Automated Rice Cooking System

Oliver Chang, Kelly Tran, Evan Hwang

Professor Anwar ME 100 — December 17, 2025

1 Problem Statement

The issue we want to address is the inconvenience of washing and cooking rice at times that conflict with busy schedules. As college students, we often have hectic days that keep us on campus longer than expected. After a long day of work or school, the ideal scenario is to come home to a fresh, warm bowl of rice without the delay of preparation. It would be convenient to schedule rice washing and cooking remotely without needing to rely on roommates.

1.1 Proposed Solution

Our innovative approach is a remote-controlled, integrated system that attaches to a standard rice cooker. This system will not only cook rice on command but will also wash and portion it automatically. Our solution improves upon existing methods by introducing a complete, remote-controlled preparation cycle from dispensing and washing to cooking.

Unlike simple timers that require pre-loading rice and water and offer no flexibility, our design allows for real-time adjustments. If plans change and dinner needs to be pushed back, or a guest comes over and portion sizes need to increase, these changes can be made instantly via an IoT connection. Furthermore, our system is designed to be an affordable and customizable add-on to existing budget rice cookers, making it adaptable to various models on the market. It can also be configured to store and dispense multiple types of rice, offering a level of versatility not found in current smart cookers.

1.2 Existing Market Solutions

The primary existing solutions are smart rice cookers that include timer settings. However, they have two significant drawbacks:

- **Cost:** Smart rice cookers are often expensive, exceeding hundreds of dollars.
- **Functionality:** They do not wash the rice; the user must still wash and portion the rice and water in the pot hours before it is scheduled to cook.

The alternative is the manual process of washing and cooking through boiling or steaming, which consumes valuable time.

2 System Design & Manufacturing

Following Lab 5 where we first conceptualized the functionality, we translated the designs into 2D architecture to appoint spaceclaims before realizing the project in 3D through Onshape.

2.1 Brainstorming

Our initial brainstorming focused on three critical subsystems (See Fig. 1):

- **Storage:** We selected a gravity-fed servo valve system for its simplicity and reduced risk of rice jamming compared to augers.
- **Washing Basin:** We decided to go with a design that rotates the entire washing mechanism and then dumps it into the rice cooker.
- **Actuation:** We chose high-torque servos (MG996 series) for the dumping mechanism because absolute positioning was critical for reliable pouring.

2.2 Mechanical Design (CAD)

The mechanical assembly was modeled in Onshape. Due to the custom geometry of the rice cooker, we used Creaform 3D scanners to create a mesh model (See Fig. 5). Following the scans, we made a 2D representation to avoid collisions (See Fig. 6). We designed for rapid prototyping utilizing lasercut plywood and 3D printed parts.
The full CAD model is available at:
`https://cad.onshape.com/documents/7d561a77970c132a0a7e45c3/w/a0e0040e268b29b3adfe7a8e/e/25e1fd5bd99ddde88d8ab5e9`

2.3 Harnessing

We made sure that we had enough pins for the project and made sure that we had consistent pin numbering throughout planning the harnessing ahead of time. (See Fig. 8).

3 Sensors

3.1 Load Cell & HX711

We implemented a **Strain Gauge Load Cell** coupled with an **HX711**. We opted to use the load cell as an emergency sensor; if no rice was being dispensed, the user would know.

3.2 Ultrasonic Sensor

We mounted an HC-SR04 ultrasonic sensor to measure the remaining rice levels. Through trial and error, we were able to find the threshold at which there is not enough rice remaining.

3.3 Optical Water Level Sensor

We used an optical infrared sensor to look at water levels and alert the user when water needs to be refilled to prevent dry-pumping.

4 Actuation

4.1 Motors

- **Servos:** 1) Dispenser Valve, 2) Dumper Mechanism (175° to 30°), 3) Lid Closer, 4) Cook Switch.
- **DC Motors:** A peristaltic pump handles water, while a high-torque DC motor powers the stirring blade.

4.2 User Interface (Controller)

- **OLED Display:** Real-time feedback ("COOKING...", "Rice Level: OK").
- **LEDs:** Green=1 Cup, Yellow=2 Cups, Red=Reset.
- **Buttons:** Input commands.

5 IoT Communication

We selected **ESP-NOW** for connectivity.

- **Justification:** Using the same techniques from Lab 7, we were able to allow for communication between the two ESP32, allowing us to send input commands and receive sensor inputs.
- **Functionality:** The Controller transmits command packets ("1 Cup", "2 Cups", "Reset") which are instantly received by the Machine ESP32.

6 Software Architecture

The codebase is divided into two firmware packages: **Machine** and **Controller**.

6.1 Modular Drivers & Safety

We abstracted hardware complexity into independent functional blocks (e.g., `dispense_rice()`). Complex sensors like the HX711 were encapsulated into Python classes.

- **Non-Blocking:** We utilized `espnow.irecv(0)` to check for messages without pausing execution. This ensures the system reacts to "Reset" commands immediately, even mid-operation.

The software enforces a strict order of operations:

$$Idle \rightarrow Dispense \rightarrow Wash/Mix \rightarrow Dump \rightarrow Cook \rightarrow Return$$

6.2 Wireless State Synchronization

The Controller sends abstract commands ("1", "2"). The Machine interprets these and broadcasts sensor data back (e.g., "Rice Level: 20cm"), updating the User OLED to display the relevant information to the user.

7 Conclusion

Our Automated Rice Cooking System successfully addresses the problem of time-consuming meal preparation. By integrating precise sensors, robust actuation, and reliable IoT communication, we created a functional prototype that automates the entire rice preparation process.

8 Project Deliverables

Project Video Folder:
`https://drive.google.com/drive/folders/1HbZ1ozAW2ieitAs9Ttc_Pyk7KUJY9Kqq?usp=sharing`

Presentation Slides:
`https://docs.google.com/presentation/d/1H4peqFkKtgU_ZcLLDiIdpKTlRQ9P9tLvqbBPX2V0dTo/edit?usp=sharing`

Overleaf Document:
`https://www.overleaf.com/read/hcfkxhvmzmhq#c7f73f`

Appendix A: Figures

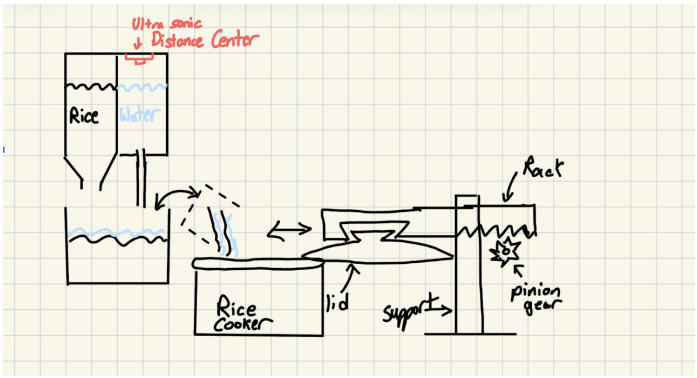

Figure 1: Gravity-fed silo.

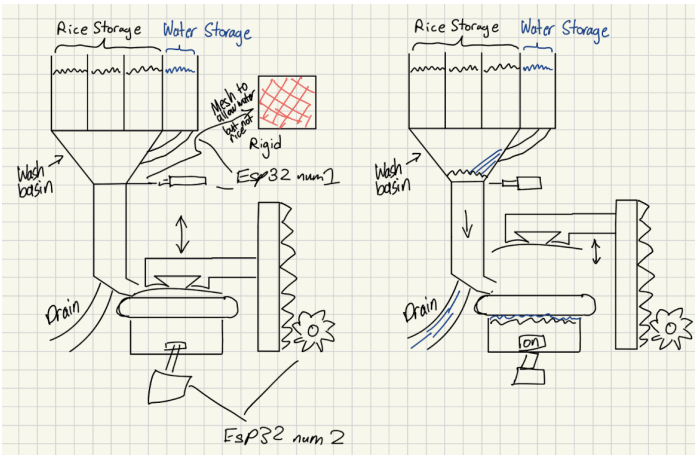

Figure 2: Washing mech.

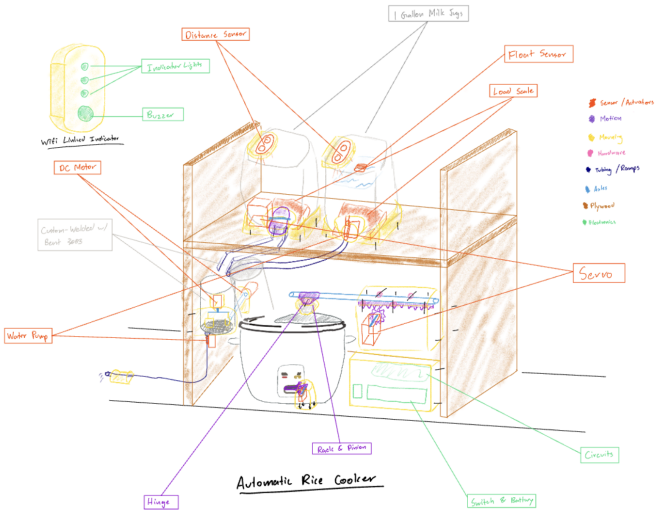

Figure 3: Annotated Design Sketch

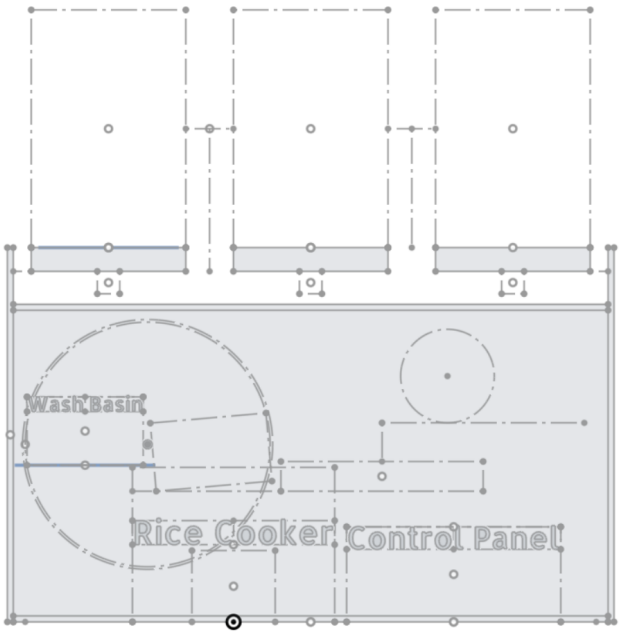

Figure 4: 2D CAD Model

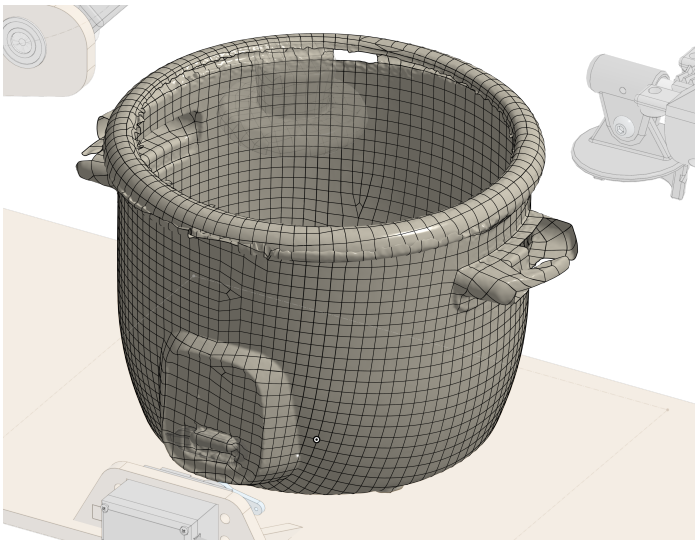

Figure 5: Mesh Scan.

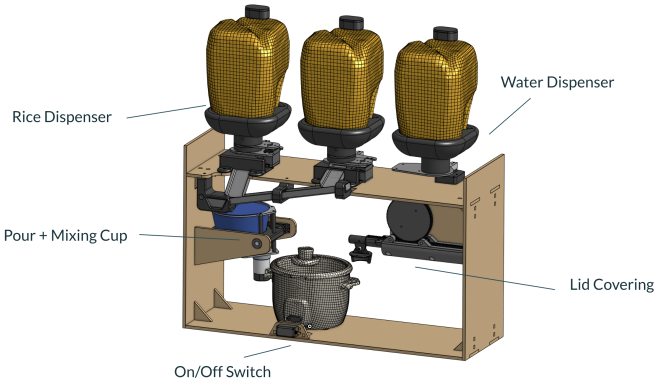

Figure 6: 3D CAD.

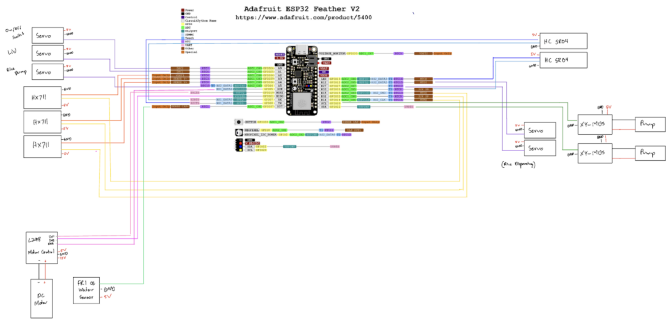

Figure 7: Controller Connections.

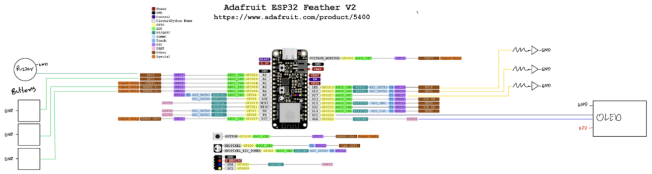

Figure 8: Actuator Drivers.

Figure 9: Team Photo.

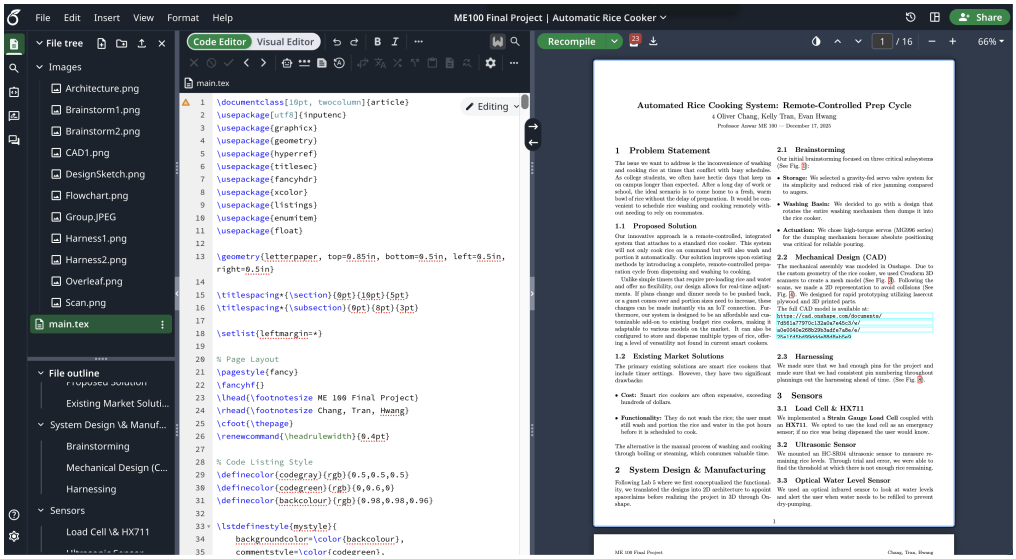

Figure 10: Compiler.

Appendix B: Code

Machine Code

```
i#ME100 Final Project Master Code
#Oliver Chang, Kelly Tran, Evan Hwang
#12-7-2025
#Imports
import time
from time import sleep
from hcsr04 import HCSR04
from machine import Pin, ADC, PWM
```

```
import network
import espnow

class HX711:
    def __init__(self, dout, pd_sck):
        self.dout = Pin(dout, Pin.IN)
        self.pd_sck = Pin(pd_sck, Pin.OUT)
        self.pd_sck.value(0)
        # Wake up
        self.read()

    def is_ready(self):
        return self.dout.value() == 0

    def read(self):
        timeout = time.ticks_ms() + 200
        while not self.is_ready():
            if time.ticks_ms() > timeout:
                raise OSError("HX711 timeout     busy or blocked by other code")

        data = 0
        # Pulse clock 24 times to read bits
        for _ in range(24):
            self.pd_sck.value(1)
            data = data << 1
            self.pd_sck.value(0)
            if self.dout.value():
                data += 1

        # 25th pulse for Gain 128
        self.pd_sck.value(1)
        self.pd_sck.value(0)

        # Convert to signed integer (24-bit 2's complement)
        if data & 0x800000:
            data |= ~0xffffff

        return data

    def read_average(self, times=3):
        total = 0
        for _ in range(times):
            total += self.read()
        return total // times

#Initialize Servos

dispenser_servo = PWM(Pin(33))
dispenser_servo.freq(50)

lid_servo = PWM(Pin(21))
lid_servo.freq(50)

switch_servo = PWM(Pin(26))
switch_servo.freq(50)

dumper_servo = PWM(Pin(4))
dumper_servo.freq(50)

pin_num_motor = 22
motor = Pin(pin_num_motor, Pin.OUT)

pin_num_pump = 20
```

```
pump = Pin(pin_num_pump, Pin.OUT)

# Define the pin number
analog_pin = Pin(39)

# Initialize the ADC
adc = ADC(analog_pin)

# Initialize Load Cell
scale = HX711(dout=34, pd_sck=15) #load cell pins

print("DOUT =", Pin(36, Pin.IN).value())
print("SCK pin set to 0")
s = Pin(15, Pin.OUT)
s.value(0)
print("DOUT now =", Pin(36, Pin.IN).value())

threshold = 1000 #calibration, adjust as needed
#Set Angle Function
def set_angle_dump(angle):
    min_duty = 26
    max_duty = 123
    angle = max(0, min(180, angle))
    duty = int(min_duty + (angle / 180) * (max_duty - min_duty))
    dumper_servo.duty(duty)
    print(f"Angle: {angle}   | Duty: {duty}")



def set_angle_lid(angle):
    min_duty = 26
    max_duty = 123
    angle = max(0, min(180, angle))
    duty = int(min_duty + (angle / 180) * (max_duty - min_duty))
    lid_servo.duty(duty)
    print(f"Angle: {angle}   | Duty: {duty}")

def set_angle_switch(angle):
    min_duty = 26
    max_duty = 123
    angle = max(0, min(180, angle))
    duty = int(min_duty + (angle / 180) * (max_duty - min_duty))
    switch_servo.duty(duty)
    print(f"Angle: {angle}   | Duty: {duty}")



def set_angle_dispenser(angle):
    min_duty = 26
    max_duty = 123
    angle = max(0, min(180, angle))
    duty = int(min_duty + (angle / 180) * (max_duty - min_duty))
    dispenser_servo.duty(duty)
    print(f"Angle: {angle}   | Duty: {duty}")



#Distribution
#Dispensing
def dispense_rice():
        # 1. Small Opening (15 degrees) - For Rice Dispensing 8 Sec = 1 cup of rice
```

```
        print("Open (15 deg)")
        set_angle_dispenser(68)
        time.sleep(8)
        # 2. Closed Position (0 degrees)
        print("Closed")
        set_angle_dispenser(38)


#Load Cell Code
#def load_cell():
#Servo Rice Distribution Code
#def servo_rice_distribution():

#Distance Sensor Code

def distance_sensor():
# ESP32
    class HCSR04:
    # echo_timeout_us is based in chip range limit (400cm)
        def __init__(self, trigger_pin, echo_pin, echo_timeout_us=500*2*30):
            """
            trigger_pin: Output pin to send pulses
             echo_pin: Read only pin to measure the distance. The pin should be protected with
    1k resistor
            echo_timeout_us: Timeout in microseconds to listen to echo pin.
            By default is based in sensor limit range (4m)
            """
            self.echo_timeout_us = echo_timeout_us
            # Init trigger pin (out)
            self.trigger = Pin(trigger_pin, mode=Pin.OUT, pull=None)
            self.trigger.value(0)

            # Init echo pin (in)
            self.echo = Pin(echo_pin, mode=Pin.IN, pull=None)

        def _send_pulse_and_wait(self):
            """
            Send the pulse to trigger and listen on echo pin.
            We use the method `machine.time_pulse_us()` to get the microseconds until the echo
    is received.
            """
            self.trigger.value(0) # Stabilize the sensor
            time.sleep_us(5)
            self.trigger.value(1)
            # Send a 10us pulse.
            time.sleep_us(10)
            self.trigger.value(0)
            try:
                pulse_time = machine.time_pulse_us(self.echo, 1, self.echo_timeout_us)
                return pulse_time
            except OSError as ex:
                if ex.args[0] == 110: # 110 = ETIMEDOUT
                    raise OSError('Out of range')
                raise ex

        def distance_mm(self):
            """
            Get the distance in milimeters without floating point operations.
            """
            pulse_time = self._send_pulse_and_wait()

            # To calculate the distance we get the pulse_time and divide it by 2
            # (the pulse walk the distance twice) and by 29.1 becasue
```

```
            # the sound speed on air (343.2 m/s), that It's equivalent to
            # 0.34320 mm/us that is 1mm each 2.91us
            # pulse_time // 2 // 2.91 -> pulse_time // 5.82 -> pulse_time * 100 // 582
            mm = pulse_time * 100 // 582
            return mm

        def distance_cm(self):
            """
            Get the distance in centimeters with floating point operations.
            It returns a float
            """
            pulse_time = self._send_pulse_and_wait()

            # To calculate the distance we get the pulse_time and divide it by 2
            # (the pulse walk the distance twice) and by 29.1 because
            # the sound speed on air (343.2 m/s), that It's equivalent to
            # 0.034320 cm/us that is 1cm each 29.1us
            cms = (pulse_time / 2) / 29.1
            return cms




#Water Code
#Water Level Sensor Code
def water_sensor():
    print("Starting Analog Read...")
    print("-----------------------")

    # Read the raw 12-bit value (0-4095)
    raw_value = adc.read()

    print(raw_value)

    if raw_value > 1000:
        print('No water')
        return False

    return True

#Pump Code
def pump_out():
    pump.value(1)
    time.sleep(8)
    pump.value(0)
#run for 8 sec -> 1 cup of water

#-----------------------------------------
#Mixing
#Motor Code
def motor_mix():
    motor.value(1)
    time.sleep(8)
    motor.value(0)
#Servo Dumping Code

def servo_dump():
    print('x')
    set_angle_dump(175)
    #set_angle(175, dumper_servo) #sets current angle to -180
    print('x')
```

```
    for angle in range(175, 30, -15): #extends out to 0
        set_angle_dump(angle) #dumper_servo)
        time.sleep(0.2)

    time.sleep(3)

    #spin motor to clean
    motor.value(1)
    time.sleep(2)
    motor.value(0)

    time.sleep(1)
    # Slow return from 30   back to 175
    for angle in range(30, 176, 5):   # 30     175
        set_angle_dump(angle)#, dumper_servo)
        time.sleep(0.05)               # adjust speed here

#-----------------------------------------
#Cooking

#Lid Code

def lid_close():
    set_angle_lid(-180)
    for angle in range(-180, 1, 5): #extends out to 0
        set_angle_lid(angle)
        time.sleep(0.05)
    time.sleep(0.05)
    #set_angle_lid(180)    #goes back to -180


#Servo Cook - Warm Switch Code
#def cook():


def on_off():
    set_angle_switch(10)

#-----------------------------------------

#Refill Code
#ESP32 Communication
#ESP32_1 Sensor Data -> ESP32_2 Refill Board
#Distance Sending Code

button = Pin(7, Pin.IN, Pin.PULL_UP)

sta = network.WLAN(network.STA_IF)
sta.active(True)
sta.disconnect()

sta.config(channel=1)
e = espnow.ESPNow()
e.active(True)

#distance sensing code
# ESP32
#sensor = HCSR04(trigger_pin=7, echo_pin=8, echo_timeout_us=10000)
# ESP8266
#sensor = HCSR04(trigger_pin=12, echo_pin=14, echo_timeout_us=10000)
```

```
peer = b'\x14\x2b\x2f\xaf\xf1\x00'  # MAC address of peer's wifi interface
e.add_peer(peer)

e.send(peer, "Starting...")
print("Sender ready.") #Press button to send message.")

def send_distance_loop():
    distance = sensor.distance_cm()
    message =  "Distance: {:.2f} cm".format(distance)
    print(message)
    e.send(peer, message)  # Send the message to the peer

def reset():
    set_angle_dispenser(38)
    set_angle_switch(25)
    set_angle_dump(175)
    set_angle_lid(180)

    motor.value(0)
    pump.value(0)
    print("RESET")



#-------------------------------------------------------------------------------

#Code Run
#x = 1
#while x == 1:
    #send_distance_loop()
#set_angle_dump(175)
#time.sleep(2)
#Set to warm function


#Initialize all servos to correct starting position
set_angle_dispenser(38)
set_angle_switch(25)
set_angle_dump(175)
set_angle_lid(180)

#Turn off motor and pump
motor.value(0)
pump.value(0)

print('Servos Intialized')
print('--------------------')

print('\n')
print('Checking Rice and Water Levels')
print('-------------------- \n')

sensor = HCSR04(trigger_pin=7, echo_pin=8, echo_timeout_us=10000)

empty_rice_level = 22


distance = sensor.distance_cm()
print(distance)

#Send Refill Messages If Needed to Display on OLED

if distance > empty_rice_level and water_sensor() == False:
```

```
    message =  "out no water"
    print(message)

    e.send(peer, message)


elif distance > empty_rice_level and water_sensor() == True:
    message = "out"
    print(message)

    e.send(peer, message)

elif water_sensor() == False and distance < empty_rice_level:
    print('Refill the Water!')
    message = "no water"
    e.send(peer, message)




pump.value(0)
# Then print message based on value
if distance >= empty_rice_level:
    print("Need to refill rice!") #this needs to be communicated to other esp32

#Only Run if water sensor and rice are full
if water_sensor() == True and distance < empty_rice_level:
    while True:

        print('x')
        host, msg = e.irecv(1000)

        #Read Message and execute corresponding action
        if msg:
            # Convert bytearray to string
            message_str = msg.decode('utf-8')
            print(f"{message_str}")

            #1 Cup of Rice
            if message_str[0] == "1":

                print('1')
                #Read raw average of 5 samples
                pre_dispense = scale.read_average(times=5)

                print(f"Raw: {pre_dispense}")

                time.sleep(0.1)
                print('2')

                dispense_rice()
                time.sleep(3)

                print('3')
                #Check weight before and after dispense function
                post_dispense = scale.read_average(times=5)

                #If Weight is the same break code
                if pre_dispense - post_dispense < threshold:
                    e.send(peer, "stable")
                    break

                time.sleep(1)
```

```
            pump_out()
            time.sleep(1)

            motor_mix()
            time.sleep(1)

            servo_dump()
            time.sleep(1)

            lid_close()
            time.sleep(1)

            on_off()

            #Reset
            #time.sleep(3)

            #set_angle_switch(30)

            time.sleep(3)
            e.send(peer, "done")
            print("im done :tongue_sticking_out_emoji:")

            break


        #2 Cups of Rice
        elif message_str[0] == "2":

            dispense_rice()
            time.sleep(3)

            pump_out()
            time.sleep(1)

            motor_mix()
            time.sleep(1)

            servo_dump()
            time.sleep(1)

            dispense_rice()
            time.sleep(3)

            pump_out()
            time.sleep(1)

            motor_mix()
            time.sleep(1)

            servo_dump()
            time.sleep(1)

            lid_close()
            time.sleep(1)

            on_off()

            e.send(peer, "done")

            print("im done :tongue_sticking_out_emoji:")
```

```
                break



            #Reset Function
            elif message_str == "Reset!":
                reset()
                e.send(peer, "reset")
                print("reset :)")
```

Controller Code

```
#ME100 Final Project Master Code
#Oliver Chang, Kelly Tran, Evan Hwang
#12-7-2025

#Imports
from machine import Pin, I2C
import ssd1306
import time
import network
import espnow

sta = network.WLAN(network.STA_IF)
sta.active(True)
sta.disconnect()

sta.config(channel=1)

e = espnow.ESPNow()
e.active(True)

peer = b'\xf4\x65\x0b\x33\x38\xd4'  # MAC address of peer's wifi interface
e.add_peer(peer)

e.send(peer, "Starting...")
print("Sender ready. Press button to send message.")

# HARDWARE CONFIGURATION
# LEDs
led_red    = Pin(27, Pin.OUT)
led_yellow = Pin(12, Pin.OUT)
led_green  = Pin(13, Pin.OUT)

# Buttons
btn1 = Pin(25, Pin.IN, Pin.PULL_UP)
btn2 = Pin(4,  Pin.IN, Pin.PULL_UP)
btn3 = Pin(5,  Pin.IN, Pin.PULL_UP)

# OLED Display
try:
    i2c = I2C(0, scl=Pin(22), sda=Pin(20), freq=400000)
    oled = ssd1306.SSD1306_I2C(128, 64, i2c)
    HAS_SCREEN = True
except Exception as e:
    print(f"Screen Error: {e}")
    HAS_SCREEN = False

# FUNCTIONS
# LED Functions
def set_led(color, state):
    """
    Control LEDs by name.
```

```
    Usage: set_led('red', 1) or set_led('all', 0)
    """
    state = 1 if state else 0 # Force valid 1/0

    if color == 'red':
        led_red.value(state)
    elif color == 'yellow':
        led_yellow.value(state)
    elif color == 'green':
        led_green.value(state)
    elif color == 'all':
        led_red.value(state)
        led_yellow.value(state)
        led_green.value(state)

def toggle_led(color):
    """Switches an LED state (On->Off or Off->On)"""
    if color == 'red':
        led_red.value(not led_red.value())
    elif color == 'yellow':
        led_yellow.value(not led_yellow.value())
    elif color == 'green':
        led_green.value(not led_green.value())

# Button Functions
def read_button(num):
    """
    Returns True if button is pressed, False if released.
    Handles the logic inversion (0=Pressed) automatically.
    Usage: if read_button(1): ...
    """
    if num == 1:
        return not btn1.value()
    elif num == 2:
        return not btn2.value()
    elif num == 3:
        return not btn3.value()
    return False

# Screen Functions
def display_text(line1, line2="", line3=""):
    """
    Clears screen and displays up to 3 lines of text.
    """
    if not HAS_SCREEN:
        print(f"[NO SCREEN]: {line1} | {line2}")
        return

    oled.fill(0) # Clear
    oled.text(line1, 0, 0)
    oled.text(line2, 0, 16)
    oled.text(line3, 0, 32)
    oled.show()

def clear_screen():
    if HAS_SCREEN:
        oled.fill(0)
        oled.show()


# MAIN LOOP

print("Controller Ready.")
```

```
print("Press buttons to dispense rice or reset.")

# Initial Reset
set_led('all', 0)
display_text("SYSTEM READY", "Press Buttons", "1, 2, or Reset")

try:
    while True:
        # Check Inputs
        b1 = read_button(1)
        b2 = read_button(2)
        b3 = read_button(3)

        # Initialize message_str
        message_str = 'we luv ME100 <3'

        host, msg = e.irecv(1000)
        if msg:
            # Convert bytearray to string
            message_str = msg.decode('utf-8')
            print(f"Received: {message_str}")

        # Logic: b1 -> 1 Cup, b2 -> 2 Cups, b3 -> Reset
        if b1:
            set_led('green', 1)
            display_text("ACTION:", "1 Cup Rice", "Green LED ON")
            message = "1 cup of rice"
            print(message)
            e.send(peer, message)

        if b2:
            set_led('yellow', 1)
            display_text("ACTION:", "2 Cups Rice", "Yellow LED ON")
            message = "2 cups of rice"
            print(message)
            e.send(peer, message)

        if b3:
            set_led('red', 1)
            display_text("ACTION:", "Reset", "Red LED ON")
            message = "Reset!"
            print(message)
            e.send(peer, message)

        if message_str == "reset":
            set_led('all', 0)

        if message_str == "stable":
            display_text("ACTION:", "Rice has stopped", "dispensing")

        if message_str == "done":
            set_led('all', 0)
            break

        # Logic: recieve and output message from controller esp32
        if message_str == "out no water":
            display_text("WARNING:", "Refill rice", "and water")

        if message_str == "out":
            display_text("WARNING:", "Level is low", "Refill rice")

        if message_str == "no water":
            display_text("WARNING:", "Level is low", "Refill water")
```

```
        # Optional: Clear text if nothing is pressed
        if not (b1 or b2 or b3):
            # Just to keep screen from flickering, we only update if needed
            # For this simple test, we leave the last message on screen
            pass

        time.sleep(0.05) # Small delay to debounce/prevent CPU overload

except KeyboardInterrupt:
    set_led('all', 0)
    clear_screen()
    print("Stopped.")
```