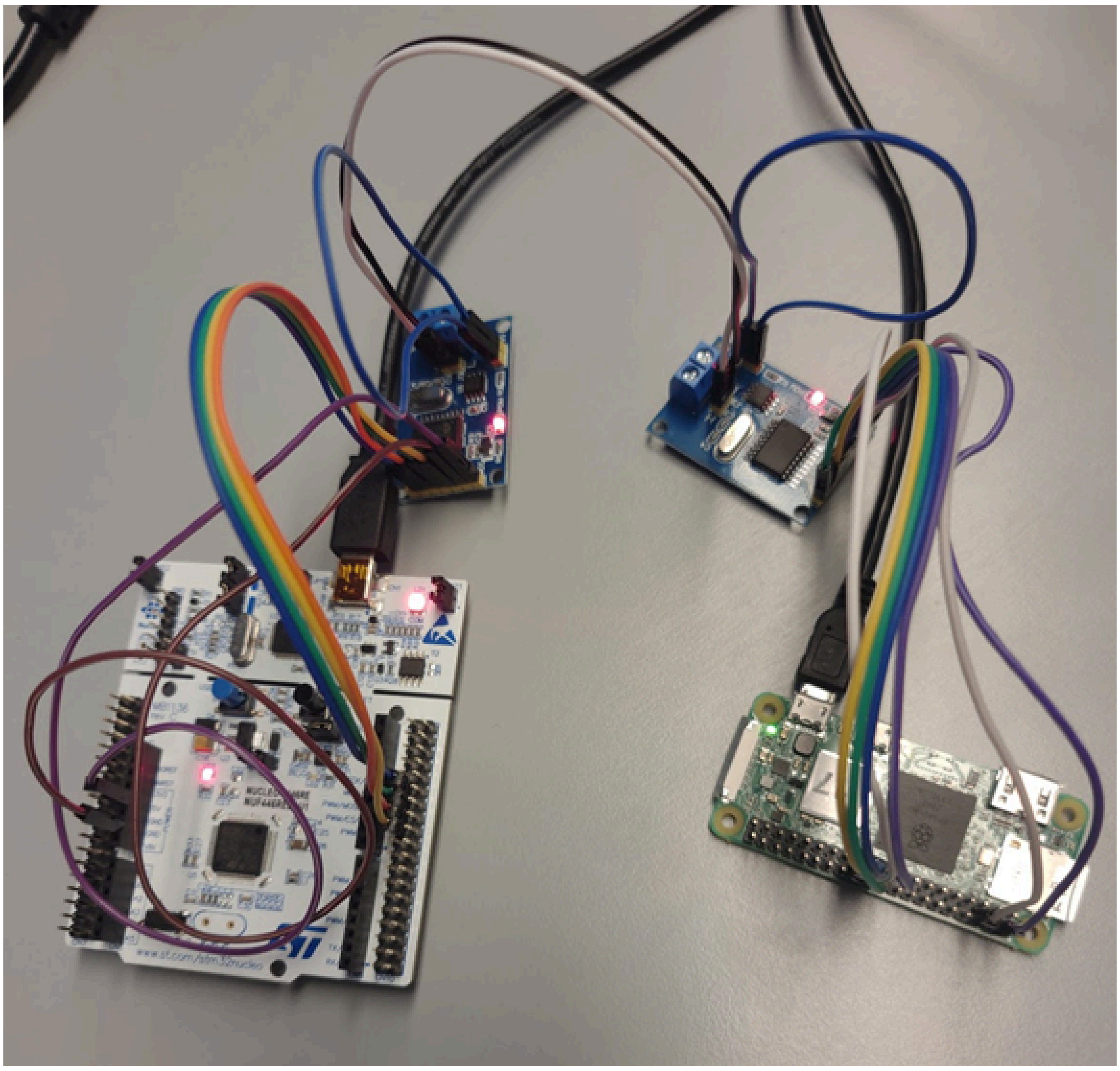
Student: Niklas Brodén

CAN Traffic Simulator and Driver Information Module (DIM) Project

STM32 + MCP2515/TJA1050 + Raspberry Pi Dashboard



Introduction

This project sets up a complete CAN bus test environment, simulating vehicle communication between an STM32-based node and a Raspberry Pi dashboard. It is built as a Driver Information Module (DIM) prototype, similar to the systems used in modern car dashboards.

The STM32 acts as an electronic control unit (ECU), sending out signals such as light indicators, sensor readings, and status messages. The Raspberry Pi runs a web-based dashboard that receives these messages and allows the user to send commands back to the STM32.

The main goal is to provide a practical and modular platform for learning, prototyping, and testing CAN-based embedded systems.

Hardware Components

- STM32 Nucleo-F446RE board
- MCP2515 CAN controller module
- TJA1050 CAN transceiver
- Raspberry Pi Zero 2

STM32 Firmware Overview

The STM32 firmware sets up all hardware interfaces and configures the MCP2515 into normal CAN operation. It performs the following tasks:

- Sends periodic CAN messages (heartbeat, status, sensor data)
- Detects button state changes and transmits them
- Receives commands from the Raspberry Pi (e.g., LED toggle)
- Sends feedback via CAN (e.g., LED status)
- Logs activity to UART
- Uses a ring buffer for safe, non-blocking CAN reception
- Supports both polling and interrupt-based message handling via EXTI3 (PB3)

Raspberry Pi Dashboard Overview

The Raspberry Pi hosts a Flask-based web interface, simulating the DIM UI:

- Displays the latest CAN messages with labels and timestamps
- Logs all traffic to CSV for traceability
- Sends control commands (e.g., LED ON) to the STM32
- Uses AJAX polling for smooth live updates
- Receives CAN frames in the background using `python-can`

Project Structure

The embedded application runs on an STM32 Nucleo board and uses an MCP2515 CAN controller over SPI. Key files:

STM32 Firmware (`stm32/`)

- **`src/`**
    - **`main.c`** — Main application loop and CAN message logic
    - **`utils.c`, `utils.h`** — Utility functions for CAN, UART, and test message helpers
    - **`can_buffer.c`, `can_buffer.h`** — Ring buffer for received CAN frames
    - **`interrupts.c`, `interrupts.h`** — EXTI interrupt setup and handler for MCP2515 INT pin (PB3)
    - **`spi.c`** — SPI1 peripheral and MCP2515 chip select (CS) initialization
    - **`uart.c`, `uart.h`, `uart_log.h`** — UART2 initialization, logging, and print helpers
    - **`clock.c`** — System clock configuration
    - **`gpio.c`** — GPIO initialization (LED, user button, INT pin)
    - **`stm32f4xx_it.c`, `stm32f4xx_it.h`** — Core interrupt handlers (SysTick, EXTI3)

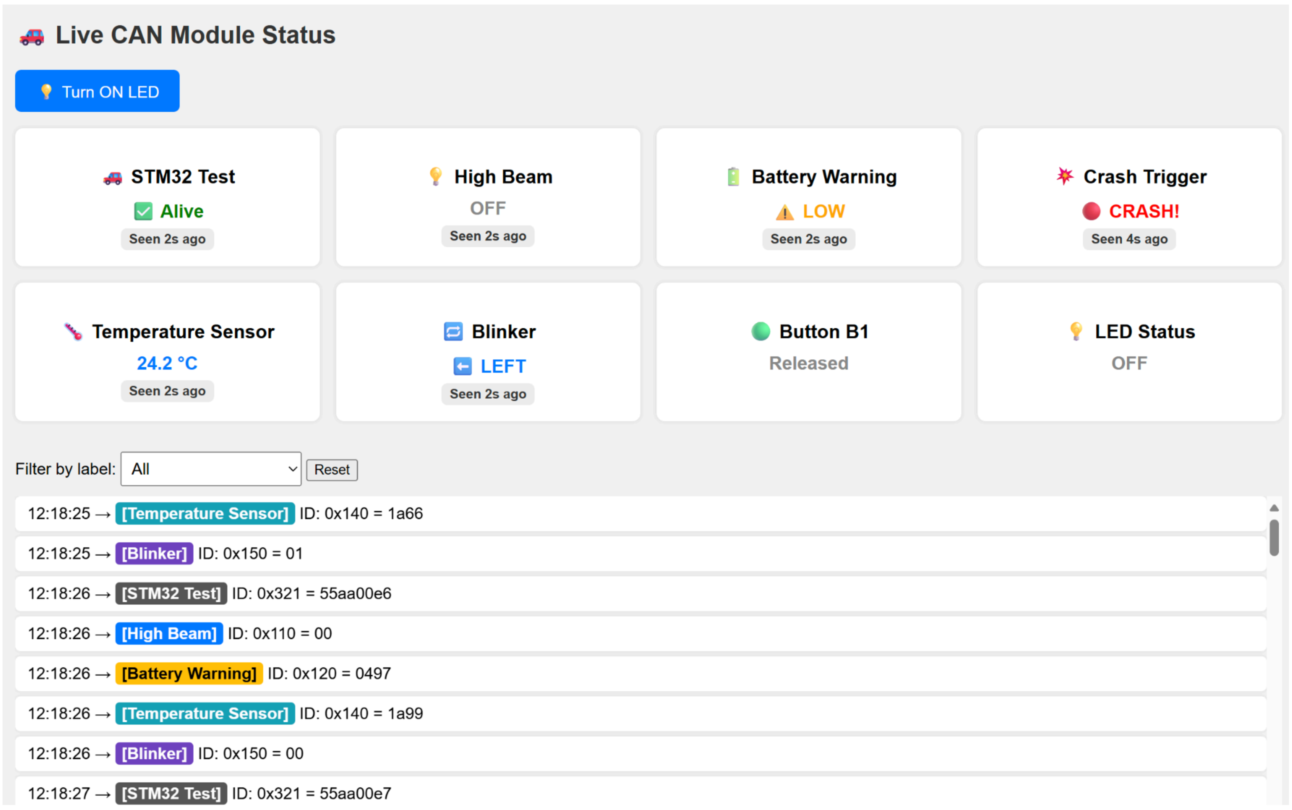
Raspberry Pi Dashboard (`pi_can_dashboard/`)

- **`app.py`** — Flask web app for CAN interface, live dashboard, and control endpoints
- **`templates/`**
    - **`index.html`** — HTML template for the dashboard user interface

---

Dashboard

The web-based dashboard, built using Flask + AJAX, runs on the Raspberry Pi and displays live status updates from the STM32 via the CAN bus. Each tile represents a signal or sensor value, updated every 500 ms via background polling.



🚗 Live CAN Module Status

💡 Turn ON LED

🚗 STM32 Test ✅ Alive Seen 2s ago	💡 High Beam OFF Seen 2s ago	🔋 Battery Warning ⚠️ LOW Seen 2s ago	💥 Crash Trigger 🔴 CRASH! Seen 4s ago
🌡️ Temperature Sensor 24.2 °C Seen 2s ago	🔄 Blinker ⬅️ LEFT Seen 2s ago	🟢 Button B1 Released	💡 LED Status OFF

Filter by label: All Reset

```
12:18:25 → [Temperature Sensor] ID: 0x140 = 1a66
12:18:25 → [Blinker] ID: 0x150 = 01
12:18:26 → [STM32 Test] ID: 0x321 = 55aa00e6
12:18:26 → [High Beam] ID: 0x110 = 00
12:18:26 → [Battery Warning] ID: 0x120 = 0497
12:18:26 → [Temperature Sensor] ID: 0x140 = 1a99
12:18:26 → [Blinker] ID: 0x150 = 00
12:18:27 → [STM32 Test] ID: 0x321 = 55aa00e7
```

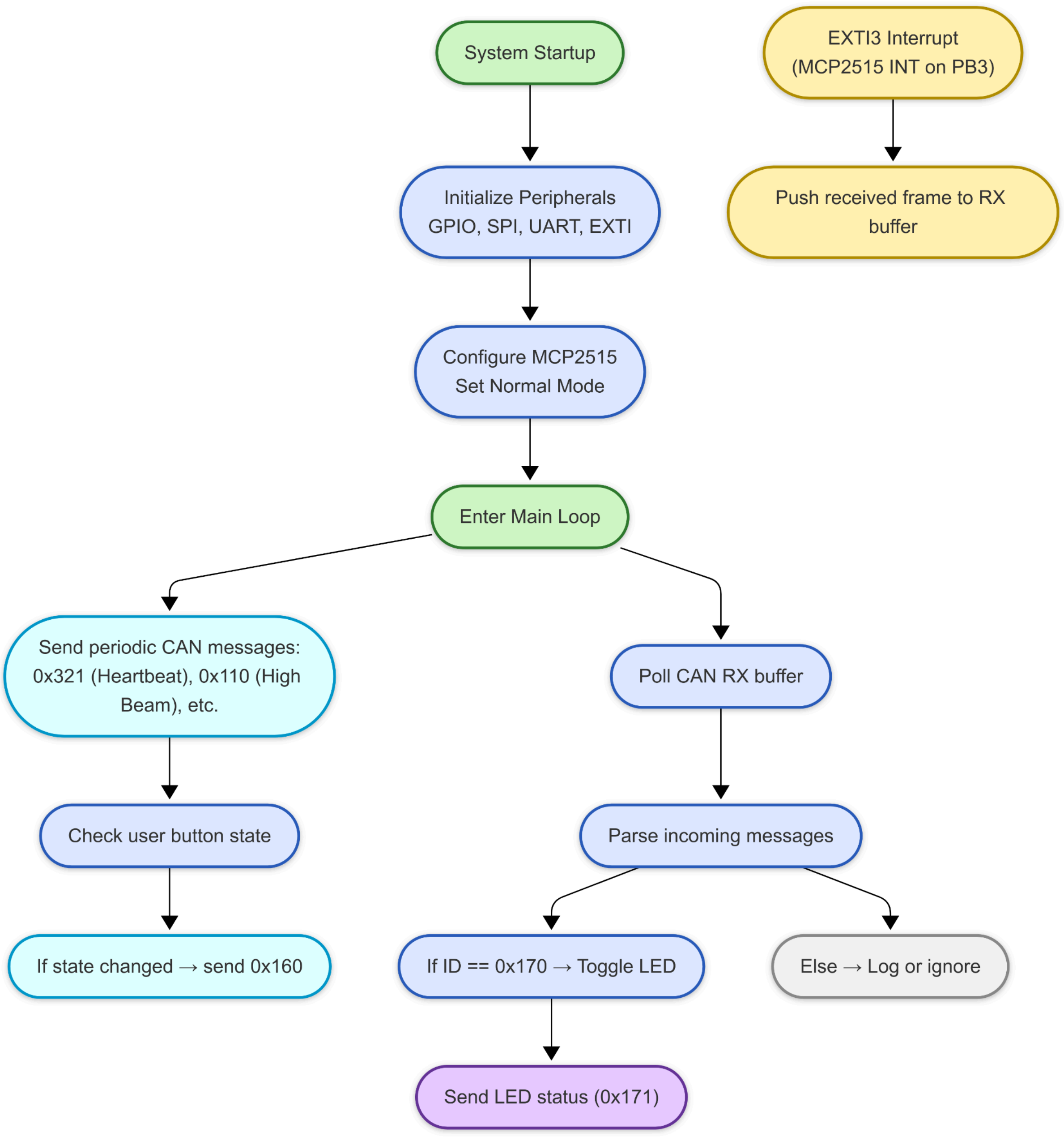
- The **top section** shows the latest interpreted values from messages such as heartbeat, high beam, temperature, and crash triggers.
- The **log section below** captures raw CAN traffic in real time, displaying message timestamps, IDs, and decoded labels.
- A **“Turn ON LED”** button sends a `0x170` CAN command to the STM32, which responds with a `0x171` status message reflecting the LED state.

CAN Message Mapping

CAN ID	Description	Direction	Purpose
0x321	STM32 Heartbeat/Test	STM32 → CAN	Periodic system status
0x110	High Beam Status	STM32 → CAN	Alternates ON/OFF
0x120	Battery Voltage	STM32 → CAN	Simulated voltage warning
0x130	Crash Trigger	STM32 → CAN	Occasional crash event
0x140	Temperature Sensor	STM32 → CAN	Simulated temperature reading
0x150	Blinker Status	STM32 → CAN	Left/Right toggle indicator
0x160	Button B1 State	STM32 → CAN	Reports user button status
0x170	LED Control	Pi → STM32	Command to toggle LED
0x171	LED Status Response	STM32 → Pi	Acknowledges LED state

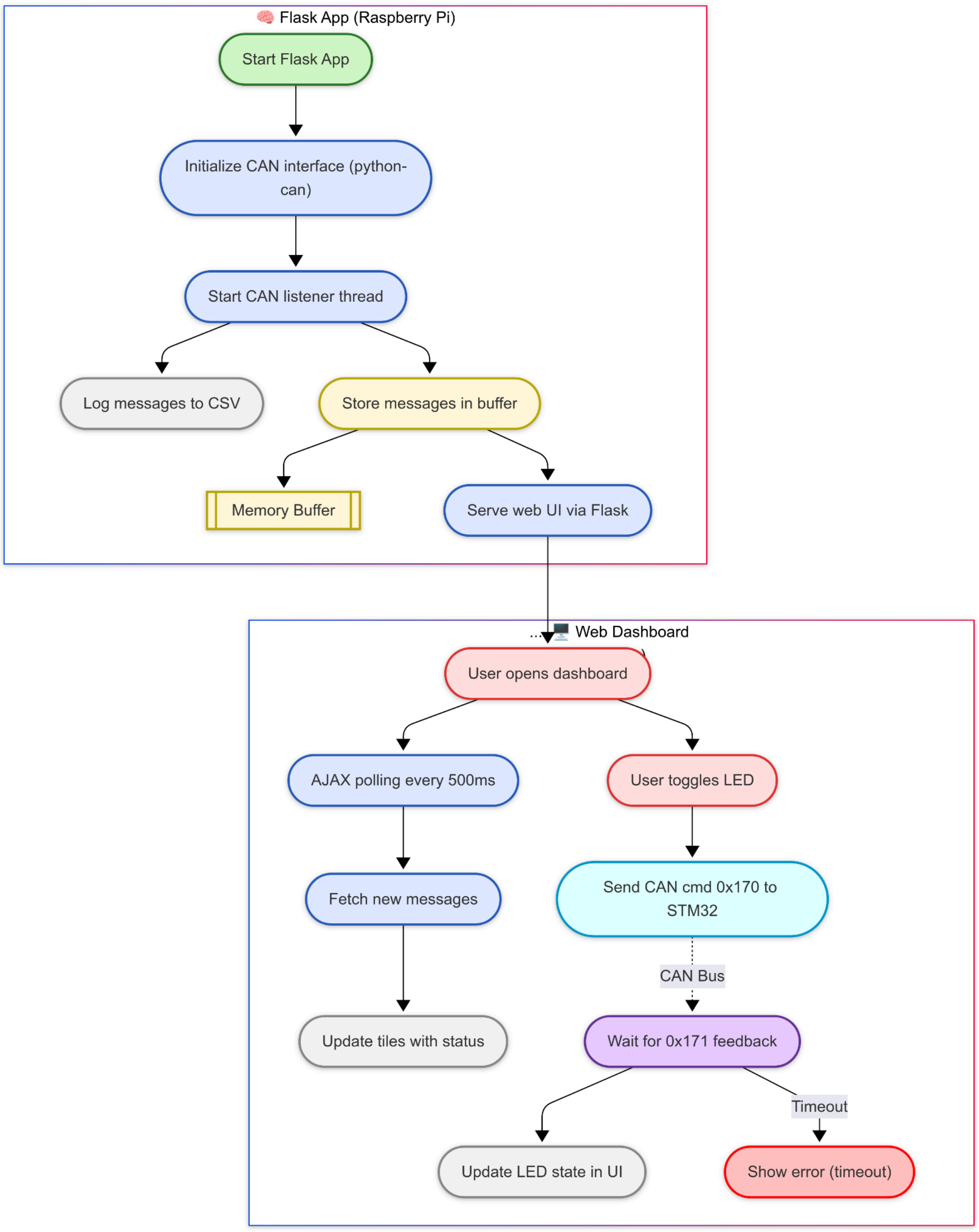
STM32 Flowchart: CAN Communication Overview

This diagram illustrates the STM32 firmware logic for CAN-based communication. Upon startup, all peripherals and the MCP2515 CAN controller are initialized. The main loop periodically transmits status messages (e.g., heartbeat, high beam) and checks for user input. Incoming CAN frames are processed either through polling or interrupts (EXTI3 from MCP2515), with specific IDs triggering actions such as toggling the LED and sending feedback. A ring buffer ensures safe message handling even at higher traffic rates.


System Startup
EXTI3 Interrupt (MCP2515 INT on PB3)
Initialize Peripherals GPIO, SPI, UART, EXTI
Push received frame to RX buffer
Configure MCP2515 Set Normal Mode
Enter Main Loop
Send periodic CAN messages: 0x321 (Heartbeat), 0x110 (High Beam), etc.
Poll CAN RX buffer
Check user button state
Parse incoming messages
If state changed → send 0x160
If ID == 0x170 → Toggle LED
Else → Log or ignore
Send LED status (0x171)

Raspberry Pi Flowchart

This flowchart outlines the backend (Flask + python-can) and frontend (browser-based UI) communication. A background thread listens to CAN traffic, stores recent messages, and serves data via Flask. The dashboard polls periodically to stay updated, and allows user interaction like toggling an LED via CAN.



Conclusion

This project demonstrates a complete and functional CAN bus communication system between an STM32 microcontroller and a Raspberry Pi. The STM32 acts as an embedded ECU that sends and receives messages, while the Raspberry Pi provides a live web dashboard for monitoring and control.

The system is modular, well-structured, and easy to follow. It combines clean embedded C code with a user-friendly Python Flask application. This makes it a great learning tool for anyone interested in automotive communication, CAN protocols, or embedded system testing.

.

Source code

brodenn/canbus_project