Oil-States Version 1 Documentation

Isaiah Hajabollhassan & Nathan Rago

March 3, 2025

Contents

1 Introduction

The *SlipProduction* project is designed as a robotic control system that integrates computer vision with motor control on a Raspberry Pi. Its purpose is to detect features (such as holes/cork positions) via a camera and perform precise pick-and-place operations using stepper motors. This document describes a prospective version 1 of the software methods, providing a general overview and explanation of each current component and its intended functionality.

2 System Overview

The system consists of two main functional components:

- **Computer Vision Module:** Uses the Picamera2 and OpenCV libraries to capture images and detect target positions.
- **Motor Control Module:** Utilizes the RPi.GPIO library to control stepper motors for picking up and placing objects.

These modules work in tandem within a main loop that manages the overall operational flow:

1. **Homing:** Calibrates position of the motors using limit switches.
2. **Detection:** Waits for a sensor (belt stoppage) signal, captures image, and processes it to detect cork holes.
3. **Actuation:** Uses the detected coordinates to perform pickup and placement actions (with the placement routine pending implementation).

3 Dependencies and Environment Setup

3.1 Required Libraries

- **OpenCV (`cv2`):** For image processing and computer vision operations.
- **NumPy (`numpy`):** For numerical operations and array handling.
- **Picamera2:** For interfacing with the Raspberry Pi camera.
- **RPi.GPIO:** For controlling GPIO pins on the Raspberry Pi.
- **pigpio:** Imported for potential future use in advanced motor control.
- **PROSPECTIVE PWM LIBRARIES**

3.2 Hardware Requirements

- **Raspberry Pi:** Running a compatible Linux distribution.
- **Pi Camera:** Configured with Picamera2 library.
- **Stepper Motors and Drivers:** Connected via GPIO pins.
- **Limit Switches:** For Limit calibration.
- **Belt Sensor:** To detect when the belt is stopped.

4 Software Architecture

- **Initialization and Configuration:** Define pin assignments, motor direction constants, and system status variables.
- **Main Loop (`main`):** Orchestrates the system's operation by alternating between homing, image capture, and subsequent motor actuation.
- **Utility Functions:** Includes helper functions for homing (`home`), performing a pickup (`Pickup`), and capturing/detecting features (`capture_and_detect`).
- **Future Modules:** Placeholder function (`Place`) that will eventually handle object placement based on detected coordinates.

5 Detailed Code Documentation

5.1 Function Descriptions

5.1.1 `main()`

Purpose: Acts as the primary loop controlling the sequence of operations: homing, detection, and pickup/placement routines.

Flow:

1. Continuously loops, checking whether the system is homed.
2. Calls the `home()` function if the system is not yet homed.
3. Once homed, waits for the belt to stop (intended to be a sensor check).
4. Captures an image and processes it via `capture_and_detect()`.
5. Proceeds to pick up and place operations (pickup function is partially implemented; place function is a stub).

Known Issues:

- Uses `while(true):` instead of `while(True):` (Python is case-sensitive).
- Inconsistent usage of the `Homed` flag (global vs. local variable).

5.1.2 `home(Homed)`

Purpose: Calibrates the motors by moving them until all limit switches indicate the home position.

Flow:

1. Checks the state of each limit switch.
2. Sets the `Homed` flag to `True` if all limit switches are activated.

Known Issues:

- Uses bitwise operator (`&`) instead of the logical `and` for condition checks.
- Directly compares pin numbers to `1` instead of reading the input state.
- Confusing use of the parameter name `Homed` versus the global variable.

5.1.3 `Pickup()`

Purpose: Controls the stepper motor for picking up a cork by executing a fixed number of steps.

Flow:

1. Checks if the task is completed (variable `task_completed` is referenced but not defined).
2. Iterates through a loop to toggle the step pin with a delay determined by `Pulse_width`.

Known Issues:

- Undefined variables `task_completed` and `steps` require definition or proper parameterization.
- Lacks integration with motor direction control.

5.1.4 `Place()`

Purpose: Intended to move the motor based on coordinates provided by the vision system for object placement.

Status: Currently a stub (implementation pending).

5.1.5 `capture_and_detect()`

Purpose: Captures an image using the Pi Camera, processes the image to detect circles, and calculates distances between detected features.

Flow:

1. Initializes the camera and captures an image.
2. Converts the image from RGB to BGR format for compatibility with OpenCV.
3. Converts the image to grayscale and applies median blur and adaptive thresholding.
4. Uses the Hough Circle Transform to detect circles.
5. If at least four circles are detected:
   - Draws circles and connecting lines on the image.
   - Calculates distances and coordinate differences between circles.
   - Overlays the distances onto the image.
6. Returns a tuple containing distances and coordinate differences.

Error Handling:

- If a `RuntimeError` occurs during camera access, the function prints the error, waits for 0.5 seconds, and retries by recursively calling itself.
- **Note:** Recursion here may lead to a stack overflow if the error persists; consider using a loop with a retry limit.

6 Error Handling and Known Issues

- **GPIO State Checks:** The code incorrectly compares pin numbers to literal values instead of reading the GPIO pin state with `GPIO.input(pin)`.
- **Logical Operators:** Use of bitwise `&` instead of logical `and` in conditions.
- **Variable Inconsistencies:** Confusion between local and global variables (e.g., `Homed` vs. `homed`). Undefined variables (`task_completed`, `steps`) must be defined.

- **Infinite Recursion Risk:** The recursive retry mechanism in `capture_and_detect()` can lead to infinite recursion under persistent failure conditions.

7 Usage and Operation Instructions

1. **Hardware Setup:** Connect the limit switches, stepper motors, belt sensor, and camera to the specified GPIO pins as defined in the code.
2. **Software Setup:**
   - Install the required Python libraries.
   - Configure the Raspberry Pi for camera and GPIO access.
3. **Running the Code:**
   - Ensure that an entry point is defined (e.g., add an `if __name__ == "__main__": main()` block at the end of the file).
   - Execute the script using Python:

     ```
     python SlipProduction.py
     ```
4. **Operation:**
   - The system will begin by homing the motors.
   - Once homed and when the belt is detected as stopped, the camera will capture an image and attempt to detect circles.
   - Detected coordinates and calculated distances are printed for debugging.
   - Pickup and (eventually) placement routines are executed based on these detections.

8 Future Enhancements

- **Complete the `Place()` Function:** Develop and integrate the placement routine using the detected coordinates.
- **Improve GPIO Handling:** Implement proper GPIO state reading (using `GPIO.input()`) and logical conditions.

- **Refine Error Handling:** Replace recursive retries in `capture_and_detect()` with a loop that includes a retry limit to avoid potential stack overflow.
- **Variable and State Management:** Standardize naming conventions and ensure that all referenced variables (e.g., `task_completed`, `steps`) are appropriately defined.
- **Integrate Additional Libraries:** Utilize the imported `pigpio` library if advanced motor control becomes necessary.

9 Virtual PWM Generation on Non-PWM Pins

9.1 Approach Overview

This section explains how to simulate a PWM signal on a non-PWM pin using a software "bit-banging" technique.

- **Bit-Banging Technique:** The method involves continuously toggling a digital pin's state in a loop. The pin is set high for the "on" duration and low for the "off" duration within each PWM cycle, effectively creating a virtual PWM output.
- **Key Considerations:**
  - **Timing Accuracy:** Software-based timing (e.g., using `time.sleep()`) may not provide microsecond-level precision required for high-frequency PWM, which can lead to inaccuracies or jitter.
  - **CPU Utilization:** Continuously running a loop to toggle the pin can result in higher CPU usage, which might be problematic for certain applications.
- **Alternative Methods:**
  - **Software PWM Libraries:** Libraries such as RPi.GPIO offer a built-in PWM class that manages the toggling in a separate thread for improved reliability.
  - **pigpio Library:** The pigpio library provides functions like `set_PWM_dutycycle()` that can generate PWM signals with better timing accuracy and lower CPU overhead.

  - **Hardware PWM:** When available, using dedicated hardware PWM is the most efficient approach, as it offloads the PWM generation to dedicated circuitry.

10 Revision History

- **Version 1.0**:
  - Initial version of the code documentation.
  - Outlines basic architecture, functionality, and known issues.
  - Serves as the foundation for future iterations and improvements.
- **Version 2.0: PWM modulation via Non-PWM pins**
  - Added a new subsection *Virtual PWM Generation on Non-PWM Pins* describing the software-based approach to simulate PWM on non-PWM pins.
  - Documented key considerations regarding timing precision and CPU utilization.
  - Included alternative methods such as using software PWM libraries (e.g., RPi.GPIO, pigpio) and, where available, hardware PWM.

11 References

- OpenCV Documentation
- Picamera2 Documentation
- RPi.GPIO Documentation
- pigpio Library