Advanced Microcontrollers - ARM

Agenda

- CAN protocol

CAN protocol

- Refer slides

Core concepts

CAN (Controller Area Network), is a robust and widely used communication protocol in the realm of embedded systems, particularly in automotive and industrial applications. It was developed by Bosch in the mid-1980s to facilitate reliable communication between various electronic control units (ECUs) in vehicles.

Here's an overview of how CAN protocol works:

1. **Message-Based Communication**: CAN is a message-based protocol, meaning data is transmitted in discrete packets called frames. These frames consist of a header and payload, and they are transmitted over the CAN bus.
2. **Differential Signaling**: CAN uses differential signaling to enhance noise immunity. In this scheme, data is transmitted differentially between two wires: CAN_H and CAN_L. This helps mitigate the effects of electromagnetic interference (EMI) and noise, making CAN suitable for use in noisy industrial environments.
3. **Arbitration**: CAN employs a non-destructive bitwise arbitration mechanism to ensure that multiple nodes on the network can access the bus without causing data corruption. When multiple nodes attempt to transmit simultaneously, the one with the highest priority (determined by its identifier, or ID) wins the arbitration process and gains bus access.
4. **Error Detection and Handling**: CAN includes built-in error detection mechanisms to ensure data integrity. This includes cyclic redundancy check (CRC) for error detection and acknowledgement mechanisms for error handling. If an error is detected during transmission, the transmitting node can retry the transmission.

5. **Two Frame Formats**: CAN supports two frame formats: standard and extended. Standard frames use an 11-bit identifier, while extended frames use a 29-bit identifier. This allows for a larger number of unique message IDs, enabling more flexible and scalable communication networks.

6. **Masterless and Peer-to-Peer Communication**: CAN operates in a masterless fashion, meaning there is no central node controlling communication. Instead, all nodes have equal access to the bus, facilitating peer-to-peer communication and decentralized control.

Overall, CAN protocol offers reliable, deterministic, and efficient communication suitable for real-time embedded systems in automotive, industrial automation, medical devices, and other applications requiring robust communication networks. Its widespread adoption and proven track record make it a cornerstone technology in the field of embedded systems.

CAN Standard/Extended Frames

In the CAN (Controller Area Network) protocol, there are two main frame formats: standard and extended. These formats define how data is structured and transmitted over the CAN bus. Here's a breakdown of each:

1. **Standard CAN Frame**:
   - **Identifier**: Standard CAN frames use an 11-bit identifier, which allows for up to 2^11 (or 2048) unique message IDs. This identifier determines the priority of the message during arbitration on the bus.
   - **Data Length Code (DLC)**: The DLC field specifies the number of bytes of data contained in the frame. It can range from 0 to 8 bytes.
   - **Data**: This field contains the actual data payload of the message, with a maximum length of 8 bytes.
   - **CRC (Cyclic Redundancy Check)**: The CRC field is used for error detection. It contains a checksum calculated from the data and is used by the receiving node to verify the integrity of the received message.
   - **Acknowledgement (ACK)**: The ACK field indicates whether the message was successfully received by other nodes on the bus.
   - **End of Frame (EOF)**: The EOF field marks the end of the frame transmission.

2. **Extended CAN Frame**:
   - **Identifier**: Extended CAN frames use a 29-bit identifier, significantly expanding the number of unique message IDs to 2^29 (over 500 million). This allows for a much larger address space, accommodating a wider range of devices and applications.
   - **Data Length Code (DLC)**: Similar to standard frames, the DLC field specifies the number of bytes of data contained in the frame, ranging from 0 to 8 bytes.
   - **Data**: Just like in standard frames, this field contains the actual data payload of the message, with a maximum length of 8 bytes.

- **CRC (Cyclic Redundancy Check)**: The CRC field is used for error detection, calculating a checksum from the data to verify its integrity.
- **ACK**: Similar to standard frames, the ACK field indicates whether the message was successfully received by other nodes on the bus.
- **EOF**: The End of Frame field marks the end of the frame transmission.

The key difference between standard and extended CAN frames lies in the length of the identifier field, with extended frames allowing for much larger address spaces. This distinction enables extended frames to be used in applications requiring a greater number of unique message IDs, such as complex automotive networks or industrial automation systems.

CAN Bus Electrical Characteristics

The CAN (Controller Area Network) bus is a robust communication protocol used in various applications, including automotive, industrial automation, and more. Understanding its electrical characteristics is crucial for designing and implementing reliable communication networks. Here are the key electrical characteristics of the CAN bus:

1. **Differential Signaling**: CAN uses differential signaling, which means that it transmits data over two wires: CAN_H and CAN_L. The voltage difference between these two wires determines the state of the signal. This differential signaling helps improve noise immunity and allows for reliable communication in noisy environments.

2. **Voltage Levels**:

   - CAN_H and CAN_L voltages are typically within the range of 2.0 to 3.5 volts when transmitting a dominant (logic 0) state.
   - In the recessive (logic 1) state, both CAN_H and CAN_L are pulled to approximately 2.5 volts using termination resistors.
   - The voltage difference between CAN_H and CAN_L during dominant and recessive states is typically around 2 to 0 volts.

3. **Termination Resistors**:

   - Termination resistors are used at both ends of the CAN bus to match the characteristic impedance of the bus (usually 120 ohms).
   - These resistors help minimize signal reflections and ensure signal integrity by properly terminating the transmission line.
   - Signal reflections can cause signal distortion, interference, and data corruption, leading to communication errors.
   - Termination resistors help maintain signal integrity by absorbing reflections and preventing them from interfering with the transmitted signals.

4. **Bit Timing**:

   - CAN uses a bit-wise arbitration mechanism to determine message priority on the bus.

- The bit timing parameters, such as bit rate, synchronization, and sampling points, are crucial for proper communication.
- CAN controllers typically support various bit rates, commonly ranging from a few kilobits per second (Kbps) to several megabits per second (Mbps), depending on the application requirements.

5. **Physical Layer Standards**:

   - CAN physical layer standards, such as ISO 11898-2 for high-speed CAN (up to 1 Mbps) and ISO 11898-3 for low-speed/fault-tolerant CAN (up to 125 Kbps), define the electrical characteristics and requirements for CAN communication networks.
   - These standards specify parameters such as voltage levels, termination resistor values, and physical connector specifications to ensure interoperability between different CAN devices.

6. **Noise Immunity**:

   - CAN is designed to provide high noise immunity, allowing for reliable communication in harsh electromagnetic environments.
   - Differential signaling, along with proper termination and shielding techniques, helps mitigate the effects of electromagnetic interference (EMI) and noise on the bus.

CAN Error States/Frames

CAN (Controller Area Network) error frames are special frames transmitted on the CAN bus to signal various error conditions that may occur during communication. These error frames help maintain the integrity and reliability of the CAN bus by providing mechanisms for error detection and recovery. Here are the main types of CAN error frames:

1. **Error Active State (ERR_ACTIVE)**:

   - This state indicates that no errors have been detected on the CAN bus, and the network is operating normally.
   - In this state, the CAN controller sends six dominant bits after detecting an error. When the limit is below TEC[NM1] (Transmit Error Counter) <127 and REC (Receive Error Counter) <127, an Error Active node will transmit Active Error Flags when it detects errors.
   - In the error active state, nodes on the bus can continue transmitting and receiving messages without any issues.

2. **Error Passive State (ERR_PASSIVE)**:

   - When errors are detected/occurred on the CAN bus above a threshold, a node may transition to the error passive state.
   - When any one of the two Error Counters raises above 127(TEC>127 and REC>127), the node will enter a state known as Error Passive.

- In this state, the node is still able to transmit messages, but it observes stricter error handling rules. For example, it may delay transmission attempts or limit the number of retransmissions.

3. **Bus Off State (BUS_OFF)**:
   - If the number of transmission errors exceeds a certain threshold within a specified time period, a node may enter the bus off state.
   - When the Transmit Error Counter raises above 255(TEC>255 and REC>255), the node will enter the Bus Off state. It happens when the CAN controller fails or at times of extreme accumulations of errors.
   - In this case, the CAN controller disconnects itself from the CAN bus. As a result, that particular CAN node will get switched off from the CAN network; Resulting in no communication at all (until it undergoes a recovery process).

4. **Error Frames**:
   - Error frames are special messages transmitted on the bus to indicate the occurrence of errors.
   - There are two types of error frames: active error frames and passive error frames.
     - **Active Error Frame**: This frame is transmitted when a node detects a bus error while in the error active state. It consists of six dominant bits (logical 0s) followed by an error delimiter bit.
     - **Passive Error Frame**: This frame is transmitted when a node detects a bus error while in the error passive state. It consists of six recessive bits (logical 1s) followed by an error delimiter bit. Receiving nodes don't globalize such errors (do not transmit again).

5. **Error Detection and Handling**:
   - CAN nodes continuously monitor the bus for errors, such as bit errors, frame errors, and stuff errors.
   - When an error is detected, the node takes appropriate actions based on its error handling rules, such as transitioning to a different error state or transmitting error frames.

CAN Mailboxes

In the context of the CAN (Controller Area Network) protocol, a mailbox refers to a storage buffer or memory location within a CAN controller that is used to send or receive messages. Mailboxes play a crucial role in both transmission and reception processes in CAN communication. Here's how they work:

1. **Transmission with Mailboxes**:

- **Message Buffering**: When a node wants to transmit a message onto the CAN bus, it writes the message data, including the identifier and data bytes, into a transmit mailbox within its CAN controller.
- **Arbitration and Transmission**: The CAN controller then initiates the transmission process by attempting to access the bus. During arbitration, the CAN controller compares the identifiers of messages in the transmit mailboxes to determine the priority of transmission.
- **Sending the Message**: If the node wins arbitration, the message stored in the highest-priority transmit mailbox is transmitted onto the bus. The CAN controller manages the timing and framing of the message transmission, including handling collisions and retries if necessary.

2. **Reception with Mailboxes**:
   - **Message Filtering**: When a node receives a message from the CAN bus, the CAN controller checks the message identifier against a set of receive filters configured for each receive mailbox. These filters determine which messages are accepted and stored in the receive mailboxes.
   - **Storing Received Messages**: If a received message matches one of the configured filters, the CAN controller stores the message data, including the identifier and data bytes, in the corresponding receive mailbox.
   - **Notification and Retrieval**: The node's software can then access the receive mailboxes to retrieve and process the received messages. The CAN controller may also generate interrupts or trigger software flags to notify the processor of newly received messages in the mailboxes.
3. **Role in Communication**:
   - **Buffering and Queuing**: Mailboxes act as buffers or queues for messages waiting to be transmitted or processed. They provide temporary storage for message data within the CAN controller, allowing for efficient handling of communication tasks.
   - **Priority Handling**: Mailboxes help manage the priority of messages during transmission by providing a mechanism for arbitration based on message identifiers. This ensures that higher-priority messages are transmitted before lower-priority ones, helping maintain deterministic behavior in the communication network.
   - **Filtering and Segregation**: Receive mailboxes enable filtering and segregation of incoming messages based on their identifiers. This allows nodes to selectively process messages of interest while ignoring others, improving system efficiency and reducing processor overhead.

CAN programming

- Clock Setup

    - HSE clock
    - HCLK = 72 MHz
    - APB1 PCLK = 36 MHz
- CAN1 Basic config
    - Activate
    - Select RX & TX pins: PB8, PB9
    - Prescalar = 18
    - CAN time quanta = PCLK / Prescalar
        - = 18 / 36 M = 500 nsec
    - Time Quatas
        - Bit Seg 1 -- 2 Quanta
        - Bit Seg 2 -- 1 Quanta
        - SJW Seg -- 1 Quanta
    - Bit timing = 4 Quantas * Time Qunta = 2000 ns.
    - Baud Rate = 1 / Bit timing = 1 / 2000 ns = 500000 (bits per sec)
    - Mode = Loopback mode
    - Interrupt Enable = RX0 interrupt -- RX0 FIFO Pending message
- CAN Initialization (In main())
    - MX_CAN1_Init(...);

```
CAN_FilterTypeDef FilterConfig;
FilterConfig.FilterActivation = CAN_FILTER_ENABLE;
FilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0;
FilterConfig.SlaveStartFilterBank = 14;
FilterConfig.FilterBank = 10;        // Any number from 0 to SlaveStartFilterBank
FilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
FilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;    //CAN_FILTERMODE_IDLIST;
FilterConfig.FilterMaskIdLow = 0x0000;
FilterConfig.FilterMaskIdHigh = 0x07F8 << 5;        //id1 = 0x07FA << 5
FilterConfig.FilterIdLow = 0x0000;
FilterConfig.FilterIdHigh = 0x00A8 << 5;            //id2 = 0x07FB << 5
```

```
HAL_CAN_ConfigFilter(&hcan1, &FilterConfig);
```

  - HAL_CAN_Start(...);
- CAN Receiving:
  - Acceptance Filter Config (In MX_CAN1_Init --> After CAN init)
  - HAL_CAN_ActivateNotification(...); (After HAL_CAN_Start());
  - Callback function: HAL_CAN_RxFifo0MsgPendingCallback(...);

```
CAN_RxHeaderTypeDef RxHeader;
uint8_t RxData[8];

void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan) {
   HAL_CAN_GetRxMessage(&hcan1, CAN_RX_FIFO0, &RxHeader, RxData);
}
```

- CAN Transmit (In main())
  - Put Msg metadata into TxHeader.
  - Put Msg data in TxData -- array.
  - Send message (into mailbox using HAL_CAN_AddTxMessage()).

```
CAN_TxHeaderTypeDef TxHeader1, TxHeader2, TxHeader3;
uint32_t TxMailbox1, TxMailbox2, TxMailbox3;
uint8_t TxData1[8], TxData2[8], TxData3[8];

HAL_CAN_Start(&hcan1);
// for receiving messages
HAL_CAN_ActivateNotification(&hcan1, CAN_IT_RX_FIFO0_MSG_PENDING);

TxHeader1.TransmitGlobalTime = DISABLE;
```

```
TxHeader1.IDE = CAN_ID_STD;
TxHeader1.ExtId = 0;
TxHeader1.StdId = 0x0A9;
TxHeader1.RTR = CAN_RTR_DATA;
TxHeader1.DLC = 1;

TxData1[0] = 0x11;
if(HAL_CAN_AddTxMessage(&hcan1, &TxHeader1, TxData1, &TxMailbox1) != HAL_OK)
   Error_Handler();

TxHeader2.TransmitGlobalTime = DISABLE;
TxHeader2.IDE = CAN_ID_STD;
TxHeader2.ExtId = 0;
TxHeader2.StdId = 0x0AD;
TxHeader2.RTR = CAN_RTR_DATA;
TxHeader2.DLC = 1;

TxData2[0] = 0x22;
if(HAL_CAN_AddTxMessage(&hcan1, &TxHeader2, TxData2, &TxMailbox2) != HAL_OK)
   Error_Handler();

TxHeader3.TransmitGlobalTime = DISABLE;
TxHeader3.IDE = CAN_ID_STD;
TxHeader3.ExtId = 0;
TxHeader3.StdId = 0x0A8;
TxHeader3.RTR = CAN_RTR_DATA;
TxHeader3.DLC = 8;

strcpy((char*)TxData3, "SUNBEAM");
if(HAL_CAN_AddTxMessage(&hcan1, &TxHeader3, TxData3, &TxMailbox3) != HAL_OK)
   Error_Handler();
```