
    debouncingFlag = true; // Say we are debouncing
    portEXIT_CRITICAL(&timerMux0); // Prep the timer?
    timerStart(timer0);  // Start the timer
    return true; // Say the button has been pressed
  }
  return false;
}

bool CheckForLimitSwitchPress() {
  if (limitSwitchFlag && !limitDebouncingFlag) { // If limit switch is pressed and not debouncing
    portENTER_CRITICAL(&timerMux1);
    limitDebouncingFlag = true; // Say we are debouncing
    portEXIT_CRITICAL(&timerMux1);
    timerStart(timer1); // Start debounce timer for limit switch
    return true;
  }
  return false;
}

bool CheckForWaitTime() {
  unsigned long currentTime = millis();
  if (currentTime - waitStartTime >= waitTime) {
    return true;
  }
  return false;
}

bool CheckForLoadCellChange() {
  float currentLoadCellValue = scale.get_units(10); // The current value we read
  float difference = abs(currentLoadCellValue - initialLoadCellValue);

  Serial.print("Load Cell Reading: ");
  Serial.println(currentLoadCellValue);

  if (difference >= loadCellThreshold) { // Remember we set the original threshold above
    return true;
  }
  return false;
}

void ButtonResponse() { // What happens when we actually press the button
  Serial.println("Button Pressed! Moving to 'UP' state.");
  state = UP;
}

void LimitSwitchResponse() {
  Serial.println("Limit switch activated! Moving to 'WAIT' state.");
  MotorStop(); // Stop the motor
  waitStartTime = millis(); // Record the time when entering WAIT state
  state = WAIT; // Transition to WAIT state
}

void WaitResponse() {
  Serial.println("Wait time over. Moving to 'DOWN' state.");
  state = DOWN;
}

void LoadCellResponse() {
  Serial.println("Load cell change detected! Moving to 'IDLE' state.");
  MotorStop(); // Stop the motor
  state = IDLE; // Transition back to IDLE state
}

void MotorUp() {
  ledcWrite(BIN_1, MAX_PWM_VOLTAGE); // Sets the speed
  ledcWrite(BIN_2, LOW); // Initializes the 2nd bin to LOW
}

void MotorStop() {
  ledcWrite(BIN_1, LOW); // Sets the speed
  ledcWrite(BIN_2, LOW); // Initializes the 2nd bin to LOW
}

void MotorDown() {
  ledcWrite(BIN_1, LOW); // Initializes the 1st bin to LOW
  ledcWrite(BIN_2, MAX_PWM_VOLTAGE); // Sets the speed
}
```

Code w/ Annotations

APPENDIX D: CODE-UNANNOTATED

For the following code, we added feedback control which is not depicted in the above screenshots (did not affect the state transition diagram or functionality):

```
#include <Arduino.h>

#include <ESP32Encoder.h>

#include "HX711.h" //This library can be obtained here
http://librarymanager/All#Avia_HX711


#define ENC_1 27 // Motor OUT A

#define ENC_2 33 // Motor OUT B

#define BTN 14  // For the button

#define BTN2 15 // For the limit switch (currently a button, but changing when it
comes in)

#define BIN_1 25 // Motor Driver (Purple line) Bin 1

#define BIN_2 26 // Motor Driver (Orange line) Bin 2

#define LOADCELL_DOUT_PIN  12 // Load Cell DT (Orange Line)

#define LOADCELL_SCK_PIN  13 // Load Cell SCK (Purple Line)


HX711 scale;


enum states {IDLE, UP, WAIT, DOWN};

enum states state = IDLE; // The first state

enum states previousState = (states) - 1;


// Debouncing booleans for main button

volatile bool buttonFlag = false;

volatile bool debouncingFlag = false;

bool isButtonOn = false;


// Debouncing variables for the limit switch

volatile bool limitSwitchFlag = false;

volatile bool limitDebouncingFlag = false;

bool isLimitSwitchOn = false;


// Variable for motor control

const int freq = 5000;

const int resolution = 8;
```

```
const int MAX_PWM_VOLTAGE = 1000;
const int NOM_PWM_VOLTAGE = 150;

// motor feedback timer
hw_timer_t* feedback_timer = NULL;
portMUX_TYPE timerMux2 = portMUX_INITIALIZER_UNLOCKED;
volatile bool feedback_timer_fired = false;
int timer_frequency = 1000000; // 1Mhz

// Load cell timer
hw_timer_t* load_cell_timer = NULL;
portMUX_TYPE timerMux3 = portMUX_INITIALIZER_UNLOCKED;
volatile bool load_cell_timer_fired = false;

// motor encoder
ESP32Encoder encoder;
int omegaSpeed = 0;
int omegaDes = 0;
int D = 0;
int Kp = 20;
int Ki = 3;
int IMax = 0;
int sum_err = 0;
int max_sum = 300;
volatile int count = 0;

// WAIT start time
unsigned long waitStartTime = 0;
unsigned long waitTime = 5000; // 5 seconds

// Load cell variables
float initialLoadCellValue = 0.0;
float loadCellThreshold = 1.0; // We need to adjust this based on the sensitivity

// Create the timer for button debouncing
hw_timer_t * timer0 = NULL;
```

```
portMUX_TYPE timerMux0 = portMUX_INITIALIZER_UNLOCKED;

// Create the timer for limit switch debouncing
hw_timer_t * timer1 = NULL;
portMUX_TYPE timerMux1 = portMUX_INITIALIZER_UNLOCKED;

// ISR for main button press
void IRAM_ATTR isr() {
 buttonFlag = true;
}

// ISR for limit switch press
void IRAM_ATTR limitSwitchISR() {
 limitSwitchFlag = true;
}

// ISR for timer (debouncing) for main button
void IRAM_ATTR onTime0() {
 portENTER_CRITICAL_ISR(&timerMux0);
 debouncingFlag = false;
 portEXIT_CRITICAL_ISR(&timerMux0);
 timerStop(timer0);
 buttonFlag = false;
}

// ISR for timer (debouncing) for limit switch
void IRAM_ATTR onTime1() {
 portENTER_CRITICAL_ISR(&timerMux1);
 limitDebouncingFlag = false;
 portEXIT_CRITICAL_ISR(&timerMux1);
 timerStop(timer1);
 limitSwitchFlag = false;
}

void IRAM_ATTR feedback_timer_isr() {
 portENTER_CRITICAL_ISR(&timerMux2);
```

```
 count = encoder.getCount();
 encoder.clearCount();
 feedback_timer_fired = true;
 portEXIT_CRITICAL_ISR(&timerMux2);
}

void IRAM_ATTR load_cell_timer_isr() {
  load_cell_timer_fired = true;
}


void setup() {
 Serial.begin(115200); // Baud rate
 pinMode(BTN, INPUT); // Set the BTN as the input (main button)
 attachInterrupt(BTN, isr, RISING); // Now triggers interupt when you click

 pinMode(BTN2, INPUT); // Limit switch
 attachInterrupt(BTN2, limitSwitchISR, RISING); // Now triggers interupt when you
click

 // Timer setup for button debouncing
 timer0 = timerBegin(1000000); // 1 MHz
 timerAttachInterrupt(timer0, &onTime0); // Connects timer to interrupt
 timerAlarm(timer0, 50000, true, 0); // Sets how long the timer runs for
 timerStop(timer0); // We just made sure the timer is properly on

 // Timer setup for limit switch debouncing
 timer1 = timerBegin(1000000); // 1 MHz
 timerAttachInterrupt(timer1, &onTime1); // Connects timer to interrupt
 timerAlarm(timer1, 50000, true, 0); // Sets how long the timer runs for
 timerStop(timer1); // We just made sure the timer is properly on

 // Initialize the encoders
 ESP32Encoder::useInternalWeakPullResistors = puType::up;
 encoder.attachHalfQuad(ENC_1, ENC_2);
 encoder.setCount(0);
```

```
  // initialize motor feedback timer
 feedback_timer = timerBegin(timer_frequency);     // Set timer frequency to 1Mhz
 timerAttachInterrupt(feedback_timer, &feedback_timer_isr);
 timerAlarm(feedback_timer, 10000, true, 0);      // 10000 * 1 us = 10 ms, autoreload
true


 // initialize load cell timer
 load_cell_timer = timerBegin(timer_frequency);     // Set timer frequency to 1Mhz
 timerAttachInterrupt(load_cell_timer, &load_cell_timer_isr);
 timerAlarm(load_cell_timer, 1000000, true, 0);

 // Attach the motor drivers
 ledcAttach(BIN_1, freq, resolution);
 ledcAttach(BIN_2, freq, resolution);

 // Initialize the loac cell
 scale.begin(LOADCELL_DOUT_PIN, LOADCELL_SCK_PIN);
 scale.set_scale(-7050.0); // Set the calibration factor
 scale.tare(); // Resets the scale to 0
 initialLoadCellValue = scale.get_units(1); // Used for the actual function

 Serial.println("Setup complete, current state: IDLE");
}

void loop() {
   if (state != previousState) {
     Serial.print("Entered ");
     switch (state) {
       case IDLE:
         Serial.println("IDLE");
         break;
       case UP:
         Serial.println("UP");
         break;
       case WAIT:
         Serial.println("WAIT");
         break;
```

```
      case DOWN:
        Serial.println("DOWN");
        break;
    }
  previousState = state; // Update previous state
}

switch (state) {
  case IDLE:
    MotorStop(); // Continuously runs (not a service function)
    if (CheckForButtonPress()) { // EVENT CHECKER: Check that the button has actually been pressed
      if (!isButtonOn) { // Checks that isButtonOn is false (off) but we pressed it (here for the debouncing)
        ButtonResponse(); // SERVICE FUNCTION: Actually perform what happens when the button is pressed
      }
      isButtonOn = !isButtonOn; // Changed the flag to say that its on
    }
    break;

  case UP: // Motor is continuously moving up!
    MotorUp(); // Continuously runs (not a service function)
    if (CheckForLimitSwitchPress()) { // EVENT CHECKER: Checks for the limit switch
      if (!isLimitSwitchOn) { // Checks that isLimitSwitchOn is false (off) but we pressed it (here for the debouncing)
        LimitSwitchResponse(); // SERVICE FUNCTION: Actually performs what happens when the limit switch is activated
      }
      isLimitSwitchOn = !isLimitSwitchOn; // Changed the flag to say its on
    }
    break;

  case WAIT:
    MotorStop(); // Continuously runs (not a service function)
    if (CheckForWaitTime()) { // EVENT CHECKER: Check if the designated number of seconds have passed
      WaitResponse(); // SERVICE FUNCTION: Transition to the next state in this response
```

```
      }
      break;

    case DOWN:
      MotorDown(); // Continuously runs (not a service function)
      if (load_cell_timer_fired) {
        load_cell_timer_fired = false;
        if (CheckForLoadCellChange()) { // EVENT CHECKER: Check if there has been an
actual change
          LoadCellResponse(); // SERVICE FUNCTION: Transitions the state to IDLE
        }
      }

      break;
  }
}

bool CheckForButtonPress() {
  if (buttonFlag && !debouncingFlag) { // If button is pushed AND we have yet to
debounce
    portENTER_CRITICAL(&timerMux0); // Prep the timer?
    debouncingFlag = true; // Say we are debouncing
    portEXIT_CRITICAL(&timerMux0); // Prep the timer?
    timerStart(timer0);  // Start the timer
    return true; // Say the button has been pressed
  }
  return false;
}

bool CheckForLimitSwitchPress() {
  if (limitSwitchFlag && !limitDebouncingFlag) { // If limit switch is pressed and not
debouncing
    portENTER_CRITICAL(&timerMux1);
    limitDebouncingFlag = true; // Say we are debouncing
    portEXIT_CRITICAL(&timerMux1);
    timerStart(timer1); // Start debounce timer for limit switch
    return true;
```

```
 }
 return false;
}

bool CheckForWaitTime() {
 unsigned long currentTime = millis();
 if (currentTime - waitStartTime >= waitTime) {
   return true;
 }
 return false;
}

bool CheckForLoadCellChange() {
 if (load_cell_timer_fired != 0) {
   load_cell_timer_fired = false;
   float currentLoadCellValue = scale.get_units(1); // The current value we read
   float difference = abs(currentLoadCellValue - initialLoadCellValue);

   Serial.print("Load Cell Reading: ");
   Serial.println(currentLoadCellValue);

   if (difference >= loadCellThreshold) { // Remember we set the original threshold
above
     return true;
   }
   return false;
 }
}

void ButtonResponse() { // What happens when we actually press the button
 Serial.println("Button Pressed! Moving to 'UP' state.");
 state = UP;
 sum_err = 0;
}

void LimitSwitchResponse() {
 Serial.println("Limit switch activated! Moving to 'WAIT' state.");
```

```
 waitStartTime = millis(); // Record the time when entering WAIT state
 state = WAIT; // Transition to WAIT state
}

void WaitResponse() {
 Serial.println("Wait time over. Moving to 'DOWN' state.");
 state = DOWN;
 sum_err = 0;
 initialLoadCellValue = scale.get_units(1);
}

void LoadCellResponse() {
 Serial.println("Load cell change detected! Moving to 'IDLE' state.");
 state = IDLE; // Transition back to IDLE state
}

void MotorUp() {
 omegaDes = 2;
 if (feedback_timer_fired) {
   portENTER_CRITICAL_ISR(&timerMux2);
   feedback_timer_fired = false;
   portEXIT_CRITICAL_ISR(&timerMux2);
   motor_feedback();
 }
}

void MotorStop() {
 ledcWrite(BIN_1, LOW); // Sets the speed
 ledcWrite(BIN_2, LOW); // Initializes the 2nd bin to LOW
}

void MotorDown() {
 omegaDes = -2;
 if (feedback_timer_fired) {
   portENTER_CRITICAL_ISR(&timerMux2);
   feedback_timer_fired = false;
```

```
    portEXIT_CRITICAL_ISR(&timerMux2);
    motor_feedback();
  }
}

void motor_feedback() {
 omegaSpeed = count;
 int err = omegaDes-count;
 sum_err += err;
 if (sum_err > max_sum){
    sum_err = max_sum;
 }else if (sum_err < -max_sum){
    sum_err = -max_sum;
 }
 D = Kp*err + Ki*sum_err;
 if (D > MAX_PWM_VOLTAGE) {
    D = MAX_PWM_VOLTAGE;
 } else if (D < -MAX_PWM_VOLTAGE) {
    D = -MAX_PWM_VOLTAGE;
 }
 if (D > 0) {
    ledcWrite(BIN_1, LOW);
    ledcWrite(BIN_2, D);
 } else if (D < 0) {
    ledcWrite(BIN_2, LOW);
    ledcWrite(BIN_1, -D);
 } else {
    ledcWrite(BIN_2, LOW);
    ledcWrite(BIN_1, LOW);
 }
 // Serial.println(D);
}
```