OBD Can Monitor Project Documentation
ESP32 – TWAI Driver Layer

Embedded Systems for IoT Project - Group 35

2026

Contents

Chapter 1

System Architecture and Project Structure

1.1 Project Folder Structure

The folder structure of the project, inferred from `CMakeLists.txt` and including header files, is as follows:

```
CMakeLists.txt
├── partitions.csv
└── main/
    ├── CMakeLists.txt
    ├── app_main.c
    ├── can/
    │   ├── can_bus.c
    │   └── can_bus.h
    ├── obd/
    │   ├── obd.c
    │   ├── obd.h
    │   └── obd_data.c
    └── web/
        ├── web_server.c
        ├── web_server.h
        ├── index.html
        ├── graph.html
        ├── errors.html
        ├── diagnostics.html
        ├── script.js
        ├── style.css
        ├── js/
        │   ├── chart-script.js
        │   ├── animations.js
        │   └── diagnostics-script.js
        ├── lib/
        │   ├── chart.js
        │   └── luxox.js
        └── fonts/
            ├── font-awesome/css/all.min.css
            ├── font-awesome/webfonts/fa-solid-900.woff2
            └── font-awesome/webfonts/fa-regular-400.woff2
```

```
│
│
│         font-awesome/webfonts/fa-brands-400.woff2
│         Inter-Bold.woff2
│         Inter-Regular.woff2
└── lcd/
     ├── lcd.c
     └── lcd.h
```

1.2 Block Diagram: Data Flow

The data flow in the OBD-II system is illustrated as:

CAN Bus → Scheduler → OBD Data Processing → JSON Encoder → Web Server → Browser Dashboard

1.2.1 Description of Components

- **CAN Bus:** Receives raw sensor data from the vehicle using OBD-II protocol.
- **Scheduler:** Manages timing and prioritization of reading tasks.
- **OBD Data Processing:** Parses CAN packets and calculates key parameters (RPM, speed, coolant temperature, engine load, battery voltage, etc.).
- **JSON Encoder:** Serializes processed data into JSON for efficient transmission.
- **Web Server:** Serves embedded HTML/CSS/JS pages to display real-time dashboards.
- **Browser Dashboard:** Displays KPIs, charts, and trends.

1.3 Architectural Diagram

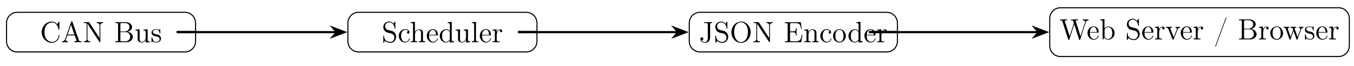


Figure 1.1: Communication flow from CAN Bus to Web interface

1.4 CMakeLists.txt

The `main/CMakeLists.txt` file registers all source files, include directories, and embedded web files. The main sections are:

- **SRCS:** Lists all C source files: `app_main.c`, `can_bus.c`, `obd.c`, `obd_data.c`, `web_server.c`, `lcd.c`.
- **INCLUDE_DIRS:** Header directories corresponding to the source files: `.`, `can`, `obd`, `web`, `lcd`.
- **REQUIRES:** ESP-IDF components such as: `esp_http_server`, `driver`, `nvs_flash`, `esp_wifi`, `esp_netif`, `esp_event`, `lwip`.
- **EMBED_FILES:** Web resources are embedded into the firmware: HTML, CSS, JS, chart libraries, fonts.

1.5 Partition Table and Functional Dependencies

1.5.1 Partition Table Overview

The ESP32 project uses a custom partition table to organize flash memory for code, configuration, and storage. The table is defined as follows:

Name	Type	SubType	Offset	Size	Flags
nvs	data	nvs	-	0x4000	-
otadata	data	ota	-	0x2000	-
phy_init	data	phy	-	0x1000	-
factory	app	factory	-	0x300000	-
storage	data	spiffs	-	0xDE000	-

Table 1.1: Custom ESP32 Partition Table

Note: The `factory` partition has been allocated 3 MB to ensure there is sufficient space for all embedded fonts, JavaScript libraries, and web assets. This large allocation avoids running out of space when including files such as `chart.js`, `luxox.js`, and all the font files required by the web dashboard.

1.5.2 Functional Dependencies Between Project Files

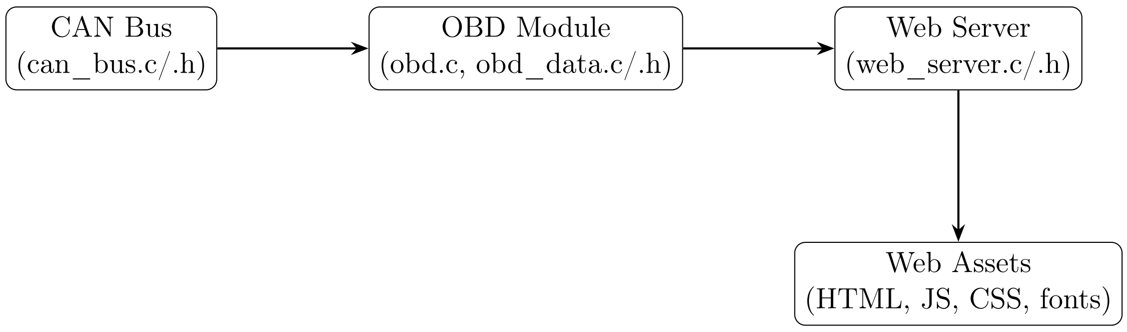
The project is structured into multiple components that interact through well-defined interfaces. Understanding the dependencies is crucial for maintenance, debugging, and further development. The main functional relationships are summarized below:

- **CAN Bus Module** (`can/can_bus.c, .h`)
  - Responsible for reading raw OBD-II data from the vehicle CAN bus.
  - Provides processed signals to the OBD module.
  - Depends on `driver` components from ESP-IDF for CAN interface.
- **OBD Module** (`obd/obd.c, obd_data.c, .h`)
  - Converts raw CAN messages into meaningful OBD-II parameters.
  - Maintains internal data structures representing engine status, fuel system, environmental conditions, and electrical system.
  - Exposes APIs for both the `web` module and charting scripts.
  - Depends on `can_bus` module for data acquisition.
- **Web Server** (`web/web_server.c, .h`)
  - Hosts the web dashboard and serves static assets (HTML, CSS, JS, fonts).
  - Converts internal OBD-II structures into JSON format for web consumption.
  - Provides endpoints for chart data, diagnostic reports, and export functionality.
  - Depends on `obd` module for real-time data access.
- **Web Assets** (`web/*.html, web/*.js, web/*.css, web/fonts`)
  - Handle UI rendering, animation, and charting.
  - Scripts like `script.js`, `chart-script.js`, `animations.js` consume JSON data from the web server.

- Dependencies between scripts are carefully managed to avoid overwriting DOM elements: for instance, `animations.js` updates the gauges, while `chart-script.js` only reads OBD data for charts.

- **CMake Build System** (`CMakeLists.txt`)
  - Registers all source files and include directories.
  - Embeds web assets into flash memory via `EMBED_FILES` for SPIFFS access.
  - Declares component dependencies on ESP-IDF modules such as `esp_http_server`, `driver`, `nvs_flash`, `esp_wifi`, `lwip`.

Summary Diagram of Dependencies



This diagram highlights the **functional flow**: raw vehicle data is acquired by the CAN module, processed by the OBD module, exposed via the web server, and finally rendered in the browser through embedded assets and scripts.

Chapter 2

CAN Communication Module

2.1 Overview

The file `main/can/can_bus.c` implements a hardware abstraction layer (HAL) for CAN communication using the ESP-IDF TWAI driver (Two-Wire Automotive Interface), Espressif's implementation of the CAN 2.0 protocol controller.

This module encapsulates:

- CAN peripheral initialization
- Frame transmission
- Frame reception

It provides a clean interface to higher-level application modules without exposing low-level TWAI driver configuration details.

2.2 Dependencies

The module depends on:

- `can_bus.h` – local interface header
- `esp_log.h` – logging system
- ESP-IDF TWAI driver

The TWAI driver is configured in **Normal Mode** at **500 kbit/s**, which is standard for OBD-II CAN (ISO 15765-4).

2.3 Static Module Variables

```c
static const char *TAG = "CAN_BUS";
```

Purpose

This static string is used by the ESP-IDF logging system to identify log messages originating from this module.

Design Notes

- Declared `static` to limit visibility to this translation unit.
- Ensures modular logging traceability.

2.4 Function: `can_bus_init`

2.4.1 Prototype

```c
void can_bus_init(void);
```

2.4.2 Purpose

Initializes the ESP32 TWAI driver and starts the CAN peripheral.

This function performs:

1. General driver configuration
2. Bit timing configuration
3. Acceptance filter configuration
4. Driver installation
5. Driver start

2.4.3 Implementation

```c
void can_bus_init(void)
{
    twai_general_config_t g_config =
        TWAI_GENERAL_CONFIG_DEFAULT(GPIO_NUM_5, GPIO_NUM_4,
   TWAI_MODE_NORMAL);

    twai_timing_config_t t_config =
        TWAI_TIMING_CONFIG_500KBITS();

    twai_filter_config_t f_config =
        TWAI_FILTER_CONFIG_ACCEPT_ALL();

    ESP_ERROR_CHECK(twai_driver_install(&g_config, &t_config, &f_config)
   );
    ESP_ERROR_CHECK(twai_start());

    ESP_LOGI(TAG, "CAN init OK");
}
```

2.4.4 Configuration Breakdown

General Configuration

- TX pin: GPIO 5
- RX pin: GPIO 4
- Mode: Normal mode

Normal mode enables full CAN communication (transmit and receive).

Timing Configuration

- Bitrate: 500 kbit/s

This bitrate is compliant with:

- ISO 15765-4 (OBD-II over CAN)
- Most automotive CAN networks

Filter Configuration

- Accept-all filter

All CAN frames are accepted regardless of ID.

2.4.5 Error Handling

The macro:

```
ESP_ERROR_CHECK(...)
```

- Aborts execution if initialization fails.
- Ensures system integrity.

2.4.6 Design Considerations

- The function does not return an error code; failure results in system abort.
- Suitable for early system initialization phase.
- Could be extended to support configurable bitrate or filtering.

2.5 Function: `can_bus_send`

2.5.1 Prototype

```
esp_err_t can_bus_send(twai_message_t *msg);
```

2.5.2 Purpose

Transmits a CAN frame over the bus.

2.5.3 Parameters

- `msg`: Pointer to a `twai_message_t` structure containing:
  - Identifier
  - Data length code (DLC)
  - Data payload
  - Flags

2.5.4 Implementation

```
esp_err_t can_bus_send(twai_message_t *msg)
{
    return twai_transmit(msg, pdMS_TO_TICKS(100));
}
```

2.5.5 Timeout Behavior

- Timeout: 100 ms
- Converted using `pdMS_TO_TICKS()`

If the transmit queue is full:

- The function blocks up to 100 ms.
- Returns timeout error if unsuccessful.

2.5.6 Return Values

- `ESP_OK` – Transmission successful
- `ESP_ERR_TIMEOUT` – Timeout occurred
- Other TWAI driver errors

2.5.7 Design Considerations

- Non-infinite blocking prevents deadlocks.
- Suitable for task-level transmission.
- ISR-safe transmission would require zero timeout.

2.6 Function: `can_bus_receive`

2.6.1 Prototype

```c
esp_err_t can_bus_receive(twai_message_t *msg, TickType_t timeout);
```

2.6.2 Purpose

Receives a CAN frame from the TWAI driver receive queue.

2.6.3 Parameters

- `msg`: Pointer to a message structure where the received frame will be stored.
- `timeout`: Maximum blocking time (FreeRTOS ticks).

2.6.4 Implementation

```c
esp_err_t can_bus_receive(twai_message_t *msg, TickType_t timeout)
{
    return twai_receive(msg, timeout);
}
```

2.6.5 Behavior

- Blocks until:
  - A frame is received
  - Timeout expires

2.6.6 Return Values

- `ESP_OK` – Frame received
- `ESP_ERR_TIMEOUT` – No frame received within timeout
- Other TWAI driver errors

2.6.7 Design Considerations

- Flexible timeout allows:
  - Blocking reception
  - Non-blocking polling
- Suitable for dedicated CAN receive task.

2.7 Architectural Role in the System

This module acts as:

Hardware Abstraction Layer (HAL) for CAN

It isolates:

- Pin configuration
- Driver installation
- Timing parameters
- TWAI-specific API calls

Higher-level modules should never directly call TWAI driver functions.

Chapter 3

OBD Communication Layer

3.1 Overview

The OBD module implements a structured and deterministic polling system for retrieving diagnostic parameters (PIDs) from an ECU over CAN using the ISO 15765-4 protocol (OBD-II over CAN, 500 kbit/s).

The architecture is divided into two main components:

- **obd.c** – Hardware interaction layer using the CAN HAL
- **obd_data.c** – Real-time priority scheduler and shared data storage

This separation ensures:

- Hardware abstraction
- Deterministic scheduling
- Thread-safe data access
- Scalability and maintainability

3.2 File: `obd.c`

3.2.1 Purpose

The file `obd.c` implements:

- Single PID request logic
- OBD layer initialization

It does not directly use the TWAI driver. Instead, it relies exclusively on the CAN Hardware Abstraction Layer (`can_bus.c`).

3.2.2 Architectural Role

$$\text{OBD Layer} \rightarrow \text{CAN HAL} \rightarrow \text{TWAI Driver}$$

This design enforces proper layering and prevents driver duplication.

3.2.3 Function: `obd_read_pid`

Prototype

```
bool obd_read_pid(uint8_t pid, uint8_t out[4]);
```

Purpose

Sends an OBD-II request for a specific PID and waits for a matching response frame.

Request Frame Format

- CAN ID: 0x7DF (Functional Broadcast)
- Mode: 0x01 (Current Data)
- PID: variable

Implementation

```
twai_message_t tx = {
    .identifier = 0x7DF,
    .data_length_code = 8,
    .data = {0x02, 0x01, pid, 0, 0, 0, 0, 0}
};
```

Response Filtering

The function waits for:

- Identifier in range 0x7E8 – 0x7EF
- Matching PID in data[2]

Timeout Policy

- Total wait window: 100 ms
- Internal receive timeout: 50 ms

Return Value

- `true` – Valid PID response received
- `false` – Timeout or transmission error

3.2.4 Function: `obd_init`

Purpose

Initializes shared data storage and starts the real-time polling scheduler.

Implementation

```c
void obd_init(void)
{
    obd_data_init();
    ESP_LOGI(TAG, "OBD layer started. Using CAN HAL.");
    obd_data_start_polling();
}
```

Design Notes

- CAN initialization must already be performed by `can_bus_init()`
- No direct TWAI driver calls appear in this file

3.3 File: `obd_data.c`

3.4 Overview

This file implements:

- Real-time PID polling scheduler
- Priority-based request logic
- Adaptive retry and backoff
- Thread-safe shared data storage

The scheduler continuously queries 18 PIDs with different update rates.

3.5 Shared Data Storage

3.5.1 Structure

```c
static obd_full_data_t s_obd;
static SemaphoreHandle_t s_obd_mutex;
```

3.5.2 Design Pattern

- Local snapshot updated in RT task
- Atomic publish using mutex
- Readers obtain a copy

This eliminates partial-update race conditions.

3.6 Scheduler Model

3.6.1 Priority Levels

- HIGH (100 ms period)
- MEDIUM (500 ms period)
- LOW (2000 ms period)

3.6.2 Job Descriptor

```c
typedef struct
{
    uint8_t pid;
    pid_prio_t prio;
    uint32_t period_ms;
    uint32_t next_due_ms;
    uint8_t fail_count;
    uint32_t backoff_until;
} pid_job_t;
```

3.6.3 Key Fields

- `period_ms` – Desired update interval
- `next_due_ms` – Absolute scheduling timestamp
- `fail_count` – Consecutive failures
- `backoff_until` – Temporary suspension time

3.7 Job Selection Algorithm

3.7.1 Function: `pick_next_job`

The scheduler selects:

1. Only jobs whose deadline has expired
2. Jobs not in backoff
3. Highest priority first
4. Among equal priority, most late job

3.7.2 Lateness Calculation

```
int32_t lateness = (int32_t)(tnow - j->next_due_ms);
```

This ensures deterministic priority scheduling.

3.8 Real-Time Task

3.8.1 Function: `obd_rt_task`

Core Loop

```
while (1)
{
    int idx = pick_next_job(tnow);
    ...
    bool ok = obd_read_pid(j->pid, b);
    ...
    vTaskDelay(pdMS_TO_TICKS(OBD_REQ_SPACING_MS));
}
```

3.8.2 Execution Policy

- One PID request every 25 ms
- Prevents CAN burst flooding
- Maintains stable ECU load

3.9 Failure Handling Strategy

3.9.1 Retry Phase

If failures $<$ threshold:

- Retry after period / 2

3.9.2 Backoff Phase

If failures exceed threshold:

- Base delay: 2000 ms
- Additional delay grows per failure
- Maximum backoff cap: 20 seconds

This prevents ECU overload in case of unsupported PIDs.

3.10 Data Conversion

The function `apply_pid_value()` implements SAE J1979 conversion formulas.
Example:

Engine RPM

```
((A * 256) + B) / 4
```

Engine Load

```
(A * 100) / 255
```

All conversions are compliant with OBD-II standards.

3.11 Public API

3.11.1 Initialization

```c
void obd_data_init(void);
void obd_data_start_polling(void);
```

3.11.2 Data Access

```c
obd_full_data_t obd_get_all_data(void);
```

Returns a thread-safe copy of the latest dataset.

3.12 Real-Time Characteristics

- Deterministic scheduling
- Bounded blocking times
- ECU-safe spacing
- Adaptive error handling

3.13 Architectural Summary

Web Server
OBD Data Scheduler
OBD Request Layer
CAN HAL
TWAI Driver

Chapter 4

Web Server Module

4.1 Overview

The Web Server module provides a local HTTP interface for visualizing vehicle telemetry data acquired through the OBD subsystem.

It is built using the ESP-IDF `esp_http_server` component and implements:

- A JSON REST endpoint (`/data`)
- Static file serving (HTML, CSS, JS, fonts)
- Chunked transfer for large resources

The module is designed to:

- Avoid blocking real-time tasks
- Maintain thread-safe data access
- Efficiently serve embedded static assets

4.2 Architecture Integration

$$\begin{tikzcd}
\text{Web Browser} \arrow[d, "\text{HTTP GET}"] \\
\text{Web Server Task} \arrow[d, "\texttt{obd\_get\_all\_data()}"] \\
\text{OBD Scheduler Task} \arrow[d, "\text{CAN HAL}"] \\
\text{TWAI Driver}
\end{tikzcd}$$

The web layer never directly interacts with CAN or the TWAI driver. It accesses telemetry exclusively through the OBD data API.

4.3 Embedded Static Resources

4.3.1 Binary Embedding Strategy

Static files are embedded into flash memory using the CMake binary embedding mechanism:

```c
extern const uint8_t _binary_<name>_start[];
extern const uint8_t _binary_<name>_end[];
```

Each file is accessed via linker-generated symbols.

4.3.2 Blob Abstraction

```c
typedef struct {
    const uint8_t *start;
    const uint8_t *end;
} blob_t;
```

This abstraction allows:

- Generic file selection
- Length computation via pointer subtraction
- Safe flash memory access

4.4 JSON Data Endpoint: `/data`

4.4.1 Handler Function

```c
static esp_err_t data_handler(httpd_req_t *req)
```

4.4.2 Purpose

Returns the current vehicle telemetry snapshot in JSON format.

4.4.3 Data Source

```c
obd_full_data_t d = obd_get_all_data();
```

This function retrieves a thread-safe copy of the most recent dataset.

4.4.4 JSON Construction

The JSON response is built using `snprintf()` into a fixed-size buffer:

```c
char resp[768];
```

4.4.5 Safety Mechanism

Overflow detection:

```c
if (len < 0 || len >= sizeof(resp))
```

If overflow is detected:

- A 500 error is returned
- An error is logged

4.4.6 Cache Control

To ensure real-time updates:

- Cache-Control: no-cache
- Pragma: no-cache
- Expires: 0

This prevents stale data visualization.

4.4.7 Thread Safety

Since `obd_get_all_data()` returns a copy protected by a mutex, no race condition exists between the Web Server task and the OBD scheduler.

4.5 Static File Handling

4.5.1 URI Resolution

Function:

```c
static blob_t get_blob_for_uri(const char *uri, const char **mime_out)
```

Responsibilities:

- Match requested URI
- Select embedded binary
- Assign correct MIME type

Supported content types:

- HTML
- JavaScript
- CSS
- WOFF2 fonts

4.5.2 MIME Management

Examples:

- `text/html`
- `application/javascript`
- `text/css`
- `font/woff2`

Proper MIME typing ensures correct browser rendering.

4.6 Static Content Handler

4.6.1 Function

```
static esp_err_t static_handler(httpd_req_t *req)
```

4.6.2 File Size Strategy

For files larger than 4 KB:

```
if (len > 4096)
```

Chunked transmission is used:

```
httpd_resp_send_chunk(...)
```

4.6.3 Rationale

- Avoids large memory copies
- Prevents stack pressure
- Improves TCP reliability
- Suitable for font and JS files

4.6.4 Small Files

Files smaller than 4 KB are sent in a single call:

```
httpd_resp_send(...)
```

4.7 Server Initialization

4.7.1 Function

```c
void web_server_start(void)
```

4.7.2 Configuration

- Default HTTPD configuration
- Stack size: 10240 bytes
- Wildcard URI matching enabled

4.7.3 Registered Endpoints

- `/data` → JSON telemetry
- `/*` → Static content

4.7.4 Task Model

The HTTP server runs as an independent FreeRTOS task. It does not block the OBD scheduler.

4.8 Memory Considerations

4.8.1 RAM Usage

- JSON buffer: 768 bytes
- HTTP stack: 10 KB

4.8.2 Flash Usage

All static resources are stored in flash memory, avoiding runtime filesystem overhead.

4.9 Real-Time Impact Analysis

The Web Server:

- Does not interact with CAN directly
- Does not hold mutex longer than necessary
- Does not allocate dynamic memory per request

Therefore:

- No interference with real-time OBD polling
- Deterministic system behavior preserved

4.10 Internal Data Access API

While the primary role of the Web Server is to serve JSON to connected clients, it also acts as the central repository for the latest normalized vehicle data. To support local peripherals (such as the LCD display), the module exposes specific getter functions that return the current values formatted for display.

4.10.1 Available Getters

These functions provide direct access to the values currently held in the JSON object structure:

- `int32_t get_rpm(void)`: Returns the current Engine RPM.
- `float get_battery(void)`: Returns the battery voltage (OBDII input voltage) in Volts.
- `int32_t get_temp(void)`: Returns the Engine Coolant Temperature in Degrees Celsius.

These accessors ensure that local hardware interfaces (e.g., the LCD task) display data consistent with the web dashboard.

4.11 Local Web Interface

4.11.1 General Architecture

The system exposes a local web interface hosted directly on the ESP32 via the embedded HTTP server. The architecture consists of:

- **index.html** – dashboard structure
- **style.css** – layout and dynamic indicators
- **script.js** – polling logic, JSON parsing, and DOM updates
- **/data** – REST endpoint providing OBD-II telemetry in JSON format

The browser connects to the ESP32 local IP and loads the interface. The JavaScript script periodically fetches the `/data` endpoint to update vehicle parameters in real-time.

4.11.2 Communication Model

The adopted model is:

Periodic HTTP Polling

Every 200 ms, the client performs:

```
GET /data
```

The server responds with a JSON object containing the current telemetry values.

JSON Structure

Example response:

```
{
  "rpm": 820,
  "speed": 0,
  "load": 18.4,
  "throttle": 3.1,
  "timing": 12.5,
  "maf": 2.34,
  "temp_coolant": 88,
  "temp_intake": 27,
  "batt": 13.82,
  "fuel_lvl": 54.2,
  "fuel_press": 312.4,
  "fuel_trim_s": -1.2,
  "fuel_trim_l": 0.8,
  "press_intake": 98,
  "press_baro": 101.3,
  "dist_mil": 0,
  "dtc_count": 0,
  "pending_dtc": 0,
  "uptime_s": 123
}
```

4.11.3 script.js – Software Architecture

The file `script.js` is structured as an IIFE module:

```
(() => {
  "use strict";
  ...
})();
```

This ensures:

- Scope isolation
- No global variables
- Increased safety against naming collisions

Configuration Constants

- POLL_MS = 200 ms
- RPM_MAX = 8000
- SPEED_MAX = 240 km/h
- COOLANT_MIN / MAX
- BATT_MIN / MAX
- MAP_MIN / MAX

These values normalize graphical indicators to percentages.

Robust JSON Parsing

The helper function:

```
pick(obj, keys, defVal)
```

selects the first available key from a list. This ensures future backend compatibility without modifying the frontend.

Example:

```
const coolant = Number(pick(d,
    ["temp_coolant", "coolant"], NaN));
```

Safe DOM Updates

Each DOM modification checks for element existence:

```
if (el) el.textContent = txt;
```

This guarantees robustness even if some HTML elements are missing.

4.11.4 KPI Updates

RPM and Speed

Values are converted to percentages:

$$\text{percentage} = \frac{\text{value}}{\text{maximum value}} \times 100$$

Percentages control:

- Overlay bar
- Illumination
- Cursor position

Dynamic Trends

Percent variation is computed as:

$$\Delta\% = \frac{current - previous}{previous} \times 100$$

Displayed with icons:

- Up arrow
- Down arrow
- Stable

4.11.5 Temperature Management

Coolant temperature is normalized between:

$$T_{min} = 50°C$$

$$T_{max} = 130°C$$

and converted to a graphical percentage for display.

4.11.6 Battery Management

Voltage is mapped between 10V and 15V.

A simple heuristic determines battery status:

- $V_{CAR} < 12V \rightarrow$ Status Low
- $12V <= V_{CAR} <= 17.5V \rightarrow$ Status OK
- $V_{CAR} > 17.5V \rightarrow$ Status High

4.11.7 System Statistics

The interface displays:

- RTT (Round Trip Time)
- Update rate
- Uptime
- DTC count

The approximate update rate is calculated as:

$$rate = \frac{samples}{elapsed\_time}$$

4.11.8 Network Error Handling

On fetch error:

```
setConnected(false);
```

The UI displays a disconnected state without blocking the interface.

4.11.9 Compatibility with web_server.c

The REST endpoint:

```
/data
```

is fully compatible with the JavaScript script.

No additional dependencies are required:

- WebSocket
- CORS
- Authentication
- Compression

JSON payload size is safe for the ESP32 RAM.

4.11.10 Embedded Considerations

- Polling at 200 ms $\rightarrow$ 5 requests/sec
- Compact JSON
- No dynamic allocation on the client
- No heavy computation on ESP32

Thus, the system is suitable for resource-limited microcontrollers.

4.11.11 Potential Future Improvements

- Implement WebSocket to reduce HTTP overhead
- Enable gzip compression
- Local caching of values
- Dark/light mode
- Historical data visualization

4.11.12 Conclusion

The Web interface provides:

- Real-time visualization of vehicle data
- Robust handling of missing data
- Full compatibility with the embedded backend
- Modular and maintainable architecture

The interface is efficient, scalable, and suitable for automotive ESP32-based embedded systems.

4.12 Web Interface Overview

The web interface of the OBD-II Monitoring System is organized into multiple HTML pages, supported by JavaScript modules for real-time data visualization, diagnostics, and user interaction.

4.12.1 Graph Page

The `graph.html` page provides real-time charts for vehicle telemetry. It includes:

- **Navigation Bar:** Links to the dashboard, charts, and errors pages, with export options.
- **Charts:** Separate chart cards for RPM & Speed, Temperatures, Pressures, Fuel System, Electrical System, and Performance.
- **Controls:** Options to select time intervals and choose specific charts to export.
- **Statistics:** Advanced statistics and trend analysis, including average values, maximums, and fuel efficiency trends.

The page relies on `chart-script.js` to manage chart rendering and dynamic updates.

4.12.2 Chart Script

The `chart-script.js` module handles the following:

- Rendering of charts using `Chart.js`.
- Dynamic updates of datasets according to selected parameters.
- Real-time integration with the data source or simulation.
- User interaction for exporting chart images.

4.12.3 Diagnostics Page

The `diagnostics.html` page provides a professional diagnostic view, including:

- **Vehicle Status:** Connection status, ECU details, protocol, and VIN.
- **DTC Overview:** Counts of critical, warning, and total Diagnostic Trouble Codes.
- **Monitor Readiness:** Status of emission system monitors.
- **Freeze Frame Data:** Snapshot of vehicle parameters at the time of faults.
- **Charts:** Trends of fuel, ignition, emissions, and engine parameters.

4.12.4 Diagnostics Script

The `diagnostics-script.js` module manages:

- Fetching and simulating diagnostic data from multiple sources.
- Dynamic population of DTC cards, readiness cards, and freeze frame tables.
- Real-time chart updates reflecting diagnostic parameter trends.
- User actions such as scanning all systems, clearing codes, running readiness tests, and exporting data.
- Modal windows for detailed DTC information.

4.12.5 Animations and Interactivity

Animations, transitions, and UI interactions across pages are handled by `script.js` and integrated with the AnimationEngine v3.0. It provides:

- Smooth hover and click effects for cards and buttons.
- Dynamic updates of counters, charts, and modals without reloading pages.
- Mobile-friendly toggles and responsive design adjustments.

4.12.6 Errors Documentation Page

The `errors.html` page documents standard OBD-II error codes and descriptions. It includes:

- Structured listings of DTC codes categorized by system (Powertrain, Emissions, Fuel, etc.).
- Severity labels and brief explanations.
- Navigation and search capabilities for easy reference.

4.12.7 Summary

Overall, the web system integrates multiple pages, scripts, and charts to provide a comprehensive, real-time monitoring and diagnostic interface for OBD-II vehicles. Data sources can be real, simulated, or generated by embedded simulators, ensuring flexibility for development and testing.

Chapter 5

LCD Display Module

5.1 Overview

The system includes a local hardware interface utilizing a 7-pin, 2-row character LCD display. This module provides immediate visual feedback to the driver regarding critical vehicle parameters and connection status, without requiring a web browser connection.

The implementation is contained within:

- `lcd/lcd.c`: Driver implementation and string mapping logic.
- `lcd/lcd.h`: Public interface definitions.

5.2 Driver Implementation

The `lcd.c` file manages the physical communication with the display hardware. It abstracts the cursor positioning and string writing operations.

5.2.1 Initialization

The module is initialized via the `lcd_init()` function. This function must be called during the system startup phase in `app_main.c`. It configures the GPIO pins, initializes the communication bus, and prepares the display buffer.

5.2.2 String Mapping

The driver provides specialized functions to map strings to specific coordinates on the 2×16 (or similar) grid. This ensures that labels (such as "RPM", "BAT") and their corresponding dynamic values remain aligned and readable.

5.3 Integration with Main Application

The display is updated by a dedicated FreeRTOS task defined in `app_main.c`. This task ensures the display is refreshed at a human-readable rate without blocking critical CAN or network tasks.

5.3.1 Display Task Logic

The task is created via `xTaskCreate` and executes a periodic loop every 400 ms.

```c
void lcd_display_task(void *pvParameters)
{
    while (1) {
        // Retrieve latest data from Web Server cache
        int rpm = get_rpm();
        float voltage = get_battery();
        int temp = get_temp();
        int clients = get_wifi_client_count();

        // Format strings
        char line1[16];
        char line2[16];

        snprintf(line1, sizeof(line1), "AP:%d T:%dC", clients, temp);
        snprintf(line2, sizeof(line2), "V:%0.1f R:%d", voltage, rpm);

        // Update LCD
        lcd_set_cursor(0, 0);
        lcd_print_string(line1);
        lcd_set_cursor(0, 1);
        lcd_print_string(line2);

        // 400ms Refresh Rate
        vTaskDelay(pdMS_TO_TICKS(400));
    }
}
```

5.3.2 Displayed Metrics

The screen layout is designed to show the most vital information compactly:

1. **Connected Clients (AP):** Indicates how many devices are currently connected to the ESP32 Access Point.
2. **Temperature:** Engine Coolant Temperature in degrees Celsius (°C), retrieved via `get_temp()`.
3. **Input Voltage:** The vehicle battery voltage (e.g., 12.5V), retrieved via `get_battery()`.
4. **RPM:** Real-time Engine revolutions per minute, retrieved via `get_rpm()`.

5.4 Data Flow Summary

The LCD module functions as a "passive consumer" in the system architecture:

$$\text{CAN Bus} \rightarrow \text{OBD Data} \rightarrow \text{Web Server Cache} \xrightarrow{\text{Getters}} \text{Main Task} \rightarrow \text{LCD Driver}$$

This architecture ensures that the values shown on the physical screen are mathematically identical to those transmitted via JSON to the web interface.