**Oradio wifi\_service Module Documentation**

**1. Overview**

The wifi\_service module is part of the Oradio system architecture, built to manage wireless network connectivity in an embedded or headless environment. It leverages the nmcli Python interface to control NetworkManager on Linux systems and provides seamless transitions between client (infrastructure) mode and access point (AP) mode.

This module is especially useful in devices that need to:

* Automatically join known networks,
* Fall back to AP mode for configuration or rescue,
* Provide feedback to a control process or frontend,
* Handle network transitions robustly.

It was developed to support Oradio Stichting's connected devices and is extendable to any Linux-based IoT or edge computing environment.

**2. Functional Capabilities**

**✔️ Connectivity Management**

The module supports:

* Connecting to a WiFi network with SSID/password credentials.
* Detecting if already connected and skipping redundant operations.
* Disconnecting from current connections.
* Switching between infrastructure mode and access point mode.
* Restoring previous connections after AP mode ends.

**📡 Access Point Mode**

* The device can broadcast its own SSID (OradioAP) as a fallback or setup interface.
* DNS redirection is configured to capture all domain queries and redirect them to a static IP.
* Supports rejoining previously saved networks after exiting AP mode.

**📩 Messaging Architecture**

* Inter-process communication is done via a queue mechanism (multiprocessing.Queue).
* State change messages contain type, state, and any errors encountered.
* Designed for integration with a controller that reacts to connectivity state.

**🔍 Scanning and State Inspection**

* The system can scan for available WiFi networks.
* Can list known configurations in NetworkManager.
* Detects and reports whether it’s connected to the internet, a local network, or in AP mode.

**3. Class Architecture**

**class wifi\_service**

This is the central class responsible for all WiFi operations.

**Initialization**

wifi = wifi\_service(queue)

* queue: A multiprocessing queue to report connection status or errors.

**Core Methods**

| **Method** | **Purpose** |
| --- | --- |
| wifi\_connect(ssid, password) | Connects to a WiFi network or sets up an AP. |
| wifi\_disconnect() | Disconnects and removes the current connection. |
| access\_point\_start(force\_ap) | Starts the access point, optionally overriding current connection. |
| access\_point\_stop() | Stops the access point and restores previous connection if applicable. |
| get\_wifi\_networks() | Returns list of available WiFi networks (excludes own AP). |
| get\_wifi\_connection() | Returns the currently active connection, or None. |
| get\_wifi\_nm\_connections() | Lists all WiFi connections defined in NetworkManager. |
| get\_state() | Determines WiFi state (idle, internet, no internet, access point). |

**4. Error Handling & Logging**

The module makes use of the oradio\_logging facility to output detailed logs for:

* Connection attempts and failures
* AP setup and teardown
* DNS redirection commands
* State transitions

Errors are captured in self.error and reported in the message queue with a standardized message format:

{"type": MESSAGE\_WIFI\_TYPE, "state": STATE\_X, "error": ERROR\_X}

This ensures that the controller has a clear view of what the module is doing at all times.

**5. Integration Notes**

* The class is not standalone and expects to be integrated into a larger multiprocessing system.
* The messaging queue is vital for notifying other components.
* The script provides a standalone CLI/test mode, helpful for debugging and development.
* Uses nmcli, a command-line tool for NetworkManager, requiring it to be installed and properly configured.

**Runtime Prerequisites**

* Linux OS (Debian-based recommended)
* nmcli installed
* dnsmasq installed and configured with NetworkManager
* Sudo rights to manage DNS redirection

**6. Use Case Scenarios**

**6.1 Boot with No Known Network**

1. Device powers on and fails to connect to any known WiFi.
2. Automatically sets up an AP.
3. User connects to AP, configures SSID/password via frontend.
4. Device switches to new network, saves credentials.

**6.2 Controlled Network Reset**

1. Device receives command to clear network settings.
2. Calls wifi\_disconnect() and removes configurations.
3. Switches to AP mode for new configuration.

**6.3 Fallback to Local Mode**

1. Device connects to known network without internet.
2. get\_state() reports local network instead of full connectivity.
3. Controller may choose to enter fallback logic or retry.

**7. Development and Testing**

The module includes an interactive CLI (\_\_main\_\_) that allows the developer to:

* Inspect current state
* Scan networks
* Connect/disconnect manually
* Start/stop AP
* Add/remove connections from NetworkManager

This feature makes it ideal for rapid prototyping and testing in lab environments.

**8. Future Extensions**

Possible areas for enhancement:

* Support for WEP/EAP networks
* Better garbage collection for orphaned NetworkManager entries
* Connection prioritization (e.g. auto-prefer strongest known network)
* Integration with a frontend for live configuration