Technical Module Description: oradio\_logging.py

Version: 1.0 Date: January 17, 2025

Authors: Henk Stevens, Olaf Mastenbroek, Onno Janssen

Organization: Stichting Oradio

License: GNU GPL

# 1. Purpose and Overview

The `oradio\_logging.py` module establishes a robust, thread-safe, and fault-tolerant logging system for the Oradio platform. It captures debug, informational, warning, and error messages in real-time, routing them to both file-based logs and remote monitoring services. The module leverages concurrent-safe log handlers and integrates system-level protection using Python’s `faulthandler` module.

# 2. Key Features

* Concurrent log rotation using `ConcurrentRotatingFileHandler` to support multiple threads and processes.
* Color-coded terminal output to distinguish severity levels.
* Support for structured log messages including filename, line number, and timestamps.
* Remote monitoring integration: forwards warnings and errors to a cloud-based logging service.
* Throttle-aware logging: avoids logging to SD card if the Raspberry Pi is in a throttled power state.
* Background logging queue via `setup\_logging\_queues()` for non-blocking operation.
* Built-in low-level crash tracing using `faulthandler.enable()`.

# 3. Logging Architecture

The logging system is anchored around a logger named `'oradio'`, configured at startup to write to a rotating log file. Color-coded output is applied conditionally for interactive terminals (`sys.stderr.isatty()`), and logging output is routed to background threads to improve real-time system performance.  
  
Log files are stored in a predefined directory (`ORADIO\_LOG\_DIR`) and rotated when exceeding 512KB with two backups retained. The logger can dynamically adjust its verbosity level (DEBUG, INFO, WARNING, ERROR) through test interfaces or configuration changes.

# 4. Internal Components and Classes

## 4.1 ColorFormatter

The `ColorFormatter` class extends the standard Python logging formatter to include ANSI escape codes for colored output in the terminal. Each log level is assigned a specific color for better visibility:  
- DEBUG: Grey  
- INFO: White  
- WARNING: Yellow  
- ERROR: Red  
This class is applied only when outputting to an interactive terminal (TTY).

## 4.2 ThrottledFilter

To prevent potential corruption of the SD card due to excessive writes while the Raspberry Pi is under voltage or temperature throttling, the `ThrottledFilter` suppresses log writes when throttling is active. It queries the hardware throttled state through a utility method and conditionally blocks log events.

## 4.3 RemoteMonitoringHandler

This handler sends all WARNING and ERROR messages to a cloud-based remote monitoring service. It does this by dynamically importing the `rms\_service()` and invoking its `send\_message()` method with relevant metadata. The integration ensures centralized visibility into operational issues in the field.

## 4.4 setup\_logging\_queues()

This function, imported from `concurrent\_log\_handler.queue`, enables asynchronous log handling via a shared logging queue. It ensures logs are processed by a background thread, avoiding contention in multi-threaded or multi-process scenarios.

# 5. Using and Testing the Logging System

To test and verify the behavior of the logging system, the script includes a command-line test interface. When run directly (`python3 oradio\_logging.py`), the user can select test cases to simulate different logging levels, exceptions, and crash events.

* Available test options include:
* • Log messages at different verbosity levels (DEBUG, INFO, WARNING, ERROR)
* • Generate unhandled exceptions in a `Process` or `Thread` to test robustness
* • Trigger segmentation fault to verify low-level crash capture

# 6. Advanced Behavior and Best Practices

## 6.1 Rotating File Handling with Concurrency

The `ConcurrentRotatingFileHandler` from the `concurrent-log-handler` package ensures that multiple threads and processes can write to the same log file without corruption or race conditions. Log rotation is handled automatically based on file size limits (`512 KB`) and maintains up to two backup files to prevent excessive disk usage. The handler is particularly suited for embedded Linux systems like the Raspberry Pi, where performance and SD card reliability are important.

## 6.2 Crash Handling with FaultHandler

The `faulthandler` module is enabled at the start of the script to ensure low-level crashes are captured and printed to standard error. This is critical for diagnosing issues such as segmentation faults caused by native libraries (e.g., `ctypes`, I2C, or audio drivers). The crash logs include stack traces and line numbers, which greatly aid post-mortem debugging.

## 6.3 Logging Level Management and Dynamic Changes

Logging levels can be changed dynamically using the test interface or programmatically by calling `oradio\_log.setLevel()`. This allows fine-grained control of log verbosity based on the runtime context (e.g., development, staging, or production). Careful logging level configuration helps balance insight and performance.

## 6.4 Integration Points

The `oradio\_log` logger is used consistently across all Oradio modules, providing centralized, structured logging. Modules such as `volume\_control.py`, `mpd\_control.py`, and `spotify\_connect\_direct.py` use `oradio\_log.debug()`, `info()`, `warning()`, and `error()` calls to report internal state and failures. Thanks to this centralized setup, Oradio's maintainers can monitor live logs and trace system behavior efficiently.

# 7. Recommendations for Extension

The current logging setup provides a solid foundation for production use. To enhance it further:  
- Add log message filtering based on module name or keyword.  
- Integrate email alerts for critical failures.  
- Sync log output with external log aggregators (e.g., Fluentd, Logstash).  
- Provide user-facing logging through a lightweight web interface or API.