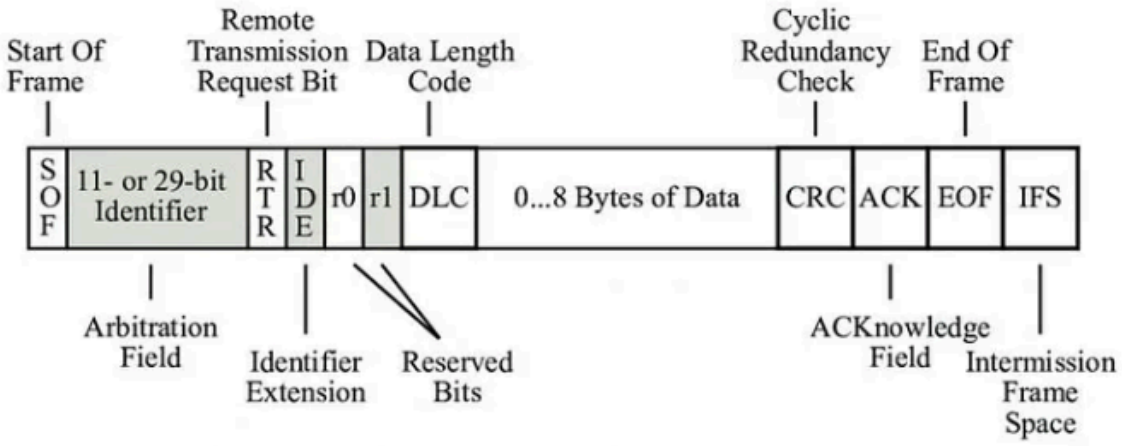
CAN COMMUNICATION

Controller Area Network (CAN) communication is a serial communication protocol designed for real-time, high-speed data exchange between electronic control units (ECUs) in embedded systems. It operates on a multi-master, message-based architecture, allowing multiple nodes to communicate efficiently without a central host. The protocol employs differential signaling over two wires (CAN_H and CAN_L) to ensure reliable data transmission, even in electrically noisy environments.

CAN Communication Data Frame and How it works:



IDLE State:

- When there is no frame on the bus, the bus is in an IDLE state.
- In the IDLE state, the bus has a recessive bit (logic 1).

Start of Frame (SOF):

- Every data frame starts with a dominant bit called the Start of Frame (SOF).
- SOF is 1 bit long and is always dominant (logic 0).
- This bit informs all nodes about the start of a new frame on the bus, signaling that the bus is no longer idle and synchronizing all receiving (Rx) nodes with the transmitting (Tx) node.

Message ID:

- CAN is a message-based protocol where each frame is identified by a Message ID.
- In a standard frame, the Message ID is 11 bits long.
- Typically, the 11th and 4th bits are masked for complex reasons, resulting in 9 unique bits in the Message ID.

- This allows for $2^9 = 512$ different CAN frames in a single CAN network, with the Message ID ranging from hexadecimal 000 to 7FF.
- Mentioned as Identifier in the above figure.

Remote Transmission Request (RTR):

- After the Message ID, there is a single bit called the Remote Transmission Request (RTR).
- RTR differentiates between a data frame and a remote frame:
- RTR = 0: Data Frame
- RTR = 1: Remote Frame

Reserved Bits:

- Following the RTR bit are two reserved bits, R1 and R0.
- These bits are always recessive and were originally intended for future enhancements of the CAN protocol.
- Nowadays, R1 is used as the IDE bit to differentiate between Standard and Extended frames, while R0 is used as the FDF bit to differentiate between CAN and CAN-FD frames.

Data Length Code (DLC):

- The next field is the Data Length Code (DLC), which indicates the number of bytes of data in the frame.
- DLC is 4 bits long and can range from 0 to 8, allowing for up to 8 bytes of data.

Data Field:

- Following the DLC is the actual data field, which contains the data bytes specified by the DLC.
- The data field can contain a maximum of 8 bytes and a minimum of 0 bytes.

Checksum (CRC):

- After the data field, there is a 15-bit CRC (Cyclic Redundancy Check) field used for error detection.
- The transmitting node computes the CRC checksum and includes it in the frame.
- The receiving node also computes its own checksum for the received data and compares it with the CRC field. If they match, the frame is considered valid; otherwise, it is invalid.

CRC Delimiter (CD):

- CRC Delimiter, a single bit used to provide time for the receiver to validate the frame.
- The CRC Delimiter is always recessive.

Acknowledgement (ACK):

- The transmitting node puts a recessive bit in the ACK slot.

- All receiving nodes that successfully received and validated the frame will overwrite this bit with a dominant bit to acknowledge receipt.
- If the transmitting node does not see a dominant bit in the ACK slot, it signals an acknowledgement error.

Acknowledgement Delimiter (AD):

- Acknowledgement Delimiter is a single recessive bit.

End of Frame (EOF):

- The EOF field consists of 7 consecutive recessive bits, marking the end of the frame.

Intermission (IFS):

- These 3 bits signify the end of the frame and the bus returning to the IDLE state.

Frame Composition:

- **Arbitration Field:** Message ID, RTR
- **Control Field:** Reserved bits (R1, R0), DLC
- **Data Field:** Actual data bytes
- **CRC Field:** CRC checksum and CRC Delimiter
- **Acknowledgement Field:** ACK slot and Acknowledgement Delimiter
- **EOF Field:** EOF and IFS bits

Every data frame ends with 11 consecutive recessive bits: 1 bit of Acknowledgement Delimiter, 7 bits of EOF, and 3 bits of IFS.Once the frame ends, the bus is idle again, and any node can start transmitting its own frame.

Types of CAN Communication

1. Classical CAN (Standard CAN)

CAN is the original CAN protocol, introduced by Bosch in the 1980s. It uses a message-based communication system with a fixed data frame size.

Key Features:

- Frame Size: Supports up to 8 bytes of data per frame.
- Identifier Types:
  - Standard Frame Format (11-bit identifier)
  - Extended Frame Format (29-bit identifier)
- Baud Rate: Operates at speeds up to 1 Mbps.
- Error Handling: Uses CRC (Cyclic Redundancy Check) and automatic retransmission to ensure data integrity.

2. CAN FD (Flexible Data-Rate CAN)

CAN FD is an improved version of Classical CAN, designed to increase data throughput and support higher speeds. It is backward compatible with Classical CAN.

Key Features:

- ➢ Frame Size: Supports up to 64 bytes of data per frame (compared to 8 bytes in Classical CAN).
- ➢ Baud Rate: Allows for higher speeds (up to 5 Mbps) in the data phase.
- ➢ Efficient Bandwidth Utilization:
  - The arbitration phase runs at Classical CAN speeds.
  - The data phase runs at a higher speed, improving performance.

3. CANopen

CANopen is a high-level protocol based on the CAN standard. It defines communication profiles and a device model to enable standardized interaction between nodes.

Key Features:

- ➢ Device Profiles: Standardized formats for different device types (sensors, motors, controllers).
- ➢ Communication Objects:
  - Process Data Object (PDO): For real-time data exchange.
  - Service Data Object (SDO): For parameter configuration.
- ➢ Networking: Uses node IDs to communicate with multiple devices on the same network.

CAN Modules:

CAN Transceiver converts logic signals to differential signals for communication.

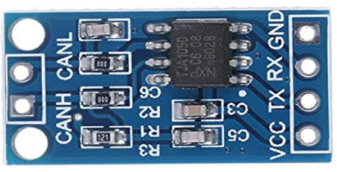
SN65HVD230:

The SN65HVD230 is a 3.3V CAN transceiver designed for use in low-power applications. It is compatible with microcontrollers operating at 3.3V logic levels.

Key Features:

- ➢ **Operating Voltage:** 3.3V (compatible with 5V systems)
- ➢ **Data Rate:** Supports up to 1 Mbps
- ➢ **Low Power Mode:** Includes silent mode and standby mode to reduce power consumption
- ➢ **Dominant Time-Out Protection:** Prevents stuck dominant state on the CAN bus
- ➢ **Thermal Shutdown Protection:** Protects against overheating
- ➢ **ISO 11898-2 Compliant:** Ensures compatibility with standard CAN networks

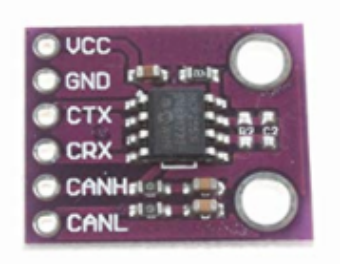
TJA1050: The TJA1050 is a 5V CAN transceiver, widely used for high-speed and robust CAN communication.



Key Features:

- **Operating Voltage:** 5V (not compatible with 3.3V logic without level shifting)
- **Data Rate:** Supports up to 1 Mbps
- **High Noise Immunity:** Improved electromagnetic compatibility (EMC) for industrial environments
- **Slope Control Mode:** Reduces EMI (Electromagnetic Interference)
- **Thermal Protection:** Includes overtemperature shutdown for safety
- **ISO 11898-2 Compliant:** Ensures compatibility with standard CAN networks

MCP2551 : The MCP2551 is a high-speed CAN transceiver It supports high-speed data transmission in industrial and automotive applications.



Key Features of MCP2551

- **Operating Voltage:** 5V
- **Max Data Rate:** 1 Mbps
- **Bus Fault Protection:** Up to ±42V on CANH and CANL
- **Thermal Protection:** Automatic shutdown in case of overheating
- **Slope Control Mode:** Reduces Electromagnetic Interference (EMI) in noisy environments
- **High Noise Immunity:** Enhanced reliability in industrial and automotive systems
- **Dominant Time-Out Protection:** Prevents bus lockup by limiting the dominant state duration.

Modes: Normal mode and loopback mode

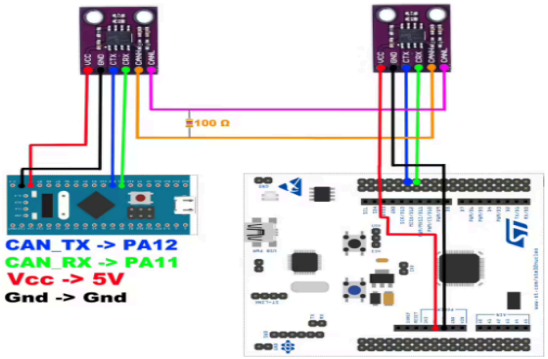
Normal Mode:

In Normal Mode, CAN transceivers send and receive messages on the CAN bus. Each node on the bus monitors the messages and responds when needed.

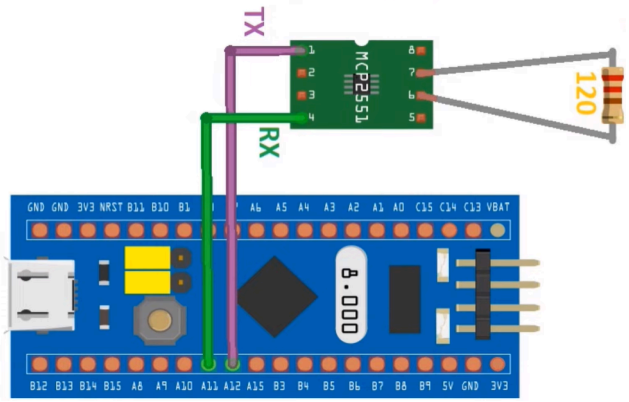
Loopback Mode:

In Loopback Mode, the CAN controller internally loops transmitted messages back to the receiver without sending them on the physical bus.

Hardware setup for normal mode:



Hardware setup for loopback mode:



For Transmitting side:

TxHeader.IDE = CAN_ID_STD;
TxHeader.StdId = 0x7FF; //stdId between 0x000 and 0x7FF
TxHeader.RTR = CAN_RTR_DATA;
TxHeader.DLC = 5; //data length
This function to use to transmit the data:
HAL_CAN_AddTxMessage(CAN_HandleTypeDef *hcan, const CAN_TxHeaderTypeDef *pHeader, const uint8_t aData[], uint32_t *pTxMailbox)

For Recieving side:

HAL_CAN_ActivateNotification(&hcan, CAN_IT_RX_FIFO0_MSG_PENDING)

```
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
  if (HAL_CAN_GetRxMessage(hcan, CAN_RX_FIFO0, &RxHeader, RxData) == HAL_OK) {
     if (RxHeader.StdId == 0x7FF) {
        //indication message
     }
  }
}
```

```
CAN_FilterTypeDef canFilter;
canFilter.FilterBank = 0;
canFilter.FilterMode = CAN_FILTERMODE_IDMASK;
canFilter.FilterScale = CAN_FILTERSCALE_32BIT;
canFilter.FilterIdHigh = (0x7FF<<5);//according to transmitting stdId
canFilter.FilterIdLow = 0x0000;
canFilter.FilterMaskIdHigh = (0x7FF<<5);
canFilter.FilterMaskIdLow = 0x0000;
canFilter.FilterFIFOAssignment = CAN_RX_FIFO0;
```

```
canFilter.FilterActivation = ENABLE;
if (HAL_CAN_ConfigFilter(&hcan, &canFilter) != HAL_OK) {
 Error_Handler();
}
```

Vesc CAN Communication

Commands ID:

```
CAN_PACKET_SET_DUTY = 0,
CAN_PACKET_SET_CURRENT = 1,
CAN_PACKET_SET_CURRENT_BRAKE = 2,
CAN_PACKET_SET_RPM = 3,
CAN_PACKET_SET_POS = 4,
CAN_PACKET_FILL_RX_BUFFER = 5,
CAN_PACKET_FILL_RX_BUFFER_LONG = 6,
CAN_PACKET_PROCESS_RX_BUFFER = 7,
CAN_PACKET_PROCESS_SHORT_BUFFER = 8,
CAN_PACKET_STATUS = 9,
CAN_PACKET_SET_CURRENT_REL = 10,
CAN_PACKET_SET_CURRENT_BRAKE_REL = 11,
CAN_PACKET_SET_CURRENT_HANDBRAKE = 12,
CAN_PACKET_SET_CURRENT_HANDBRAKE_REL = 13,
CAN_PACKET_STATUS_2 = 14,
CAN_PACKET_STATUS_3 = 15,
CAN_PACKET_STATUS_4 = 16,
CAN_PACKET_PING = 17,
CAN_PACKET_PONG = 18,
CAN_PACKET_DETECT_APPLY_ALL_FOC = 19,
CAN_PACKET_DETECT_APPLY_ALL_FOC_RES = 20,
CAN_PACKET_CONF_CURRENT_LIMITS = 21,
CAN_PACKET_CONF_STORE_CURRENT_LIMITS = 22,
CAN_PACKET_CONF_CURRENT_LIMITS_IN = 23,
CAN_PACKET_CONF_STORE_CURRENT_LIMITS_IN = 24,
CAN_PACKET_CONF_FOC_ERPMS = 25,
CAN_PACKET_CONF_STORE_FOC_ERPMS = 26,
CAN_PACKET_STATUS_5 = 27
```

These command numbers are put in the second byte of the 29 bit ID for the extended CAN frame. You need an extended frame (29 bits) vs. standard frame (11 bits) since bits 0-7 are reserved for numbering the individual speed controllers (0-255). With only 3 bits left, only 8 commands would be available if you used a standard frame.

VESC Extended ID Structure

Bits 0-7: Controller ID (0-255)
Bits 8-15: Command number
Bits 16-28: Command-specific data
ExtId should be declared as:
ExtId = (command<<8) | can_id