Intelligent Waste Sorting System

Debugging Document

Project Name: Intelligent Classification Rubbish Bin (SortMate)

Platform: Raspberry Pi 5, YOLOv5, Multithreaded C++ Real-Time Control

Objective: Ensure system real-time performance, stability, and inter-module coordination

Image Capture Time Debugging

- Related Modules:
  - CameraCapture / CameraWorker
  - Timer-based scheduling module
- Debugging Objectives:
  - Measure the total time from camera activation to image file being saved;
  - Verify the accuracy and consistency of the periodic capture cycle (target: one image every two seconds);
  - Ensure all images are correctly saved to the designated directory, with no corruption or loss.
- Key Checks:
  - Capture and save time should remain under 500 ms per image;
  - There should be no missing, incomplete, or unreadable image files;
  - The time interval between consecutive captures should remain stable, ideally within ±200 ms of the 2-second target.
- Practical Analysis and Recommendations

  During testing, we instrumented the CameraWorker::captureImage() method with time stamps to measure the image acquisition and storage duration. Results show that under stable lighting and proper camera connection, the average time for capturing and saving an image is approximately 220–280 ms, well within the acceptable range.

  However, we observed that if too many images accumulate in the /home/pi/images/ directory without being cleared, read/write performance may degrade. This issue becomes more pronounced when image capture and image processing threads run in parallel. To address this, we recommend:

  - Regularly moving or archiving processed images (e.g., to a /processed/ folder);
  - Using timestamp-based filenames to avoid naming collisions;
  - Implementing a timeout or "capture skip" mechanism in the Timer logic to prevent overlapping operations if the previous frame has not finished processing.

  To further evaluate the stability of the capture interval, we suggest logging timestamps of consecutive captures and plotting them as a line chart to identify timing drift or frame drops.

YOLO Processing Time Debugging

- Related Modules:
  - YoloDetector
  - NCNN inference engine pipeline
- Debugging Objectives:
  - Measure the total duration from image loading to YOLO inference completion;
  - Evaluate whether the YOLO model constitutes a bottleneck in real-time performance;
  -Verify that the detection callback is triggered correctly and the results are accurate.
- Key Checks:
  - Inference time should be within 800 ms;
  - The callback is invoked correctly with accurate object counts;
  - If using a lightweight model (e.g., v5lite), assess trade-offs between speed and accuracy.
- Practical Analysis and Recommendations

  To evaluate YOLO performance, we recorded timestamps at the entry and exit points of the YoloDetector::Impl::detect() function. Across multiple test images, we observed an average inference time of approximately 420 ms using the v5lite model, providing relatively smooth system response.

  However, when using higher-resolution images or under high thread concurrency, the inference time can spike to over 700 ms, which may negatively impact the timing of motor operations. In response, we recommend the following:

  For stricter real-time constraints, consider using an even smaller model like v5nano;

  Enable NCNN's Vulkan acceleration or switch to OpenVINO for improved inference speed;

  Normalize input image sizes (e.g., resize to 320×320) to reduce dynamic memory overhead;

  Ensure that the callback function is not blocked by the main thread or downstream processes—consider adding asynchronous protection mechanisms if necessary.

  Additionally, we recommend logging detection metadata, such as the number of objects per frame, label distribution, and confidence scores. This data is helpful for post-deployment analysis of model consistency and accuracy.

Stepper Motor Rotation Time Debugging

- Related Modules:
  - GarbageSorter
  - StepperMotor
- Debugging Objectives:
  - Log the total duration of each motor classification task;
  - Analyze response timing and reset behavior;
  - Ensure the delay-return (managed by the timer) is precise and consistent.

- Key Checks:
  - Motor action should complete within 1.5 seconds;
  - No thread conflicts or motor contention;
  - Detected waste type matches the motor movement.
- Practical Analysis and Recommendations

  The system uses the GarbageSorter::processWaste() function to activate different stepper motors based on the classification label provided by YOLO. In our tests, we instrumented the function with entry and exit timestamps to calculate the time taken for each classification action.

  Test Results:

  1.The average duration for a single motor action ranged between 900–1200 ms, which meets the performance target;

  2.The timer-managed reset mechanism was generally consistent. However, under high-frequency detection or rapid image capture, we observed a race condition where a motor could be reactivated before it had fully returned to its idle state.

  To ensure stable and predictable behavior, we recommend the following improvements:

  1.Sequential Execution Queue: Introduce a motor task queue to ensure classification actions are processed one at a time;

  2.Motor Busy Flags: Implement internal state flags within the StepperMotor class to prevent new actions before the previous one completes;

  3.Timeout-Protected Resets: Use scheduleTask to enforce a maximum safe return time for each motor reset;

  4.GPIO Monitoring: Log or programmatically check the GPIO usage status to detect any hardware-level conflicts or failures.

  Additionally, it is advisable to log every motor classification event with associated details such as object label, motor ID, and execution time. This data can be used to evaluate system load, categorize waste statistics, and further optimize the motor control logic.

Sensor Response Time Debugging

- Related Modules:
  - Phototransistor (light sensor) input (GPIO)
  - System initialization logic in main_sort.cpp
- Debugging Objectives:
  - Measure response time between sensor trigger and system startup;
  - Detect and avoid false positives or missed triggers;
  - Confirm system transitions correctly from "standby" to "active" mode.
- Key Checks:
  - Response time should be within 500 ms;
  - Sensor connection is stable, with minimal jitter;
  - System correctly starts the image capture and detection process after trigger.
- Practical Analysis and Recommendations

The system monitors the GPIO pin connected to the phototransistor to detect nearby waste. Upon detection, the main control thread initiates the capture and classification sequence. In our practical tests, the response time from GPIO signal detection to system activation averaged between 150 ms and 250 ms, which meets performance expectations.

However, several edge cases were observed:

1. Signal bouncing or poor wiring could cause multiple false triggers or unstable states, leading to unexpected system behavior;
2. Ambient lighting conditions, such as strong sunlight, may interfere with sensor accuracy and require shielding or a sensor upgrade;
3. When digitalRead() is polled too infrequently (e.g., >200 ms intervals), short trigger signals may be missed entirely.

To improve reliability and robustness, we recommend:

1. Increase polling frequency: Reduce sleep_for() duration in the main loop from 100 ms to 30–50 ms;
2. Debounce logic: Add a short delay (e.g., 50 ms) after detection to confirm signal stability;
3. Logging mechanism: Log timestamped trigger events and response status to analyze potential misfires or unresponsive behavior;
4. Hardware-level improvements: Use RC filters or switch to digital infrared proximity sensors with built-in noise suppression.

The sensor acts as the entry point to the entire classification workflow. Its responsiveness and reliability directly affect downstream modules. We therefore suggest treating sensor behavior as a critical real-time test item and incorporating its metrics into formal system evaluation.