Py_to_PDF

May 8, 2025

[ ]:
```python
#!/usr/bin/env python3
"""
Main control loop for water-line detection system.

This script initializes hardware (GPIO on Raspberry Pi), manages network
connectivity and ORC client setup, then repeatedly captures image bursts,
processes them for water-line detection, logs performance metrics,
and sleeps between cycles.
"""
import os
import time
import socket
import logging
from datetime import datetime

import psutil  # system and process utilities

# Attempt to import Raspberry Pi GPIO; set flag if available
try:
    import RPi.GPIO as GPIO
    RPI_AVAILABLE = True
except ImportError:
    RPI_AVAILABLE = False

# Project configuration and utilities
from wd__config_cycle import CONFIG  # global configuration dictionary
from wd__modules.wd__utilities_cycle import setup_logging
from wd__modules.wd__base_paths_cycle import get_base_paths
from wd__modules.wd__orc_api_cycle import ORC
from wd__modules.wd__capture_cycle import capture_burst_and_process,␣
 ↪set_orc_instance

# Initialize the logger using our custom setup
logger = setup_logging()

def log_message(msg: str):
    """
```

```
    Wrapper for standardized logging.

    Args:
        msg: Message string to log at INFO level.
    """
    logger.info(msg)


def is_network_available(host: str = "openrivercam.com") -> bool:
    """
    Check DNS resolution to determine if network is reachable.

    Args:
        host: Domain to resolve (default: ORC API host).

    Returns:
        True if DNS resolution succeeds, False otherwise.
    """
    try:
        socket.gethostbyname(host)
        return True
    except socket.gaierror:
        return False


def main():
    """
    Main cycle loop:
    1. Initialize ORC client if network is enabled.
    2. Configure GPIO if on Raspberry Pi.
    3. Build paths for raw and processed images.
    4. Call capture and processing routine.
    5. Log timing and system metrics.
    6. Sleep before next cycle.
    """
    # Control flag for network use; can be toggled in CONFIG
    network_enabled = False

    # psutil Process instance for CPU timing
    process = psutil.Process()

    while True:
        # Start timing for this cycle
        cycle_start_time = time.time()
        start_cpu_times = process.cpu_times()
        start_cpu_time = start_cpu_times.user + start_cpu_times.system
```

```
        log_message("----- Cycle start -----")

        # Read system type and dummy-mode flag from config
        system = CONFIG["system"]
        dummy_mode = CONFIG["dummy_mode"]

        # Initialize ORC client if network is available
        if network_enabled and is_network_available():
            log_message("Network detected. Initializing ORC client.")
            try:
                orc_cfg = CONFIG["orc"]
                orc = ORC(
                    base_url=orc_cfg["base_url"],
                    username=orc_cfg["username"],
                    password=orc_cfg["password"]
                )
            except Exception as e:
                log_message(f"Failed to init ORC: {e}")
                orc = None
        else:
            log_message(
                "Network connectivity disabled or unavailable; skipping ORC␣
↪init."
            )
            orc = None
        # Provide the ORC instance to capture module (for uploads)
        set_orc_instance(orc)

        # GPIO setup for Raspberry Pi: LEDs and status pins
        LED_PIN = 4
        TPL_DONE_PIN = 27
        if RPI_AVAILABLE and system == "raspberry_pi":
            GPIO.setmode(GPIO.BCM)
            GPIO.setup(LED_PIN, GPIO.OUT)
            GPIO.setup(TPL_DONE_PIN, GPIO.OUT)
            # Initialize pins to LOW
            GPIO.output(LED_PIN, GPIO.LOW)
            GPIO.output(TPL_DONE_PIN, GPIO.LOW)
            time.sleep(1)
            log_message("GPIO initialized.")
        else:
            log_message(
                "GPIO not available or not on Raspberry Pi; skipping GPIO setup.
↪"
            )

        # Build file system paths for raw and processed images
```

```
        paths = CONFIG["paths"][system]
        base_paths = get_base_paths(paths["image_path"], paths["output_path"])
        raw_images_folder = os.path.join(base_paths["output_path"],␣
↪"raw_images")
        processed_images_folder = base_paths["output_path"]

        # Retrieve capture parameters from config
        burst_intervals = CONFIG["capture_params"]["burst_intervals"]
        cycle_interval = CONFIG["capture_params"]["cycle_interval"]
        processing_params = CONFIG["processing_params"]

        # Perform capture and processing of image burst
        try:
            capture_burst_and_process(
                raw_folder=raw_images_folder,
                processed_folder=processed_images_folder,
                burst_intervals=burst_intervals,
                cycle_interval=cycle_interval,
                processing_params=processing_params,
                system=system,
                dummy_mode=dummy_mode
            )
            log_message("Waterline detection completed (or dummy mode)." )
        except Exception as e:
            log_message(f"Error in capture/processing: {e}")

        # End timing and compute metrics
        cycle_end_time = time.time()
        runtime = cycle_end_time - cycle_start_time
        end_cpu_times = process.cpu_times()
        end_cpu_time = end_cpu_times.user + end_cpu_times.system
        cpu_time_used = end_cpu_time - start_cpu_time
        cpu_usage = psutil.cpu_percent(interval=0.5)
        mem_usage = psutil.virtual_memory().percent

        # Log cycle performance metrics
        log_message(
            f"Cycle Metrics - Runtime: {runtime:.2f}s, CPU Time: {cpu_time_used:
↪.2f}s, "
            f"CPU%: {cpu_usage}%, Mem%: {mem_usage}%"
        )

        # Sleep for configured rest period before next cycle
        rest_time = CONFIG.get("cycle_rest_seconds", 600)
        log_message(f"Cycle end. Sleeping {rest_time}s before next run.")
        log_message("----- Cycle end -----\n")
        time.sleep(rest_time)
```

```
if __name__ == "__main__":
    main()
```