# Architectural Integration of Embedded ESP32 Control Systems with OpenWrt Network Infrastructure

## 1. Executive Summary and System Architecture

The convergence of high-performance embedded networking equipment and low-power IoT microcontrollers necessitates a paradigmatic shift in how network administration interfaces are designed and consumed. This comprehensive research report analyzes the architectural requirements, protocol implementations, and software engineering principles required to develop a decentralized control interface for a Xiaomi Mi Router 4A Gigabit Edition running OpenWrt 24.10. Specifically, the project entails the deployment of an ESP32 microcontroller as a dedicated, mobile-optimized web server that acts as a proxy controller for the router's internal state machine.

The analysis is grounded in the specific operational environment depicted in the provided documentation: a MediaTek MT7621-based architecture operating within a 192.168.10.x subnet, utilizing the adblock package with a dnsmasq backend for traffic filtering. The primary objective is to bypass the traditional, desktop-centric LuCI web interface in favor of a lightweight, responsive dashboard hosted on the ESP32 that provides real-time telemetry—specifically connected client enumeration and data throughput—and active security management via domain allowlisting and blacklisting.

To achieve this, the system relies on the OpenWrt ubus (micro bus) architecture, accessed via a JSON-RPC over HTTP bridge. This report details the rigorous configuration of the OpenWrt uhttpd server to support Cross-Origin Resource Sharing (CORS), the precise transactional logic required to manipulate the Unified Configuration Interface (UCI) for ad-blocking, and the firmware logic required on the ESP32 to authenticate, query, and command the router securely. The proposed solution effectively decouples the control plane from the data plane, leveraging the ESP32 as an orchestration agent that translates user intent into low-level system calls.1

## 2. Analysis of the Target OpenWrt Environment

Successful integration requires a forensic understanding of the target device's software stack. The provided telemetry indicates the device is a Xiaomi Mi Router 4A Gigabit Edition, a device renowned for its price-to-performance ratio but constrained by flash storage, making efficient IPC (Inter-Process Communication) critical.

### 2.1 Firmware and Kernel Architecture

The router is running OpenWrt 24.10.0 (Snapshot r28427), leveraging the Linux Kernel 6.6.73 [Image 1]. This implies a modern netfilter architecture (likely utilizing nftables over legacy iptables) and a mature ubus implementation. The architecture is ramips/mt7621, a dual-core multi-threading platform. For the external controller (ESP32), this architecture is relevant because it dictates the performance characteristics of the API responses; the MT7621 is powerful enough to handle rapid JSON serialization without inducing significant CPU load, ensuring that the dashboard remains responsive even during high-traffic events.

The network topology identified places the router at LAN IP 192.168.10.100. Crucially, the "IPv4 Upstream" is 192.168.0.115 with a gateway of 192.168.0.1 [Image 1]. This indicates the OpenWrt device is operating in a cascaded router setup (Double NAT) or as a wireless client to a primary ISP modem. The ESP32 is identified on the network with the hostname esp32-8989A4 and IP 192.168.10.193 [Image 1]. This confirms connectivity is established, removing Layer 1 and Layer 2 connectivity as potential hurdles; the focus is entirely on Layer 7 application logic.

### 2.2 The Adblock Subsystem

The active security mechanism is the adblock package (version 4.4.2-r3), configured to use dnsmasq as the DNS backend [Image 2]. This is a pivotal architectural detail. Unlike DNS-over-HTTPS (DoH) proxies or heavy filtering engines like AdGuard Home, the adblock package works by generating large lists of domains formatted for dnsmasq (e.g., address=/ads.google.com/0.0.0.0), forcing the DNS resolver to return a null IP for blacklisted domains.

To "control" this from the ESP32, one does not interact with the blocklists directly in real-time RAM. Instead, one must manipulate the persistent UCI configuration files that define *custom* blacklists and then trigger a service reload. The reload process invokes a shell script that aggregates the subscribed feeds (seen in Image 2: "Blocked Domains: 19") with the local user-defined blacklists, compiles them into a format dnsmasq understands, and restarts the DNS resolver.4 This implies that "blocking a website" is an asynchronous operation with a latency of several seconds, which must be accounted for in the User Interface (UI) design.

## 3. The ubus Inter-Process Communication Protocol

The core enabler of this project is ubus. In OpenWrt, ubus replaces the heavy D-Bus found in desktop Linux distributions. It is a lightweight RPC mechanism that allows system daemons to expose methods and objects.

### 3.1 The HTTP-to-Ubus Bridge

While ubus natively runs over Unix Domain Sockets, the uhttpd web server includes a plugin (uhttpd-mod-ubus) that exposes this bus via HTTP POST requests. This allows the ESP32, which lacks local socket access to the router, to send commands over the network. The endpoint is typically http://192.168.10.100/ubus.

The communication protocol adheres strictly to JSON-RPC 2.0. Every interaction follows a request-response cycle. The ESP32 sends a JSON payload specifying a session ID, an object (e.g., network.device), a method (e.g., status), and parameters. The router executes the C-code function associated with that method and returns a JSON response.1

### 3.2 Authentication and Session Lifecycle

The ubus interface is not open; it is guarded by the rpcd (RPC Daemon) Access Control Lists (ACLs). The ESP32 must perform a cryptographic login handshake to obtain a Session ID (SID).

The login procedure targets the session object:

* **Method:** login
* **Parameters:** username (usually "root") and password (the router's web interface password).

Upon success, rpcd generates a 32-character alphanumeric string (the SID). This SID is valid for a default duration of 300 seconds (5 minutes).1 The analysis of the documentation highlights a critical requirement for the ESP32 firmware: **Session State Management**. The ESP32 cannot assume a static session key. It must implement a logic flow that authenticates, caches the SID, and monitors API responses for "Access Denied" (Error Code 6). Upon receiving such an error, the controller must automatically re-authenticate and retry the command, ensuring resilience against router reboots or session timeouts.1

### 3.3 Security: Cross-Origin Resource Sharing (CORS)

A major hurdle identified in the research is the browser's Same-Origin Policy. The user desires a "new website" hosted on the ESP32 (e.g., at http://192.168.10.193) that controls the router (http://192.168.10.100). If the JavaScript running on the user's mobile phone attempts to fetch data directly from the router, the browser will block the request because the origins (IP addresses) differ.

To resolve this, the OpenWrt router must be configured to send CORS headers. The uhttpd server supports a configuration option ubus\_cors. When enabled, uhttpd responds to browser preflight (OPTIONS) requests with headers permitting the cross-origin call (Access-Control-Allow-Origin: \*). Without this specific configuration on the router, the "Direct Browser-to-Router" architecture is impossible, and the ESP32 would have to act as a heavy proxy, relaying all data.7

## 4. Configuration of the OpenWrt Router

Before the ESP32 can exert control, specific configurations must be applied to the OpenWrt system to expose the necessary interfaces and ensure compatibility with the external web client.

### 4.1 Enabling External API Access (CORS)

The standard uhttpd configuration blocks cross-origin requests by default. To enable the dashboard to function, the ubus\_cors parameter must be toggled.

Implementation Procedure:

Access the router via SSH and modify /etc/config/uhttpd. The relevant configuration block is config uhttpd 'main'.

Bash

uci set uhttpd.main.ubus\_cors='1'  
uci commit uhttpd  
/etc/init.d/uhttpd restart

This command sequence instructs the UCI system to update the configuration file and restarts the web server process. Once active, uhttpd will append the Access-Control-Allow-Origin header to ubus responses, permitting the mobile browser to render the data retrieved from the router.7

### 4.2 Verifying adblock Configuration

The user's requirement to manage "allowlist and blacklist" necessitates that the adblock package is configured to accept custom lists. In the provided image, we see the package is active. The underlying configuration file is /etc/config/adblock.

We must ensure that the global configuration points to a writable blacklist. Typically, this is handled by the blacklist\_domains list option within the global section (or sometimes a specific source section depending on the exact package version, though global is standard for manual entries). The backend dnsmasq handles the actual filtering logic. When a user adds a domain, the system must trigger a reload. The efficiency of this reload depends on the trigger\_delay (set to 5 seconds in Image 2), which debounces rapid changes.4

## 5. ESP32 Controller Firmware Architecture

The ESP32 firmware serves as the bridge between the user's mobile device and the router's API. It must handle Wi-Fi connectivity, serve the HTTP assets (HTML/CSS/JS), and orchestrate the JSON-RPC calls.

### 5.1 Network Stack and Libraries

The implementation relies on the WiFi.h library for connectivity and HTTPClient.h for issuing POST requests to the router. Critically, the ArduinoJson library is required for constructing the complex nested JSON objects required by the JSON-RPC protocol and for parsing the router's responses. Given the memory constraints of the ESP32 (roughly 320KB RAM active), parsing large JSON responses—such as the full list of DHCP leases—requires careful memory management, favoring dynamic allocation or streaming parsers over large static buffers.9

### 5.2 The Frontend: Mobile-First Web Interface

The user specifically requested a "focus on mobile view." This dictates the use of Responsive Web Design (RWD) principles within the HTML served by the ESP32.

**Design Specification:**

* **Viewport:** The HTML must include <meta name="viewport" content="width=device-width, initial-scale=1"> to ensure correct scaling on smartphones.
* **CSS Architecture:** A CSS Grid or Flexbox layout is optimal. On mobile devices, the interface should stack vertically (Connected Clients -> Data Usage -> Blacklist Controls). On tablets or desktops, it can reflow to a dashboard layout.
* **Embedded Assets:** To ensure the ESP32 is self-contained, the HTML, CSS, and JavaScript should be embedded directly in the C++ code as PROGMEM strings or served from the SPIFFS/LittleFS file system, rather than relying on external CDNs which might be blocked or slow.11

**Table 1: Proposed API Endpoints on ESP32 Web Server**

| **Endpoint** | **Method** | **Function** | **Description** |
| --- | --- | --- | --- |
| / | GET | Serve UI | Returns the HTML/CSS/JS dashboard. |
| /api/stats | GET | Telemetry | Triggers ESP32 to query router stats and return simplified JSON. |
| /api/block | POST | Control | Accepts {domain: "example.com"} and triggers blacklist logic. |
| /api/allow | POST | Control | Accepts {domain: "example.com"} and triggers allowlist logic. |

### 5.3 The OpenWrt Client Class

The core logic resides in a dedicated C++ class, OpenWrtClient. This class encapsulates the complexity of the ubus protocol.

**Key Methods:**

1. **login()**: Performs the initial handshake. Stores the resulting SID.
2. **call(object, method, params)**: A generic wrapper that constructs the JSON-RPC envelope, injects the SID, sends the HTTP POST, and returns the response document.
3. **getConnectedClientCount()**: Calls dhcp/ipv4leases, counts the array elements.
4. **getDataUsage()**: Calls network.device/status, extracts RX/TX bytes.
5. **blockDomain(domain)**: Executes the multi-step transaction to update UCI and reload services.

## 6. Implementation of Monitoring Features

### 6.1 Feature: "Show How Many People Are Connected"

To enumerate connected devices, the most reliable data source in OpenWrt is the active DHCP leases. While this does not strictly represent "currently active" devices (as leases persist offline), it is the standard metric for "connected devices."

**The ubus Command:**

* **Namespace:** dhcp
* **Method:** ipv4leases (or luci-rpc getDHCPLeases if available).
* **Parameters:** {} (empty object).12

Data Processing on ESP32:

The router returns a JSON object containing an array of lease objects.

JSON

{  
 "device": {  
 "hostname": "esp32-8989A4",  
 "macaddr": "6C:C8:40:89:89:A4",  
 "ipaddr": "192.168.10.193"  
 }  
}

The ESP32 parses this array. The integer count of objects in the array represents the number of clients. To refine this, the ESP32 can filter out leases where the MAC address matches the ESP32's own MAC, preventing self-counting.

### 6.2 Feature: "Data Used"

Data usage is tracked via the network interface counters maintained by the Linux kernel.

**The ubus Command:**

* **Namespace:** network.device
* **Method:** status
* **Parameters:** {"name": "br-lan"}.1

Metric Selection:

The br-lan interface aggregates all traffic on the Local Area Network (WiFi and Ethernet). The response includes statistics:

* rx\_bytes: Total bytes received by the router interface (Upload from clients).
* tx\_bytes: Total bytes transmitted by the router interface (Download to clients).

Visualization:

The ESP32 receives raw byte integers (e.g., 45910293). The JavaScript on the frontend is responsible for formatting this into human-readable units (MB, GB). Note that these counters reset upon router reboot; they represent "Session Data Usage." For persistent usage, access to the vnstat database would be required, which is significantly more complex and likely out of scope for a basic controller.

## 7. Implementation of Control Features: The Allowlist/Blacklist

This requirement involves the most complex logic, as it transitions from "reading" status to "writing" configuration. This process in OpenWrt is transactional to prevent corruption.

### 7.1 The Transactional Model

One cannot simply "write" a domain to the router. The process involves three distinct steps via ubus:

1. **Stage:** Add the domain to the configuration delta in memory.
2. **Commit:** Write the memory delta to the persistent filesystem (/etc/config/adblock).
3. **Apply:** Trigger the service to reload the configuration and regenerate the DNS blocklists.

### 7.2 Step 1: Modifying the Configuration

The uci object handles configuration. We use the add\_list method to append a domain to the blacklist.

**Command Structure:**

* **Object:** uci
* **Method:** add\_list
* **Arguments:**  
  JSON  
  {  
   "config": "adblock",  
   "section": "global",  
   "option": "blacklist\_domains",  
   "values": "example-malware.com"  
  }

**Critical Distinction:** We use add\_list rather than set. set would replace the existing list with the single new domain, wiping out previous entries. add\_list safely appends the new entry.14

### 7.3 Step 2: Committing Changes

Once the list is updated in the staging area, it must be committed to flash memory.

* **Object:** uci
* **Method:** commit
* **Arguments:** {"config": "adblock"}.15

### 7.4 Step 3: Triggering Enforcement

The changes are now saved but not active. The dnsmasq resolver is still using the old host files. We must trigger the adblock service to reload. The adblock script handles the complex logic of downloading (if needed) and regenerating the dnsmasq compatible list.

The Reload Trigger:

The most robust method to trigger this via ubus is utilizing the service namespace if the adblock service is procd-managed (which it is in OpenWrt 24.10).

* **Object:** service
* **Method:** list (to find the instance) implies we can manage it, but service usually exposes action methods or we rely on the uci commit trigger.
* **Reliable Method:** In many setups, ubus call uci commit automatically triggers a reload if the service has registered a file trigger. However, to force it, we can use the luci-rpc method setInitAction if available, or simpler, rely on ubus call service signal if supported.
* **Alternative:** A common workaround for ensuring reload is executing ubus call uci apply. This triggers a system-wide configuration application, which acts on all pending changes, ensuring the adblock service picks up the new domains.15

## 8. Detailed ESP32 Firmware Logic (Pseudo-Code)

The following logic flow describes the ESP32's loop() or async handler for the "Block Domain" button:

C++

void handleBlockRequest(String domain) {  
 if (!openwrt.checkSession()) {  
 openwrt.login();   
 }  
  
 // Step 1: Add to List  
 // Payload: {"jsonrpc": "2.0", "method": "call", "params":}  
 bool stageSuccess = openwrt.sendRequest("uci", "add\_list",...);  
  
 if (stageSuccess) {  
 // Step 2: Commit  
 // Payload: {"jsonrpc": "2.0", "method": "call", "params":}  
 bool commitSuccess = openwrt.sendRequest("uci", "commit",...);  
   
 if (commitSuccess) {  
 // Step 3: Apply/Reload  
 // This might take 10-20 seconds. The web UI should show a spinner.  
 openwrt.sendRequest("uci", "apply",...);  
 return "Success";  
 }  
 }  
 return "Error";  
}

**Table 2: Error Handling Matrix**

| **Scenario** | **Router Response** | **ESP32 Action** | **User Feedback** |
| --- | --- | --- | --- |
| **Session Expired** | Code 6 (Permission Denied) | Call login(), update SID, retry request. | None (transparent). |
| **Invalid Domain** | UCI Error or Parse Error | Log to Serial. | "Invalid format." |
| **Router Offline** | HTTP Connection Failed | Retry 3 times, then abort. | "Router Unreachable." |
| **Commit Timeout** | No Response (Timeout) | Assume success (async reload). | "Processing..." |

## 9. Frontend Engineering: The Mobile Dashboard

The user interface serves as the command surface. To meet the "mobile view" requirement, the HTML structure uses CSS Flexbox to create a vertical flow layout that adapts to the narrow screens of mobile devices.

### 9.1 HTML/CSS Integration

The UI code is stored as a const char\* raw string in the ESP32 firmware.

**CSS Strategy:**

* **Container:** A central .container with max-width: 600px and margin: 0 auto ensures the content acts like an "app" in the center of a tablet screen but fills a mobile screen.
* **Cards:** Data points are encapsulated in .card divs with box-shadow and border-radius, mimicking modern mobile OS widgets (iOS/Android style).
* **Typography:** Large fonts (2rem) for key metrics (Client Count) ensuring readability at a glance.
* **Interactivity:** Buttons are sized to 44px height minimum to meet touch target accessibility standards.

### 9.2 Asynchronous JavaScript (AJAX)

To ensure the dashboard feels "live" without reloading the page:

1. **Polling:** A setInterval function runs every 5000ms (5 seconds).
2. **Fetching:** It calls the ESP32 endpoint /api/stats.
3. **DOM Manipulation:** It parses the JSON response and updates the innerHTML of the specific ID elements (client-count, data-rx, data-tx).
4. **Blocking UI:** When "Block" is clicked, the button is disabled and its text changed to "Blocking..." to prevent double-submissions during the slow UCI commit process.

## 10. Conclusion and Strategic Outlook

This report delineates a comprehensive architecture for creating a dedicated hardware controller for an OpenWrt router. By exploiting the ubus IPC mechanism via HTTP, we transform the ESP32 from a simple microcontroller into a powerful network administration tool.

The critical path to success involves:

1. **Configuration:** Enabling ubus\_cors on the router to permit API access.
2. **Authentication:** Implementing robust session token management on the ESP32.
3. **Transaction Management:** Correctly sequencing the uci add, commit, and apply stages to enforce security policies.

This system provides a distinct advantage over standard mobile apps or web interfaces: it is a dedicated, physical interface that operates independently of the user's primary computing device, offering immediate, at-a-glance network telemetry and control. The modular nature of the ubus API further allows this system to be expanded; future iterations could include toggling Guest WiFi, rebooting the router, or managing parental control schedules, all utilizing the same foundational JSON-RPC architecture established in this study.

#### Works cited

1. [OpenWrt Wiki] ubus (OpenWrt micro bus architecture), accessed on December 4, 2025, <https://openwrt.org/docs/techref/ubus>
2. OpenWrt ubus RPC daemon for backend server - [OpenWrt Wiki] rpcd, accessed on December 4, 2025, <https://openwrt.org/docs/techref/rpcd>
3. Getting started with OpenWrt Micro Bus - ubus - Zilogic Systems, accessed on December 4, 2025, <https://zilogic.com/blog/ubus-service-development-with-rpcd.html>
4. [OpenWrt Wiki] Ad blocking, accessed on December 4, 2025, <https://openwrt.org/docs/guide-user/services/ad-blocking>
5. Adblock on OpenWRT: A Complete Guide to Blocking Ads on All Devices in Your Network, accessed on December 4, 2025, <https://itorakul.com.ua/en/adblock-on-openwrt/>
6. OpenWRT UBUS - HackMD, accessed on December 4, 2025, <https://hackmd.io/@rYMqzC-9Rxy0Isn3zClURg/H1BY98bRw>
7. CORS on OpenWRT UCI - GitHub Gist, accessed on December 4, 2025, <https://gist.github.com/782fb3e7a6bc0413c270e83f8bcabbea>
8. Topic: Fast/lean adblocking for OpenWrt/LEDE with web ui: simple-adblock, accessed on December 4, 2025, <https://forum.archive.openwrt.org/viewtopic.php?id=62204>
9. ESP32 HTTP Client Request with JSON Body - John M. Wargo, accessed on December 4, 2025, <https://johnwargo.com/posts/2025/http-client-esp32-with-json-body/>
10. ESP32 WiFi Tutorial & Library Examples (Arduino IDE) - DeepBlueMbedded, accessed on December 4, 2025, <https://deepbluembedded.com/esp32-wifi-library-examples-tutorial-arduino/>
11. ESP32 Web Server using LittleFS Filesystem - Random Nerd Tutorials, accessed on December 4, 2025, <https://randomnerdtutorials.com/esp32-web-server-littlefs/>
12. ubus call luci-rpc getDHCPLeases: The same device returns different mac address (DHCP, DHCPv6) information · Issue #4159 - GitHub, accessed on December 4, 2025, <https://github.com/openwrt/luci/issues/4159>
13. Location of client leases in filesystem? : r/openwrt - Reddit, accessed on December 4, 2025, <https://www.reddit.com/r/openwrt/comments/fvtcrj/location_of_client_leases_in_filesystem/>
14. Execute Openwrt UCI add\_list through ubus call - Stack Overflow, accessed on December 4, 2025, <https://stackoverflow.com/questions/59544053/execute-openwrt-uci-add-list-through-ubus-call>
15. [OpenWrt Wiki] UCI (Unified Configuration Interface) – Technical Reference, accessed on December 4, 2025, <https://openwrt.org/docs/techref/uci>