CAN Protocol

Introduction

CAN (Controller Area Network) represents a paradigm shift in embedded system communication, evolving from the complex wiring nightmares of early automotive systems to an elegant, robust solution. Developed by **Robert Bosch in the mid-1980s**, CAN has transcended its automotive origins to become a cornerstone technology across diverse industries.

The Problem CAN Solved

Before CAN, automotive systems suffered from:

- **Wire Complexity**: Hundreds of individual point-to-point connections
- **Communication Barriers**: Isolated subsystems unable to share critical information
- **Maintenance Nightmares**: Troubleshooting complex wiring harnesses
- **Scalability Issues**: Adding new features required extensive rewiring

How CAN Protocol Works: Six Core Principles

1. Message-Based Communication Architecture

Unlike traditional address-based protocols, CAN operates on a **content-oriented approach**:

- **Data Packets**: Information transmitted in discrete frames with headers and payloads
- **Broadcast Nature**: Every message reaches all nodes simultaneously
- **Content Filtering**: Nodes decide message relevance based on identifiers
- **No Routing Required**: Eliminates complex routing tables and destination addressing

2. Differential Signaling for Noise Immunity

CAN's electrical robustness stems from its differential signaling implementation:

- **Two-Wire System**: CAN_H and CAN_L carry complementary signals
- **Common-Mode Rejection**: External noise affects both wires equally, canceling out
- **EMI Resistance**: Perfect for harsh industrial and automotive environments
- **Extended Range**: Reliable communication over significant distances

3. Non-Destructive Bitwise Arbitration

CAN's arbitration mechanism ensures collision-free bus access:

- **Simultaneous Transmission**: Multiple nodes can start transmitting together
- **Bit-by-Bit Comparison**: Lower identifier values win arbitration
- **Graceful Degradation**: Losing nodes become receivers without data loss
- **Priority Preservation**: Most critical messages always get through first

4. Comprehensive Error Detection and Handling

Five-layer error detection ensures data integrity:

- **CRC (Cyclic Redundancy Check)**: 15-bit checksum validates data integrity
- **Acknowledgment Mechanism**: Receivers confirm successful message reception
- **Format Checking**: Validates frame structure compliance
- **Bit Monitoring**: Transmitters verify actual transmitted bits
- **Stuff Error Detection**: Ensures synchronization through bit stuffing validation

5. Dual Frame Format Support

CAN accommodates varying system complexity through two frame formats:

Standard Frames (CAN 2.0A):

- **11-bit Identifier**: Supports 2,048 unique message IDs (2^11)
- **Compact Efficiency**: Shorter frames for faster transmission
- **Universal Compatibility**: Supported by all CAN implementations
- **Legacy Integration**: Maintains backward compatibility

Extended Frames (CAN 2.0B):

- **29-bit Identifier**: Supports 536,870,912 unique message IDs (2^29)
- **Scalability**: Accommodates complex, large-scale networks
- **Future-Proofing**: Handles expanding system requirements
- **Mixed Networks**: Can coexist with standard frames

6. Masterless Peer-to-Peer Communication

CAN eliminates single points of failure:

- **Distributed Control**: No central node controls communication
- **Equal Access Rights**: All nodes can initiate communication
- **Fault Tolerance**: Network survives individual node failures
- **Organic Scalability**: Easy addition/removal of network participants

---

Frame Formats and Structure Analysis

Standard CAN Frame Detailed Breakdown

Understanding frame structure is crucial for effective CAN implementation:

Field	Size (bits)	Purpose	Technical Details
SOF (Start of Frame)	1	Synchronization marker	Always dominant (0) - signals frame start
Identifier	11	Message ID and priority	Lower values = higher priority (0x000 highest)
RTR (Remote Transmission Request)	1	Frame type indicator	Data Frame=0 (dominant), Remote Frame=1 (recessive)
IDE (Identifier Extension)	1	Format identifier	Standard=0, Extended=1
R0 (Reserved)	1	Future expansion	Always 0 (dominant)
DLC (Data Length Code)	4	Payload size indicator	0-8 bytes (binary encoded)

Field	Size (bits)	Purpose	Technical Details
Data Field	0-64	Message payload	Application-specific content
CRC Sequence	15	Error detection	Polynomial: x^15 + x^14 + x^10 + x^8 + x^7 + x^4 + x^3 + 1
CRC Delimiter	1	CRC field terminator	Always recessive (1)
ACK Slot	1	Acknowledgment bit	Transmitter=1, Receivers override with 0
ACK Delimiter	1	ACK field terminator	Always recessive (1)
EOF (End of Frame)	7	Frame completion marker	Seven recessive bits (1111111)
IFS (Inter-Frame Space)	3	Frame separation	Minimum gap between frames

Extended CAN Frame Structure

Extended frames provide expanded addressing capability:

- **Base Identifier**: 11 bits (same position as standard frame)
- **SRR (Substitute Remote Request)**: Replaces RTR bit, always recessive
- **IDE Bit**: Set to 1 (recessive) indicating extended format
- **Extended Identifier**: Additional 18 bits
- **RTR Bit**: Remote transmission request for extended frames
- **R1 and R0**: Two reserved bits for future use

Total Identifier Space: 11 + 18 = 29 bits = 536,870,912 unique IDs

Frame Type Classifications

Data Frames

- **Purpose**: Carry actual information between nodes
- **RTR Setting**: Dominant (0) indicates data frame
- **Payload**: 0-8 bytes of application data

- **Usage**: Regular communication, sensor data, control commands

Remote Frames

- **Purpose**: Request data from other nodes
- **RTR Setting**: Recessive (1) indicates remote frame
- **No Data Field**: Only identifier specifies requested information type
- **Response**: Target node sends corresponding data frame
- **Usage**: Polling, event-driven data requests

Error Frames

- **Structure**: 6-bit error flag + 8-bit error delimiter
- **Transmission**: Generated when errors detected during communication
- **Effect**: Destroys current frame, forces retransmission
- **Types**: Active error frames (error active nodes) vs passive error frames (error passive nodes)

Overload Frames

- **Purpose**: Introduce transmission delays when needed
- **Structure**: Similar to error frames (6 + 8 bits)
- **Usage**: Prevent receiver buffer overflow, processing time requests
- **Limitation**: Maximum two consecutive overload frames allowed

---

Electrical Characteristics and Physical Implementation

CAN Bus Voltage Specifications

CAN's differential signaling provides exceptional noise immunity through precise voltage level definitions:

Recessive State (Logic 1 - Bus Idle)

- **CAN_H Voltage**: 2.5V ± 0.05V (nominal)
- **CAN_L Voltage**: 2.5V ± 0.05V (nominal)
- **Differential Voltage**: 0V (CAN_H - CAN_L)
- **Bus Condition**: Available for transmission, no dominant node

Dominant State (Logic 0 - Active Transmission)

- **CAN_H Voltage**: 3.5V ± 0.2V (driven high)
- **CAN_L Voltage**: 1.5V ± 0.2V (driven low)
- **Differential Voltage**: 2.0V ± 0.4V (CAN_H - CAN_L)
- **Noise Margin**: Minimum 1V differential for reliable detection

Critical Design Note: The 1V differential variation accounts for:

- Cable resistance and voltage drops
- Temperature-induced variations
- Component tolerances
- Electromagnetic interference effects

Termination and Impedance Matching

Proper termination prevents signal reflections and ensures data integrity:

Termination Resistors

- **Value**: 120Ω precision resistors (±1% tolerance recommended)
- **Placement**: Both physical ends of the CAN bus
- **Function**: Match characteristic impedance of twisted-pair cable
- **Effect**: Absorb signal reflections, prevent standing waves

Bus Topology Requirements

- **Linear Topology**: Avoid star configurations and stubs

- **Stub Length**: Maximum 0.3m stub length to nodes
- **Cable Type**: Twisted-pair with 120Ω characteristic impedance
- **Maximum Length**: 40m at 1 Mbps, 500m at 125 kbps

Physical Layer Standards

ISO 11898-2: High-Speed CAN

- **Data Rates**: Up to 1 Mbps
- **Applications**: Critical systems (engine control, brakes, airbags)
- **Fault Tolerance**: Limited - single wire fault disables communication
- **Power Consumption**: Lower due to faster transmission times

ISO 11898-3: Low-Speed/Fault-Tolerant CAN

- **Data Rates**: Up to 125 kbps
- **Fault Tolerance**: Continues operation with single wire failure
- **Applications**: Comfort systems (windows, seats, climate)
- **Implementation**: More complex transceivers, higher cost

Error States and Management System

CAN Node Error State Machine

Every CAN node maintains two error counters and operates in one of three states:

Error Active State (Normal Operation)

Entry Conditions:

- TEC (Transmit Error Counter) < 128
- REC (Receive Error Counter) < 128

Node Behavior:

- Full network participation capabilities
- Can transmit and receive all message types
- Transmits **Active Error Frames** when detecting errors
- Error flags consist of 6 dominant bits
- Forces error condition on all network nodes

Error Response: When detecting bus errors, node broadcasts active error flags to ensure network-wide error awareness

Error Passive State (Degraded Operation)

Entry Conditions:

- TEC > 127 OR REC > 127 (either counter exceeds threshold)

Node Behavior:

- **Restricted participation** with additional timing constraints
- Must wait for additional bit times before retransmission
- Transmits **Passive Error Frames** when detecting errors
- Error flags consist of 6 recessive bits
- **Does not force errors** on other network nodes
- Other nodes may not detect these passive error indications

Design Philosophy: Prevents faulty nodes from disrupting healthy network operation

Bus Off State (Network Isolation)

Entry Conditions:

- TEC > 255 (transmit error counter exceeds critical threshold)

Node Behavior:

- **Complete disconnection** from CAN network
- Cannot transmit or receive any messages
- Must monitor bus for recovery sequence
- Requires **128 occurrences of 11 consecutive recessive bits** for recovery
- Software intervention typically required for recovery

Recovery Process:

1. Monitor bus for stable recessive condition
2. Count 11-bit recessive sequences
3. After 128 sequences, reset error counters
4. Return to Error Active state
5. Resume normal network participation

Error Detection Mechanisms

Message Level Error Detection

CRC (Cyclic Redundancy Check):

- **15-bit CRC sequence** calculated using polynomial division
- **Polynomial**: x^15 + x^14 + x^10 + x^8 + x^7 + x^4 + x^3 + 1
- **Transmitter**: Calculates and appends CRC to frame
- **Receiver**: Recalculates CRC and compares with received value
- **Error Detection**: Mismatched CRC indicates data corruption

ACK (Acknowledgment) Error:

- **Transmitter Action**: Sends recessive bit in ACK slot
- **Receiver Action**: Overwrites with dominant bit if frame received correctly
- **Error Condition**: ACK slot remains recessive (no receivers acknowledged)
- **Implication**: No nodes successfully received the message

Form (Format) Error:

- **Fixed Fields**: EOF, IFS, ACK delimiter must always be recessive
- **Error Detection**: Dominant bits in fixed recessive fields
- **Frame Violation**: Indicates corrupted frame structure
- **Recovery**: Frame destroyed, retransmission initiated

Bit Level Error Detection

Bit Error:

- **Monitoring**: Transmitter continuously monitors bus during transmission
- **Comparison**: Compares transmitted bit with actual bus value
- **Exception Periods**: During arbitration and acknowledgment phases
- **Error Indication**: Mismatch indicates bus fault or collision

Stuff Error:

- **Bit Stuffing Rule**: After 5 consecutive identical bits, insert opposite polarity bit
- **Synchronization**: Ensures regular clock edges for receiver synchronization
- **Error Detection**: 6 consecutive identical bits indicate stuffing violation
- **Automatic Process**: Hardware handles stuffing insertion and removal
- **Exception Fields**: CRC delimiter, ACK field, and EOF are not stuffed

---

CAN Mailboxes and Message Management

Mailbox Architecture Concepts

CAN controllers implement **hardware mailboxes** as dedicated memory buffers for efficient message handling:

Transmission Mailboxes

Function: Store outgoing messages awaiting transmission opportunity

Priority Management:

- Multiple mailboxes enable **priority queuing**
- Controller automatically selects highest priority pending message
- Supports **real-time response** to critical events
- Hardware arbitration reduces software overhead

Status Tracking:

- **Transmission Complete**: Indicates successful message transmission
- **Transmission Error**: Flags transmission failures for software handling
- **Arbitration Lost**: Notification when lower priority message preempted

Reception Mailboxes

Function: Store incoming messages after acceptance filter validation

Filter Integration:

- Each mailbox associated with **acceptance filters**
- Hardware filtering reduces processor interrupt load
- Supports both **individual ID** and **ID range** filtering
- **Maskable filtering** enables flexible message acceptance

Interrupt Management:

- **Message Available**: Signals new message arrival
- **FIFO Status**: Indicates buffer levels and overflow conditions
- **Error Conditions**: Hardware error detection and reporting

Message Filtering System

Acceptance Filter Operation

ID List Mode:

- **Exact Matching**: Accepts only specifically programmed identifiers

- **High Selectivity**: Precise control over received messages
- **Limited Capacity**: Fixed number of acceptable IDs
- **Application**: Systems requiring specific message sets

ID Mask Mode:

- **Range Filtering**: Accepts identifier ranges using bit masks
- **Flexible Configuration**: Single filter covers multiple related IDs
- **Efficient Usage**: Reduces filter resource requirements
- **Application**: Systems with hierarchical message addressing

FIFO Buffer Management

Message Queuing:

- **Sequential Storage**: Messages stored in arrival order
- **Overflow Handling**: Configurable behavior when buffer full
- **Threshold Interrupts**: Programmable interrupt levels
- **Multiple Priorities**: Separate FIFOs for different message classes

Buffer Strategies:

- **Overwrite Oldest**: Maintains most recent messages
- **Block New**: Preserves existing messages until processed
- **Error Generation**: Signals overflow conditions to application

---

STM32 CAN Programming Implementation

System-Level Configuration

Clock Configuration Strategy

```
System Clock Architecture:
HSE (External Crystal) = 8 MHz
HCLK (System Clock) = 72 MHz
APB1 (Peripheral Clock) = 36 MHz ← CAN1 clock domain
```

CAN Bit Timing Calculation

Step 1: Time Quanta Calculation

```
Prescaler = 18
CAN_Time_Quanta = APB1_PCLK / Prescaler
                = 36 MHz / 18
                = 2 MHz
                = 500 ns per time quanta
```

Step 2: Bit Segment Configuration

```
Bit Timing Segments:
- Bit Segment 1: 2 Time Quanta (sample point positioning)
- Bit Segment 2: 1 Time Quanta (sample to bit end)
- SJW (Sync Jump Width): 1 Time Quanta (sync adjustment)
```

Step 3: Baud Rate Determination

```
Total Bit Time = (2 + 1 + 1) × 500 ns = 2000 ns
Baud Rate = 1 / 2000 ns = 500,000 bps = 500 kbps
```

Hardware Pin Assignment

```
CAN1 Interface Mapping (STM32F407VG):
- PB8: CAN1_RX (Receive pin)
- PB9: CAN1_TX (Transmit pin)
- Clock Domain: APB1 (36 MHz)
- Alternate Function: AF9
```

CAN Filter Configuration Deep Dive

Filter Bank Architecture

```
CAN_FilterTypeDef FilterConfig;

// Basic filter parameters
FilterConfig.FilterActivation = CAN_FILTER_ENABLE;
FilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0;  // Route to FIFO 0
FilterConfig.SlaveStartFilterBank = 14;            // STM32F4 dual CAN
FilterConfig.FilterBank = 10;                      // Filter bank selection (0-13)
```

ID Mask Mode Configuration

```
// Configure for ID range acceptance
FilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;    // 32-bit filter
FilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;     // Mask mode

// Accept messages 0x0A8 through 0x0AF (8 consecutive IDs)
FilterConfig.FilterMaskIdHigh = 0x07F8 << 5;        // Mask: bits 3-10 must match
FilterConfig.FilterMaskIdLow = 0x0000;               // Low 16 bits
FilterConfig.FilterIdHigh = 0x00A8 << 5;            // Base ID: 0x0A8
```

```
FilterConfig.FilterIdLow = 0x0000;                    // Low 16 bits

// Apply configuration
HAL_CAN_ConfigFilter(&hcan1, &FilterConfig);
```

Mask Explanation:

- **Mask 0x07F8**: Binary 11111111000 - checks bits 3-10
- **Base ID 0x0A8**: Binary 10101000 - target pattern
- **Accepted Range**: 0x0A8-0x0AF (last 3 bits can vary)

Complete Initialization Sequence

```
void CAN_System_Initialize(void) {
    // Step 1: Hardware abstraction layer initialization
    MX_CAN1_Init();  // CubeMX generated configuration

    // Step 2: Configure message acceptance filters
    Configure_CAN_Acceptance_Filters();

    // Step 3: Start CAN peripheral operation
    if(HAL_CAN_Start(&hcan1) != HAL_OK) {
        Error_Handler();  // Handle initialization failure
    }

    // Step 4: Enable interrupt notifications
    HAL_CAN_ActivateNotification(&hcan1, CAN_IT_RX_FIFO0_MSG_PENDING);
}
```

Message Reception Implementation

Reception Data Structures

```
// Global variables for message handling
CAN_RxHeaderTypeDef RxHeader;    // Message metadata
uint8_t RxData[^8];               // Message payload buffer
```

Interrupt Callback Function

```
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan) {
    // Retrieve message from hardware FIFO
    HAL_Status status = HAL_CAN_GetRxMessage(&hcan1, CAN_RX_FIFO0,
                                            &RxHeader, RxData);

    if(status == HAL_OK) {
        // Process message based on identifier
        Process_Received_Message(RxHeader.StdId, RxData, RxHeader.DLC);
    }
}

void Process_Received_Message(uint32_t message_id, uint8_t* data, uint32_t length) {
    switch(message_id) {
        case 0x0A8:  // Engine temperature
            Handle_Engine_Temperature(data, length);
            break;

        case 0x0A9:  // Vehicle speed
            Handle_Vehicle_Speed(data, length);
            break;

        case 0x0AA:  // Brake pressure
            Handle_Brake_Pressure(data, length);
            break;

        default:
```

```
                // Log unexpected message ID
                Log_Unknown_Message(message_id);
                break;
        }
    }
```

Message Transmission Implementation

Multi-Message Transmission Example

```c
void CAN_Transmit_System_Data(void) {
    // Message structures for different priorities
    CAN_TxHeaderTypeDef TxHeader1, TxHeader2, TxHeader3;
    uint32_t TxMailbox1, TxMailbox2, TxMailbox3;
    uint8_t TxData1[^8], TxData2[^8], TxData3[^8];

    // High Priority Message: Critical safety data (ID 0x0A8)
    TxHeader1.TransmitGlobalTime = DISABLE;
    TxHeader1.IDE = CAN_ID_STD;              // Standard 11-bit identifier
    TxHeader1.ExtId = 0;                     // Not used for standard frames
    TxHeader1.StdId = 0x0A8;                 // Highest priority in our system
    TxHeader1.RTR = CAN_RTR_DATA;            // Data frame (not remote request)
    TxHeader1.DLC = 8;                       // Full 8-byte payload

    strcpy((char*)TxData1, "SUNBEAM");       // System identification string
    if(HAL_CAN_AddTxMessage(&hcan1, &TxHeader1, TxData1, &TxMailbox1) != HAL_OK) {
        Handle_Transmission_Error(0x0A8);
    }

    // Medium Priority Message: Sensor data (ID 0x0A9)
    TxHeader2.TransmitGlobalTime = DISABLE;
    TxHeader2.IDE = CAN_ID_STD;
    TxHeader2.StdId = 0x0A9;                 // Medium priority
    TxHeader2.RTR = CAN_RTR_DATA;
```

```
    TxHeader2.DLC = 1;                       // Single byte payload

    TxData2[^0] = 0x11;                       // Sensor status byte
    if(HAL_CAN_AddTxMessage(&hcan1, &TxHeader2, TxData2, &TxMailbox2) != HAL_OK) {
        Handle_Transmission_Error(0x0A9);
    }

    // Lower Priority Message: Diagnostic data (ID 0x0AD)
    TxHeader3.TransmitGlobalTime = DISABLE;
    TxHeader3.IDE = CAN_ID_STD;
    TxHeader3.StdId = 0x0AD;                 // Lower priority
    TxHeader3.RTR = CAN_RTR_DATA;
    TxHeader3.DLC = 1;

    TxData3[^0] = 0x22;                       // Diagnostic code
    if(HAL_CAN_AddTxMessage(&hcan1, &TxHeader3, TxData3, &TxMailbox3) != HAL_OK) {
        Handle_Transmission_Error(0x0AD);
    }
}
```

Loopback Mode for Development

Configuration Benefits:

- **Internal Testing**: Transmitted messages immediately received
- **No Hardware Required**: Software validation without physical bus
- **Debug Capability**: Verify protocol implementation before deployment
- **Integration Testing**: Validate complete transmit/receive paths