EEE4020 – Embedded System Design

Final Report

CAN Communication Protocol

Submitted By:

Johny John & Nitin Flavier

20BEE0028 & 20BEE0341

Slot: F1

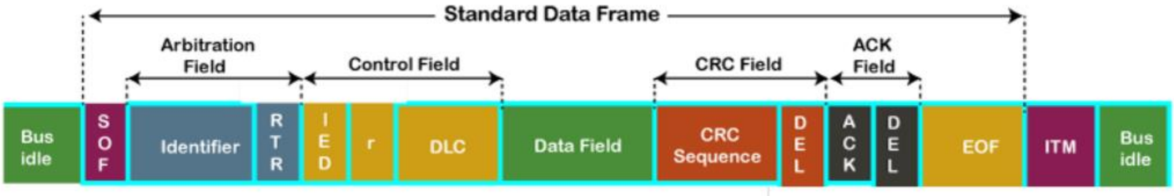
Faculty: Prof Selvakumar

Introduction:

CAN stands for **Controller Area Network** protocol. It is a protocol that was developed by **Robert Bosch** in around 1986. The CAN protocol is a standard designed to allow the microcontroller and other devices to communicate with each other without any host computer. The feature that makes the CAN protocol unique among other communication protocols is the broadcast type of bus. Here, broadcast means that the information is transmitted to all the nodes. The node can be a sensor, microcontroller, or a gateway that allows the computer to communicate over the network through the USB cable or ethernet port.

The CAN is a message-based protocol, which means that message carries the message identifier, and based on the identifier, priority is decided. There is no need for node identification in the CAN network, so it becomes very easy to insert or delete it from the network. It is a serial half-duplex and asynchronous type of communication protocol. The CAN is a two-wired communication protocol as the CAN network is connected through the two-wired bus. The wires are twisted pair having 120Ω characteristics impedance connected at each end. Initially, it was mainly designed for communication within the vehicles, but it is now used in many other contexts. Like UDS, and KWP 2000, CAN also be used for the on-board diagnostics.

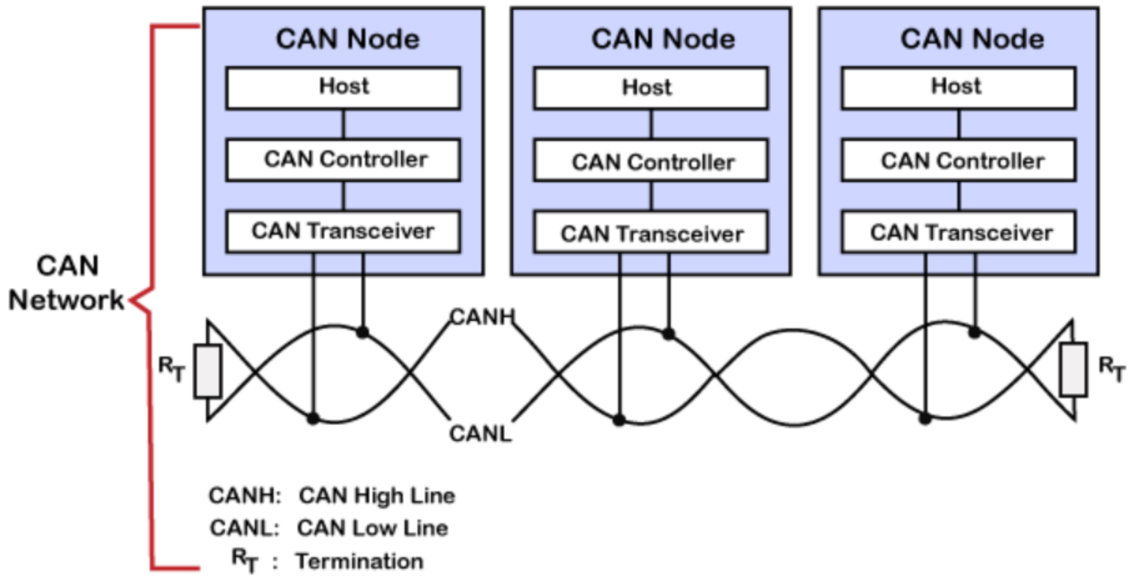
CAN Framing



- **SOF:** SOF stands for the start of frame, which indicates that the new frame is entered in a network. It is of 1 bit.
- **Identifier:** A standard data format defined under the CAN 2.0 A specification uses an 11-bit message identifier for arbitration. Basically, this message identifier sets the priority of the data frame.
- **RTR:** RTR stands for Remote Transmission Request, which defines the frame type, whether it is a data frame or a remote frame. It is of 1-bit.
- **Control field:** It has user-defined functions.
    1. **IDE:** An IDE bit in a control field stands for identifier extension. A dominant IDE bit defines the 11-bit standard identifier, whereas recessive IDE bit defines the 29-bit extended identifier.

2. **DLC:** DLC stands for Data Length Code, which defines the data length in a data field. It is of 4 bits.
3. **Data field:** The data field can contain upto 8 bytes.

- **CRC field:** The data frame also contains a cyclic redundancy check field of 15 bit, which is used to detect the corruption if it occurs during the transmission time. The sender will compute the CRC before sending the data frame, and the receiver also computes the CRC and then compares the computed CRC with the CRC received from the sender. If the CRC does not match, then the receiver will generate the error.
- **ACK field:** This is the receiver's acknowledgment. In other protocols, a separate packet for an acknowledgment is sent after receiving all the packets, but in case of CAN protocol, no separate packet is sent for an acknowledgment.
- **EOF:** EOF stands for end of frame. It contains 7 consecutive recessive bits known End of frame.



The twisting of the two lines also reduces the magnetic field. The bus is terminated with 120Ω resistance at each end.

Main Points:

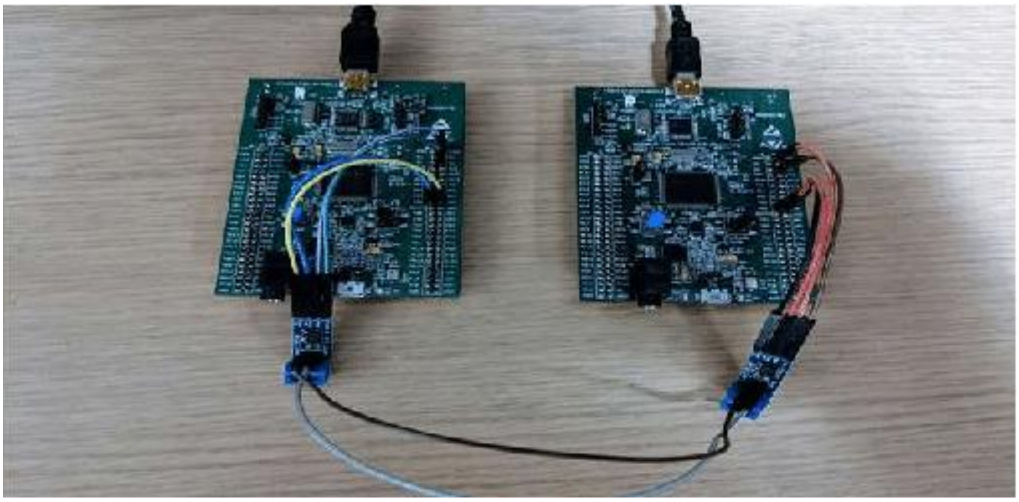
- Logic 1 is a recessive state. To transmit 1 on CAN bus, both CAN high and CAN low should be applied with 2.5V.
- Logic 0 is a dominant state. To transmit 0 on CAN bus, CAN high should be applied at 3.5V and CAN low should be applied at 1.5V.

- The ideal state of the bus is recessive.
- If the node reaches the dominant state, it cannot move back to the recessive state by any other node.

Project Aim:
To send 100 bytes of data from one CAN Node to another Node.

Components Used:

1) STM32F407 (2)
2) CAN Trans receiver CJMCU 2551 (2)
3) 120-ohm Resistors (2)
4) Twisted Wire Cables (2 pairs)
5) Breadboard (2)



Explaining Configuration for CAN Transceiver:

First, we will configure pins for CAN Tx & CAN Rx as PB9 and PB8 using alternate functionality register, and enable the peripheral clock.

```
void Set_Pin(void)
{
    RCC->AHB1ENR |= (1 << 1);                // GPIOB enabled
    GPIOB->MODER |= (0xA << 16);           // Alternate functionality mode

    // Setting AFR_H for port B pin 8,9 we are making it as CAN_Rx(8) & CAN_Tx(9)
    GPIOB->AFR[1] |= (0x9 | (0x9 << 4));
}
```

Now we will Initialize the CAN peripheral for bit rate as 125kbits/sec.

```
void CAN_Init(void)
{
    RCC->APB1ENR |= (1<<25);     // Enable CAN 1 clock
 /*
    The software requests bxCAN to enter initialization or Sleep mode
    by setting the INRQ or SLEEP bits in the CAN_MCR register. Once the mode has been
    entered, bxCAN confirms it by setting the INAK or SLAK bits in the CAN_MSR register
 */

    CAN1->MCR |= (1<<0);        // CAN INRQ requesting it to enter into initialization
    while(!(CAN1->MSR & 0x1));   // Mode has been entered it is acknowledged

    //CAN1->BTR |= (1<<30);      // CAN LoopBack Mode here

    CAN1->BTR &= ~(3<<24);     // SWJ 1 Time Quantum

    CAN1->BTR &= ~(0x7F << 16);  // clearing

    CAN1->BTR |= (12 << 16);   // Time Segment 1
    CAN1->BTR |= (1 << 20);     // Time Segment 2

    CAN1->BTR |= (7 << 0);      // Baud-Rate Prescaler
```

We can use CAN Bit timing website to calculate all the parameters:
http://www.bittiming.can-wiki.info/
We have Set the STD-ID for the CAN transmitting byte as 0x245.

```
    /*
        Exit Initialization mode
        Wait until INAK bit is cleared by hardware
    */
    CAN1->MCR &= ~(1<<0);
    while(CAN1->MSR & 0x1);

    /*
        Exit Sleep mode
        Wait until INAK bit is
        cleared by hardware
    */
    CAN1->MCR &= ~(1 << 1);
    while(CAN1->MSR & 0x2);

    /* Setting Up Transmission */

    CAN1->sTxMailBox[0].TIR = 0;

    CAN1->sTxMailBox[0].TIR |= (0x245 << 21);    // STD ID : 138

    CAN1->sTxMailBox[0].TDHR = 0;     // Data byte 4, 5, 6, 7
    CAN1->sTxMailBox[0].TDTR = 1;    // Sending only 1 byte

}
```

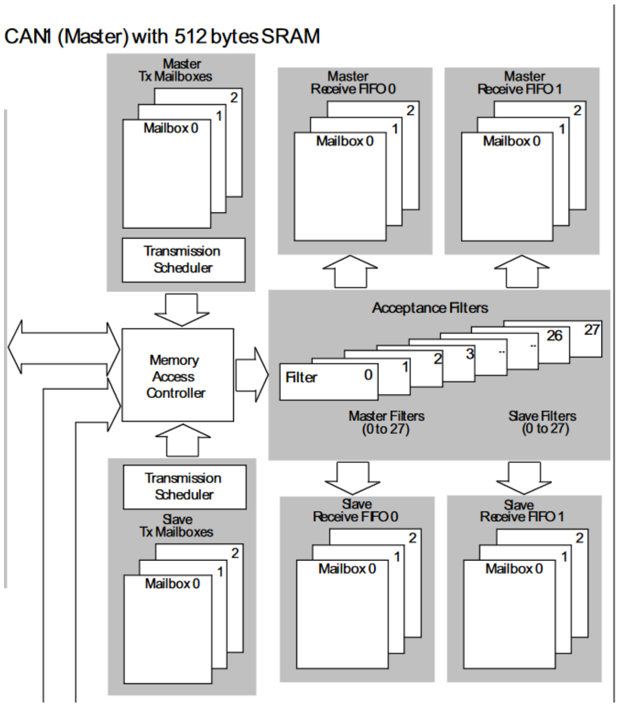
Now for transmitting data we will use 1 byte per message.

```c
void CAN_Tx(uint8_t message)
{
    CAN1->sTxMailBox[0].TDLR = message;

    /* Request for transmission */
    CAN1->sTxMailBox[0].TIR |= 1;

}
```

Explaining Configuration for CAN Receiver Side:

The initializing of CAN is similar as above on the receiver side we configure filters, bank registers & FIFO

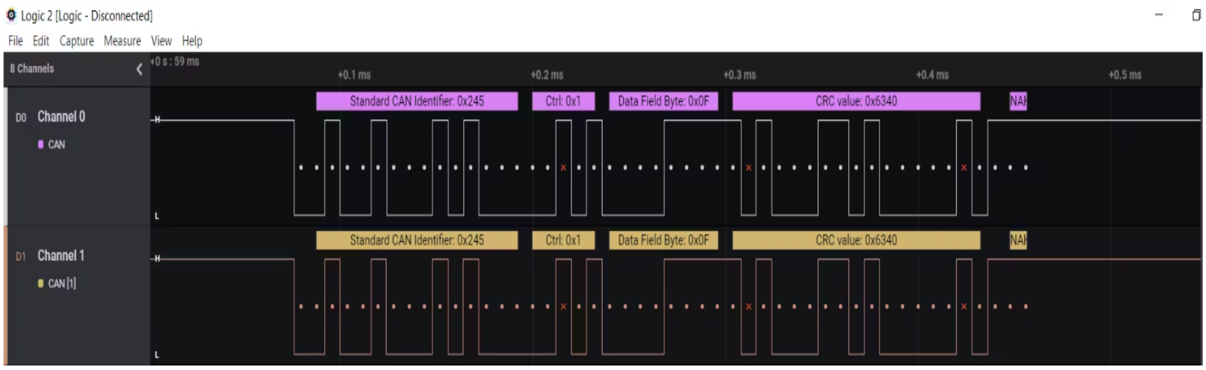
```c
    /* Filter Configuration */

    CAN1->FMR |= 1<<0;   // Initialise filter mode

    CAN1->FMR |= 14<<8;  // they define start bank for CAN2

    CAN1->FS1R |= 1<<13;  // 32 bit scale configuration Filter

    // Filtering it with respect to the ID

    CAN1->FM1R |= (1<<13);   // Identifier List-Mode

    CAN1->sFilterRegister[13].FR1 = 138 << 21;   // STD ID

    CAN1->FA1R |= (1<<13);

    CAN1->FMR &= ~(0x1);

}
```

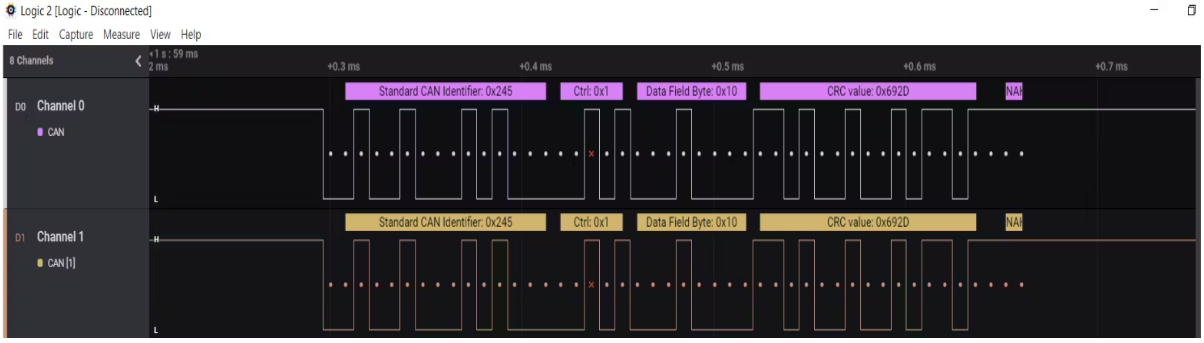


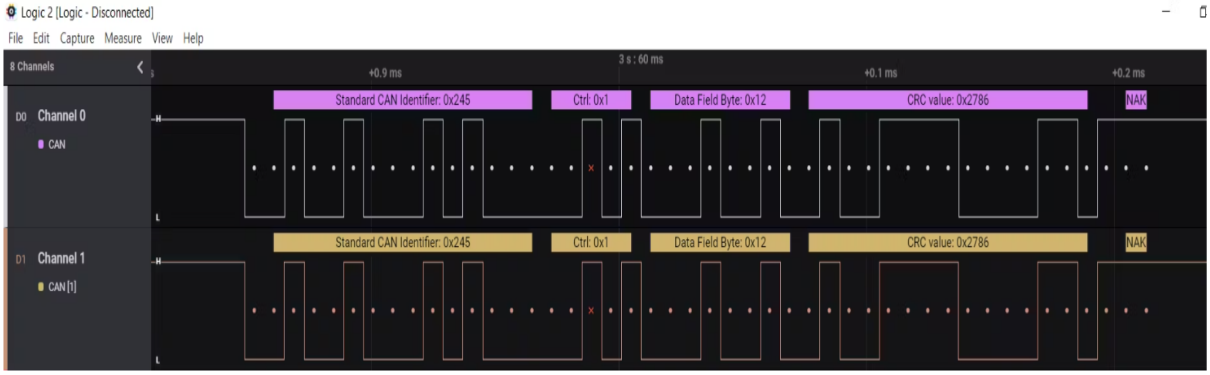
Code to get the data from the FIFO Mailbox:

```c
uint8_t CAN_Rx(void)
{
    while(!(CAN1->RF0R & 3));    // waiting for atleast one message

    uint8_t data = (CAN1->sFIFOMailBox[0].RDLR) & 0xFF; // only 1 byte

    CAN1->RF0R |= (1<<5);  // Release FIFO MailBox

    return data;
}
```

Outputs:







Conclusion:

Do not forget to put terminal resistors, so that the transmitted message does not get rebounded.

Try to use twisted pair cable for CANH & CANL here.

If encountered with CAN_Error Message then first check the bit rate, check terminal resistance across each CAN Node it should be 60 ohms, check the BTR register and calculate the right bit timings for the communication.

Code:

https://drive.google.com/drive/folders/1Ncc1N_CAI5LoJmt4OvB1E9VcgEnrJQpE?usp=share_link