Controller Area Network (CAN) Protocol

Introduction

In today's world of modern vehicles and machines, different electronic systems need to communicate with each other. For example, the engine, brakes, and airbags in a car all have to work together. To make this communication easy and reliable, we use something called the CAN protocol (Controller Area Network). It allows devices (called nodes) to share data using just two wires.

Background

The CAN protocol was developed by Bosch in 1986 for use in the automotive industry. The goal was to reduce the number of wires in vehicles and make communication between parts faster and more reliable. CAN was standardized as ISO 11898. Today, CAN is not only used in cars but also in factories, robots, and even medical devices. It has become one of the most successful network protocols in the field of embedded systems.

Features of CAN

- **Uses two wires (CAN High and CAN Low) for differential signaling**
- **Supports multi-master communication (any node can send data)**
- **Message-based protocol: nodes communicate by sending messages identified by unique IDs**
- **Built-in error detection and fault confinement mechanisms**
- **High noise immunity due to differential signaling**
- **Supports data rates up to 1 Mbps (CAN 2.0) and 5 Mbps or more (CAN FD)**
- **Priority-based arbitration using message identifiers**

How Does CAN Work?

Imagine a classroom where any student can talk, but only one speaks at a time. CAN works like that. Each message has an identifier (ID). The message with the lowest numerical ID wins the right to use the bus. This is called bitwise arbitration.

When a device sends a message:

1. **It broadcasts the message with a unique ID.**
2. **All other devices receive it.**
3. **Each device decides if the message is relevant to it.**

CAN is half-duplex, which means data can travel in both directions, but only one message can be on the bus at a time. This helps prevent collisions and ensures message integrity.

CAN Message Format

A CAN message (called a frame) includes the following fields:

- **Start of Frame (SOF): Marks the beginning of the message**
- **Identifier (ID): Determines the priority of the message**
- **Remote Transmission Request (RTR): Distinguishes between data and remote frames**
- **Control Field: Includes the Data Length Code (DLC)**
- **Data Field: Actual data (0 to 8 bytes for CAN 2.0, up to 64 bytes for CAN FD)**
- **CRC Field: Used for error checking**
- **ACK Field: Indicates successful message reception**
- **End of Frame (EOF): Marks the end of the message**

Advantages of CAN

- **Reduces wiring compared to point-to-point systems**
- **Reliable and robust communication**
- **Efficient use of bandwidth with prioritized messages**
- **Real-time performance with minimal latency**

- **Automatically retransmits corrupted messages**
- **Easy to expand with more nodes without major changes**

Where is CAN Used?

- **Automotive Systems: Engine control, ABS, airbags, power windows, infotainment**
- **Industrial Automation: Machine control, sensors, actuators**
- **Medical Equipment: Imaging systems, infusion pumps**
- **Aerospace: Avionics, control systems**
- **Robotics: Communication between sensors and controllers**
- **Marine Systems: Ship engine monitoring, navigation systems**

Limitations of CAN

- **Maximum data size in standard CAN is 8 bytes, which is small**
- **Bandwidth is limited compared to modern Ethernet-based protocols**
- **Not suitable for long-distance, high-speed communication without repeaters**
- **Half-duplex nature limits simultaneous communication**
- **Requires transceivers and additional hardware setup**

CAN Variants

- **CAN FD (Flexible Data Rate): Allows larger data payloads (up to 64 bytes) and faster transmission**
- **CANopen: Application layer protocol for industrial systems**
- **SAE J1939: Used in heavy-duty vehicles like trucks and buses**
- **DeviceNet: Used in factory automation**

These variants build on standard CAN to offer more advanced features like addressing, diagnostics, and standardized data formats.

Simple Example Project

Two Arduino boards using MCP2515 CAN modules can be connected to send messages to each other.

Example Setup:

- **Arduino 1 sends a message: "Temperature = 25°C"**
- **Arduino 2 receives the message and displays it on an LCD screen**

Hardware Needed:

- **2x Arduino Uno**
- **2x MCP2515 CAN modules**
- **Jumper wires**
- **Common ground**
- **CAN_H and CAN_L lines connected between modules**

Code (Sender):

```
CAN.sendMsgBuf(0x101, 0, 1, data); // Send temperature data
```

Code (Receiver):

```
CAN.readMsgBuf(&rxId, &len, buf);

if (rxId == 0x101) {

  // Display received temperature

}
```

Real-World Project: Bike Lighting System Using CAN

I completed an automotive-style CAN bus project using two ESP32 boards and MCP2515 CAN modules. This setup simulated a bike's Electronic Control Unit (ECU)-based lighting system.

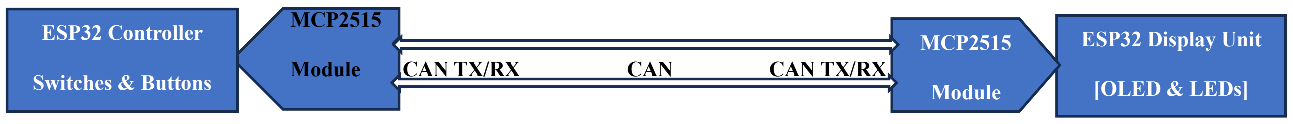
- **ESP32 Controller Unit: Connected to DIP switches and push buttons to simulate user inputs for left/right indicators, headlight, and brake light.**

- **ESP32 Display Unit: Connected to an OLED screen and physical LEDs to show the status of each lighting function based on the received CAN messages.**

How it Works:

1. **When a switch or button is pressed, the controller ESP32 sends a CAN message with a unique ID.**
2. **The display ESP32 listens for messages and updates the OLED and LEDs accordingly.**

System Diagram:



Example CAN Message Table:

Action	**CAN ID**	**Data**
Left Indicator	**0x101**	**0x01**
Right Indicator	**0x102**	**0x01**
Headlight	**0x103**	**0x01**
Brake Light	**0x104**	**0x01**

This project helped me understand real-time control and monitoring using CAN. It also demonstrated how CAN can be applied in vehicle systems for both control and diagnostics.

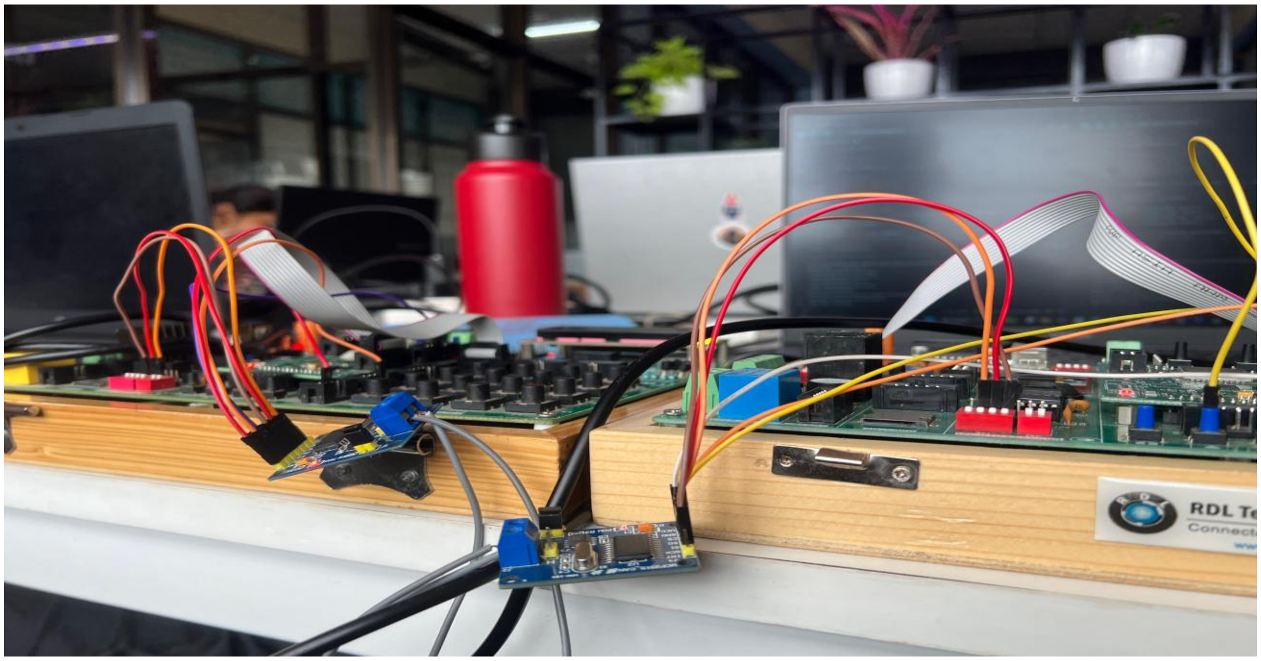
<u>ESP32 #1 – Controller (Switch Input & CAN Sender)</u>

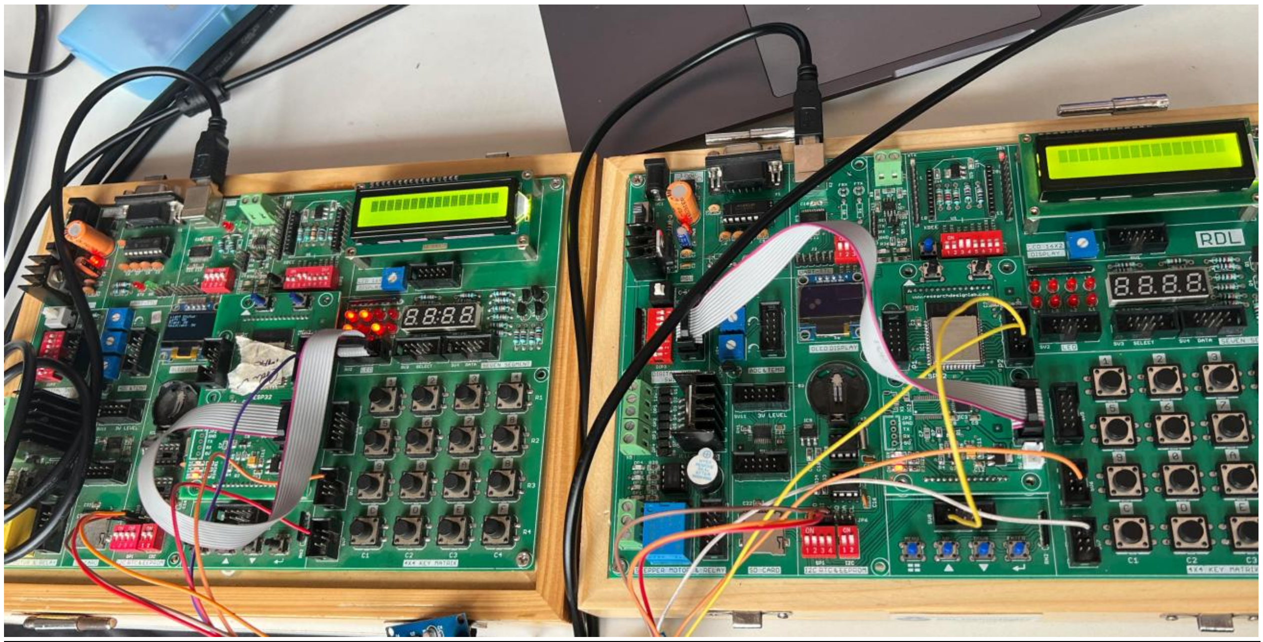
```
#include <mcp_can.h>
#include <SPI.h>

// MCP2515 Setup
#define SPI_CS 5
MCP_CAN CAN(SPI_CS);  // SPI CS pin
```

```
// Input pins
#define LEFT_IND_PIN 14
#define RIGHT_IND_PIN 15
#define HEADLIGHT_PIN 27
#define BRAKE_PIN 16

void setup() {
  Serial.begin(115200);
  while (CAN_OK != CAN.begin(MCP_ANY, CAN_500KBPS, MCP_8MHZ)) {
    Serial.println("CAN init failed");
    delay(1000);
  }
  CAN.setMode(MCP_NORMAL);
  Serial.println("CAN init success");

  pinMode(LEFT_IND_PIN, INPUT_PULLUP);
  pinMode(RIGHT_IND_PIN, INPUT_PULLUP);
  pinMode(HEADLIGHT_PIN, INPUT_PULLUP);
  pinMode(BRAKE_PIN, INPUT_PULLUP);
}

void loop() {
  sendSwitchState(0x101, digitalRead(LEFT_IND_PIN));
  sendSwitchState(0x102, digitalRead(RIGHT_IND_PIN));
  sendSwitchState(0x103, digitalRead(HEADLIGHT_PIN));
  sendSwitchState(0x104, digitalRead(BRAKE_PIN));
  delay(200);
}

void sendSwitchState(int can_id, int state) {
  byte data[1];
  data[0] = (state == LOW) ? 0x01 : 0x00; // Active LOW
  CAN.sendMsgBuf(can_id, 0, 1, data);
}
```

ESP32 #2 – Receiver (LED + OLED Display)

```
#include <mcp_can.h>
#include <SPI.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

// OLED Setup
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, -1);

// MCP2515 Setup
#define SPI_CS 5
MCP_CAN CAN(SPI_CS);

// PORT-3 Output pins for LEDs
#define LEFT_LED 14
#define RIGHT_LED 15
#define HEADLIGHT_LED 27
#define BRAKE_LED 13

bool leftState = false, rightState = false, headlightState = false, brakeState = false;

void setup() {
  Serial.begin(115200);

  pinMode(LEFT_LED, OUTPUT);
  pinMode(RIGHT_LED, OUTPUT);
  pinMode(HEADLIGHT_LED, OUTPUT);
  pinMode(BRAKE_LED, OUTPUT);

  display.begin(SSD1306_SWITCHCAPVCC, 0x3C);
  display.clearDisplay();
  display.display();

  while (CAN_OK != CAN.begin(MCP_ANY, CAN_500KBPS, MCP_8MHZ)) {
    Serial.println("CAN init failed");
    delay(1000);
  }
  CAN.setMode(MCP_NORMAL);
  Serial.println("CAN init success");
}

void loop() {
  if (CAN_MSGAVAIL == CAN.checkReceive()) {
    unsigned long id; id;
    byte len;
    byte buf[8];
    CAN.readMsgBuf(&id, &len, buf);
    updateState(id, buf[0]);
    updateDisplay();
  }
}

void updateState(long id, byte value) {
  bool state = (value == 0x01);
  switch (id) {
    case 0x101:
      leftState = state;
      digitalWrite(LEFT_LED, state);
      break;
    case 0x102:
      rightState = state;
      digitalWrite(RIGHT_LED, state);
      break;
    case 0x103:
      headlightState = state;
      digitalWrite(HEADLIGHT_LED, state);
      break;
    case 0x104:
      brakeState = state;
      digitalWrite(BRAKE_LED, state);
      break;
  }
}

void updateDisplay() {
  display.clearDisplay();
  display.setTextSize(1);
  display.setTextColor(SSD1306_WHITE);
  display.setCursor(0,0);
  display.println("Light Status:");
  display.print("Left: "); display.println(leftState ? "ON" : "OFF");
  display.print("Right: "); display.println(rightState ? "ON" : "OFF");
  display.print("Headlight: "); display.println(headlightState ? "ON" : "OFF");
  display.print("Brake: "); display.println(brakeState ? "ON" : "OFF");
  display.display();
}
```





Conclusion

The CAN protocol is a powerful tool for enabling communication between multiple devices in a reliable and efficient way. It is widely used in many critical applications and continues to evolve with technologies like CAN FD. Understanding CAN is essential for students and professionals interested in embedded systems, automotive electronics, and industrial control systems.