Project Luggable Smart Espresso

v1.0.2 Development Plan

Tanay Jha

June 22, 2025

Contents

1 Project Abstract & Core Philosophy

This document outlines the development plan for a luggable, smart espresso machine. The initial goal (v1.0.2) is to engineer a functional proof-of-concept capable of producing espresso by precisely controlling brew parameters. This is not merely a timed or volumetric device; it is an intelligent system designed to execute complex pressure or flow profiles.

Core Philosophy (v1.0.2)

The machine will function as a **real-time interpreter** for brew profiles. It will take a small set of ”keyframe” points defined by the user and dynamically calculate the precise target for any given moment during the extraction. This architecture allows for highly complex and smooth control curves to be defined by a minimal amount of data, demonstrating a sophisticated level of control engineering.

The successful completion of this version will serve as a robust technical demonstration for securing further project funding and will establish a solid software foundation for future hardware upgrades and feature enhancements.

2 Hardware Component Manifest

The v1.0.2 prototype will be constructed using the following well-defined and standard components.

2.1 Control Unit

- **Microcontroller:** Raspberry Pi Pico 2 W (RP2350-based)
- **Role:** Serves as the central brain of the machine, responsible for running all software, reading sensor data, performing PID calculations, and controlling all hardware peripherals.

2.2 Actuation System

- **Motor:** NEMA 17 Stepper Motor (0.42 Nm or similar)
- **Driver:** TMC2208 or A4988 Stepper Motor Driver
- **Role:** Provides the mechanical force to drive the plunger via a lead screw. For this version, it acts as the actuator within a closed-loop pressure control system, where its speed is modulated by the PID controller.

2.3 Sensing Suite

- **Temperature Sensing:**
  - **Component:** MAX31865 PT100 RTD Amplifier Breakout Board with a PT100 probe.
  - **Role:** Accurately measures the water temperature in the brew chamber.
  - **Interface:** SPI (Serial Peripheral Interface).
- **Pressure Sensing:**
  - **Component:** 1.2MPa (12 Bar) Stainless Steel Pressure Transducer.
  - **Role:** Provides the primary real-time feedback for the closed-loop control system by measuring the brew pressure.
  - **Interface:** Analog output, connected to an Analog-to-Digital Converter (ADC) pin on the Pico 2 W.

2.4 User Interface

- **Display:** 0.96” or 1.3” I2C OLED Display (128x64 pixels).
- **Role:** Provides essential, real-time feedback to the user, including system status, live brew parameters, and a post-shot summary graph.
- **Input:** Tactile Push Buttons for user interaction (e.g., Start, Stop, Select).

2.5 Power System

- **Primary Power:** 12V DC Power Adapter.
- **Role:** Provides stable power for the stepper motor and initial heater tests. The battery power system is deferred to a later version.

3 System Architecture Overview

The system operates as a closed-loop feedback controller. The architecture is designed to be modular and event-driven, orchestrated by a central state machine.

1. **Profile Definition:** The user defines a desired brew curve (e.g., target pressure over time) in a simple JSON file stored on the Pico’s flash memory.
2. **State Management:** The main program operates as a state machine (`IDLE`, `HEATING`, `BREWING`, etc.). User button presses trigger transitions between states.
3. **Real-Time Control Loop (During Brew):**
   (a) At each time step (e.g., every 100ms), the software dynamically calculates the precise **target pressure** for that instant based on the user’s profile.
   (b) It reads the **actual pressure** from the pressure transducer.
   (c) A PID (Proportional-Integral-Derivative) control algorithm compares the target pressure to the actual pressure and calculates an error.
   (d) This error is converted into a speed adjustment command for the stepper motor. If pressure is too low, the motor speeds up; if too high, it slows down.
4. **Feedback:** Throughout the process, the OLED display provides live data. After the shot, a summary graph visually confirms the performance of the system against the desired profile.

4 Software Architecture & Module Breakdown

The MicroPython codebase will be organized into logical modules to ensure clarity, maintainability, and separation of concerns. The following sections provide a detailed breakdown of each module's responsibilities and internal logic.

4.1 Profile Data Structure (`profile.json`)

This file acts as the recipe for a single shot, defining all key parameters. Its keyframe-based structure allows for complex profiles to be defined concisely.

```
{
  "profile_name": "Classic 9 Bar",
  "control_variable": "pressure",
  "total_duration_ms": 30000,
  "time_step_ms": 100,
  "target_temp_c": 93.0,
  "profile_points": [
    // Format: [Time_ms, Target_Value, "Interpolation_Type"]
    [0,     0.0, "linear"],
    [8000,  3.0, "hold"],
    [10000, 9.0, "linear"],
    [30000, 9.0, "none"]
  ]
}
```

Listing 1: Example 'profile.json' structure

- **`control_variable`:** Determines if the 'target' values refer to "pressure" (Bar) or "flow" (ml/s).
- **`time_step_ms`:** Defines the frequency of the main control loop (100ms = 10Hz).
- **`profile_points`:** An array of keyframes. Each keyframe is a list containing timestamp, target value, and the interpolation method to use *from this point forward*.

4.2 Module Descriptions

4.2.1 `main.py` (The Conductor)

- **Responsibilities:** The top-level application entry point. Manages the overall machine state, orchestrates module initialization, and handles user input that causes state transitions. It contains no real-time control logic itself, delegating that to the `control` module.
- **Defined States:** `INIT`, `IDLE`, `HEATING`, `BREWING`, `POST_BREW`.
- **Logic Flow:**
  1. **Initialization:** Instantiate hardware objects (`motor_driver`, `display_controller`, `sensors_interface`). Display a splash screen. Perform a one-time motor homing sequence by calling `motor_driver.home()`. Load the default profile via `profile_handler.load_profile("default.json")`. Set initial state to `IDLE`.
  2. **Main Loop:** An infinite loop that polls for state changes and button presses.
     - **In `IDLE` state:** Display idle screen. Poll "Start" button. On press, change state to `HEATING`.
     - **In `HEATING` state:** A simplified state for v1.0.2. It will immediately transition to `BREWING` upon a second "Confirm" press.
     - **In `BREWING` state:** This is the critical hand-off. It calls `control.execute_brew()` and passes the loaded profile and hardware objects. This is a **blocking call**; `main.py` will wait here for the entire duration of the shot.

- Upon return from `execute_brew()`, it receives the `shot_log` data and transitions to `POST_BREW` .
- **In `POST_BREW` state:** Pass the `shot_log` and 'profile' data to the display module to render the summary graph. Wait for any button press to transition back to `IDLE`.

4.2.2 `control.py` (The Real-Time Interpreter)

- **Responsibilities:** To execute the high-frequency, time-critical brew cycle with precision. This module is the "smart" core of the machine.
- **Key Function:** `execute_brew(profile, hardware_objects)`
- **Logic Flow:**

  1. **Setup Phase:**
     - Create an instance of the 'PID' class from `pid.py`.
     - Extract key parameters from the 'profile' dictionary (`time_step_ms`, `total_duration_ms`).
     - Initialize an empty `shot_log` list.
     - Record the `start_time_ms`. Calculate the `next_tick_ms` for the first loop iteration.
  2. **Real-Time Loop:** Begins and runs as long as the elapsed time is less than `total_duration_ms`.
     - **Get Target:** Call `profile_handler.get_target_at_time()` for the current moment.
     - **Get Measurement:** Call `hardware_objects.sensors.read_pressure()`.
     - **Calculate:** Update the PID controller: `pid_output = pid.update(target, measurement)`.
     - **Actuate:** Convert the PID output into a new motor speed and command the motor via `hardware_objects .motor.set_speed()`.
     - **Log:** Store `(time, target_pressure, actual_pressure)` to the `shot_log`.
     - **Timing Control (Critical):** Enter a tight wait loop (`while time.ticks_ms() < next_tick_ms : pass`) to ensure the next iteration starts at the exact, scheduled tick. This enforces the loop frequency and prevents jitter. Update `next_tick_ms` for the subsequent loop.
  3. **Cleanup Phase:** Call `hardware_objects.motor.stop()` and return the populated `shot_log`.

4.2.3 `profile_handler.py` (The Math Engine)

- **Responsibilities:** A stateless library for all logic related to parsing and interpreting profile files. It contains no hardware control code and is thus easily testable.
- **Key Function:** `get_target_at_time(profile_points, current_time_ms)`
- **Logic Flow:**

  1. Searches the `profile_points` list to find the segment defined by two consecutive points, `p_a` and `p_b`, that bracket the `current_time_ms`.
  2. Extracts the `interpolation_method` string (e.g., "linear") from `p_a`.
  3. Based on the method string, calls the appropriate internal calculation function (e.g., `_calculate_linear ()`).
  4. The calculation function returns a single floating-point number, which is the final return value. For the sprint, only "linear" and "hold" will be implemented.

4.2.4 `pid.py` (The Controller Class)

- **Responsibilities:** A simple, reusable PID controller class.
- **Key Function:** `update(current_value)`
- **Logic Flow:**
  1. Calculates the error (`setpoint - current_value`).
  2. Calculates the Proportional term (`P * error`).
  3. Calculates the Integral term (`I * integral_sum`) and updates the integral sum, with anti-windup.
  4. Calculates the Derivative term (`D * (error - last_error)`).
  5. Returns the sum of the P, I, and D terms.

4.2.5 Hardware Driver Modules (`/hardware/`)

- **`motor.py`:**
  - **Responsibilities:** Abstracts the stepper driver.
  - **Key Functions:** `__init__()`, `set_speed(speed)`, `home()`. The `set_speed` function is responsible for calculating the correct pulse frequency and using a PIO state machine for precise, non-blocking pulse generation.
- **`sensors.py`:**
  - **Responsibilities:** Consolidates all sensor reading logic.
  - **Key Functions:** `read_pressure()` handles reading a raw 12-bit integer from the ADC and applying the linear transformation to convert it to Bar. `read_temperature()` handles the SPI communication with the MAX31865.
- **`display.py`:**
  - **Responsibilities:** Wraps a standard SSD1306 I2C OLED driver library to provide application-specific screens.
  - **Key Functions:** `show_idle_screen()`, `update_live_stats()`, and `show_summary_graph(log)`. The `update_live_stats()` function must be highly efficient, only redrawing small, specific regions of the screen.