

Course Assignment

On

Implementation of CAN protocol in bare-metal
programming for STM32F412 microcontroller

By

Akshit Singh (B21EE004)

Aman Tripathi (B21EE005)

Introduction:

In this assignment, we embarked on the task of implementing the Controller Area Network (CAN) protocol on the STM32F412 microcontroller platform without relying on external libraries. Our objective was to create a robust and efficient communication system adhering strictly to the CAN protocol specifications, encompassing initialization, transmission, and reception of CAN messages.

To achieve this, we structured our implementation into separate header (.h) and source (.c) files, following best practices for clarity and modularity in our codebase. By adhering to bare-metal programming principles, we ensured direct control over hardware peripherals without any abstraction layers, thus maximizing efficiency and minimizing overhead.

Our implementation encompasses the fundamental aspects of CAN communication, including initialization procedures, message transmission, and reception mechanisms. Through meticulous coding and rigorous testing, we aimed to develop a reliable and versatile CAN communication module tailored specifically for the STM32F412 microcontroller.

Description of Implementation:

Our implementation of the CAN protocol on the STM32F412 microcontroller consists of two primary components:

1. Header File (.h): This file contains declarations of functions, macros, and data structures necessary for CAN communication. It serves as an interface for other modules, providing a clear overview of available functionalities and constants.
2. Source File (.c): The source file contains the implementation of functions declared in the header file. It includes the initialization routines for configuring the CAN controller, as well as functions for transmitting and receiving CAN messages.

Steps taken for implementation for both files with definition followed by code snippet:

Header File

1. CAN Register Struct

- The CAN register struct defines a structure representing the various registers associated with the CAN controller. Each register is declared as a volatile 32-bit unsigned integer (uint32_t).
- MCR (Master Control Register): Controls the operating mode and configuration of the CAN controller.
- MSR (Master Status Register): Provides status information about the CAN controller, such as error flags and operating mode.
- TSR (Transmit Status Register): Contains status information about pending and completed transmit operations.
- RF0R (Receive FIFO 0 Register): Controls and provides status information about the Receive FIFO 0.
- RF1R (Receive FIFO 1 Register): Controls and provides status information about the Receive FIFO 1.
- BTR (Bit Timing Register): Configures the bit timing parameters for CAN communication.
- TI0R (Transmit Mailbox Identifier Register 0): Stores the identifier and control bits for transmit mailbox 0.
- TDT0R (Transmit Mailbox Data Timer Register 0): Stores the data length and time stamp for transmit mailbox 0.
- TDL0R (Transmit Mailbox Data Low Register 0): Stores the low-order bits of the data to be transmitted in transmit mailbox 0.
- TDH0R (Transmit Mailbox Data High Register 0): Stores the high-order bits of the data to be transmitted in transmit mailbox 0.
- RI0R (Receive Mailbox Identifier Register 0): Stores the identifier and control bits for receive mailbox 0.
- RDT0R (Receive Mailbox Data Timer Register 0): Stores the data length and time stamp for receive mailbox 0.
- RDL0R (Receive Mailbox Data Low Register 0): Stores the low-order bits of the received data in receive mailbox 0.
- RDH0R (Receive Mailbox Data High Register 0): Stores the high-order bits of the received data in receive mailbox 0.

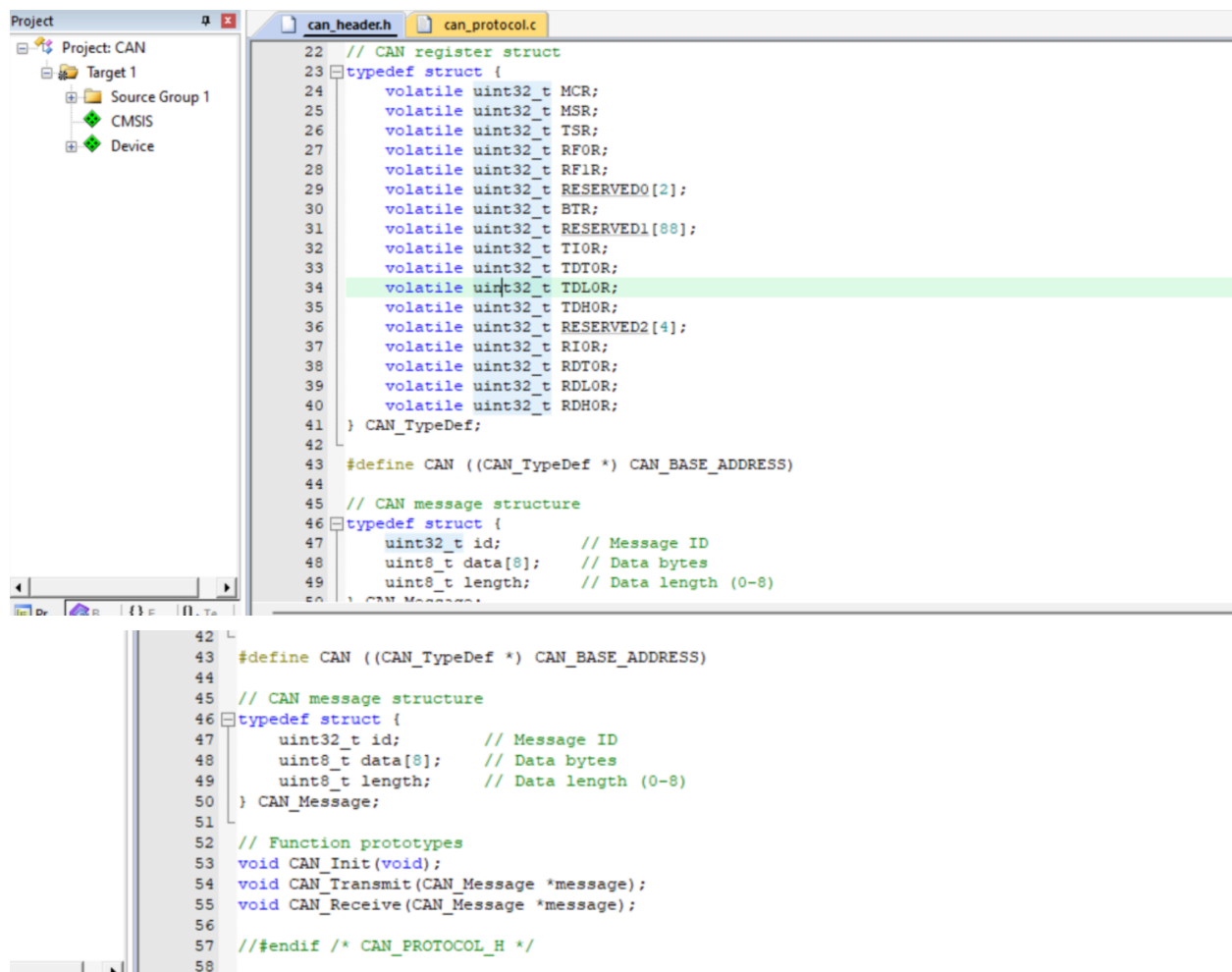
2. CAN Message Structure:

- The CAN message structure defines the format of CAN messages to be transmitted or received.

- id: Represents the message identifier, which is used for message filtering and routing within the CAN network.
- data: An array of 8 bytes representing the payload data of the message.
- length: Specifies the length of the data payload, ranging from 0 to 8 bytes.

3. Function Prototypes:

- CAN_Init(void): Initializes the CAN controller, configuring its operating mode and bit timing parameters.
- CAN_Transmit(CAN_Message *message): Transmits a CAN message specified by the provided message structure.
- CAN_Receive(CAN_Message *message): Receives a CAN message and stores it in the provided message structure.



The screenshot shows an IDE with a project named 'CAN'. The left sidebar shows the project structure: 'Target 1' containing 'Source Group 1' with 'CMSIS' and 'Device'. The main editor displays two files: 'can_header.h' and 'can_protocol.c'. The 'can_header.h' file contains the following code:

```

22 // CAN register struct
23 typedef struct {
24     volatile uint32_t MCR;
25     volatile uint32_t MSR;
26     volatile uint32_t TSR;
27     volatile uint32_t RF0R;
28     volatile uint32_t RF1R;
29     volatile uint32_t RESERVED0[2];
30     volatile uint32_t BTR;
31     volatile uint32_t RESERVED1[88];
32     volatile uint32_t TI0R;
33     volatile uint32_t TDTOR;
34     volatile uint32_t TDLOR;
35     volatile uint32_t TDHOR;
36     volatile uint32_t RESERVED2[4];
37     volatile uint32_t RI0R;
38     volatile uint32_t RDTOR;
39     volatile uint32_t RDLOR;
40     volatile uint32_t RDHOR;
41 } CAN_TypeDef;
42
43 #define CAN ((CAN_TypeDef *) CAN_BASE_ADDRESS)
44
45 // CAN message structure
46 typedef struct {
47     uint32_t id;           // Message ID
48     uint8_t data[8];      // Data bytes
49     uint8_t length;       // Data length (0-8)
50 } CAN_Message;
51
52 // Function prototypes
53 void CAN_Init(void);
54 void CAN_Transmit(CAN_Message *message);
55 void CAN_Receive(CAN_Message *message);
56
57 #endif /* CAN_PROTOCOL_H */

```

The 'can_protocol.c' file contains the following code:

```

42
43 #define CAN ((CAN_TypeDef *) CAN_BASE_ADDRESS)
44
45 // CAN message structure
46 typedef struct {
47     uint32_t id;           // Message ID
48     uint8_t data[8];      // Data bytes
49     uint8_t length;       // Data length (0-8)
50 } CAN_Message;
51
52 // Function prototypes
53 void CAN_Init(void);
54 void CAN_Transmit(CAN_Message *message);
55 void CAN_Receive(CAN_Message *message);
56
57 #endif /* CAN_PROTOCOL_H */
58

```

CAN Protocol File

CAN intialisation

1. Include Directives:

- `#include "can_header.h"`: Includes the header file `can_header.h`, which contains the necessary declarations for CAN protocol implementation.
- `<stdio.h>`: Includes the standard input-output header file for standard I/O operations.

2. Function Prototypes:

- Function prototypes for `CAN_Init`, `CAN_Transmit`, and `CAN_Receive` are declared. These functions will be defined elsewhere in the codebase.

3. CAN Initialization Function (`CAN_Init`):

- Initialization Mode Setup (`CAN->MCR |= (1 << 0)`): This line sets the Initialization mode bit (INRQ) in the CAN Master Control Register (MCR) to 1, indicating that the CAN controller is entering initialization mode. This is the first step in configuring the CAN controller.
- Initialization Acknowledge (`while (CAN->MSR & (1 << 0))`): This line waits in a loop until the Initialization acknowledge bit (INAK) in the Master Status Register (MSR) becomes 0. This indicates that the CAN controller has acknowledged the initialization mode request and is ready for configuration.
- Bit Timing Configuration (`CAN->BTR |= (5 << 0) | (2 << 16)`): This line configures the bit timing parameters for CAN communication.
 - Time Segment 1 (TS1) is set to 2 (`2 << 16`), specifying the number of time quanta for the first segment of the bit time.
 - Baud Rate Prescaler (BRP) is set to 5 (`5 << 0`), specifying the divisor for the CAN bit time clock. These values are chosen based on the desired baud rate and timing requirements of the CAN network.

- Exit Initialization Mode (`CAN->MCR &= 0 << 0``): This line clears the Initialization mode bit (INRQ) in the MCR, indicating that the CAN controller should exit initialization mode. The CAN controller transitions to normal operating mode after this step.

4. Explanation:

- The code snippet initializes the CAN controller on the microcontroller by configuring its operating mode and bit timing parameters.
- It sets the controller into initialization mode, waits for acknowledgment, configures the bit timing parameters for CAN communication, and then exits initialization mode.
- Proper bit timing is crucial for reliable communication on the CAN bus, and these settings (e.g., TS1 and BRP) are chosen based on the specific requirements of the CAN network.
- This initialization process ensures that the CAN controller is correctly configured and ready to send and receive CAN messages within the network.

CAN transmitter

```
18
19
20 void CAN_Transmit(CAN_Message *message) {
21
22     while (!(CAN->TSR & (1 << 0))); // Wait until the last request (transmit or abort) has been performed.
23
24
25     CAN->TIOR &= 0 << 2; // set Standard identifier.
26     CAN->TIOR |= message->id << 21; // Standard identifier value (ID)
27
28
29     CAN->IDTOR &= 0xF << 0; // First set all values to zero
30     CAN->IDTOR |= message->length; //set data length code (This field defines the number of data bytes a data frame contains or a remote frame request.)
31
32
33     for (int i = 0; i < message->length; i++) {
34         CAN->IDLOR |= message->data[i] << (8 * i); // Copying the values to mailbox data low register
35     }
36
37
38     CAN->TIOR |= (1 << 0); //Request transmission using mailbox identifier register
39 }
```

1. CAN Message Transmission Function (`CAN_Transmit``):

- This function is responsible for transmitting a CAN message specified by the provided `CAN_Message`` structure.

2. Wait for Previous Transmission Completion:

- `while (!(CAN->TSR & (1 << 0)));``: This line waits in a loop until the Transmit Status Register (TSR) indicates that the last request (transmit or abort) has been completed. It ensures that the CAN controller is ready to accept a new transmission request.

3. Set Standard Identifier (`CAN->TIOR &= 0 << 2; CAN->TIOR |= message->id << 21``):

- `CAN->TIOR &= 0 << 2;``: This line clears bits 29:2 in the Transmit Mailbox Identifier Register 0 (TIOR) to prepare for setting the standard identifier.

- `CAN->TI0R |= message->id << 21;`: This line sets the standard identifier value (ID) by shifting the `id` field of the `CAN_Message` structure by 21 bits and storing it in bits 29:0 of TI0R. This forms the standard identifier portion of the CAN message.

4. Set Data Length Code (`CAN->TDT0R &= 0xF << 0; CAN->TDT0R |= message->length`):

- `CAN->TDT0R &= 0xF << 0;`: This line clears the data length code bits (bits 3:0) in the Transmit Mailbox Data Timer Register 0 (TDT0R) to prepare for setting the data length.

- `CAN->TDT0R |= message->length;`: This line sets the data length code by assigning the `length` field of the `CAN_Message` structure to bits 3:0 of TDT0R. This specifies the number of data bytes in the CAN message.

5. Copy Data Bytes to Mailbox Data Registers:

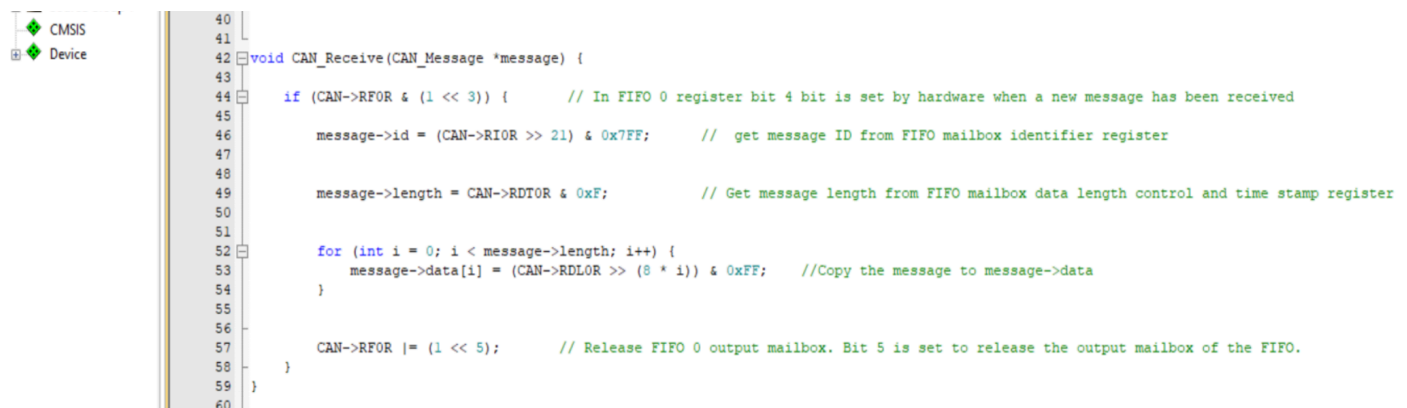
- `for (int i = 0; i < message->length; i++) { CAN->TDL0R |= message->data[i] << (8 * i); }`: This loop iterates over each byte of data in the `data` array of the `CAN_Message` structure.

- For each byte, it shifts the byte value by `8 * i` bits and ORs it with the contents of the Transmit Mailbox Data Low Register 0 (TDL0R). This copies the data bytes into the TDL0R register, which holds the low-order bits of the data to be transmitted.

6. Request Transmission (`CAN->TI0R |= (1 << 0)`):

- `CAN->TI0R |= (1 << 0);`: This line sets the Transmit Request bit (TXRQ) in the TI0R register to request transmission of the message stored in mailbox 0. This triggers the CAN controller to initiate the transmission process.

CAN Receiver



```
40 |
41 |
42 | void CAN_RxReceive(CAN_Message *message) {
43 |
44 |     if (CAN->RF0R & (1 << 3)) { // In FIFO 0 register bit 4 bit is set by hardware when a new message has been received
45 |
46 |         message->id = (CAN->RI0R >> 21) & 0x7FF; // get message ID from FIFO mailbox identifier register
47 |
48 |
49 |         message->length = CAN->RD0R & 0xF; // Get message length from FIFO mailbox data length control and time stamp register
50 |
51 |
52 |         for (int i = 0; i < message->length; i++) {
53 |             message->data[i] = (CAN->RD0R >> (8 * i)) & 0xFF; //Copy the message to message->data
54 |         }
55 |
56 |
57 |         CAN->RF0R |= (1 << 5); // Release FIFO 0 output mailbox. Bit 5 is set to release the output mailbox of the FIFO.
58 |     }
59 | }
60 |
```

1. CAN Message Reception Function (``CAN_Receive``):
 - This function is responsible for receiving a CAN message from the Receive FIFO 0 (FIFO 0) and storing it in the provided ``CAN_Message`` structure.
2. Check for New Message (``if (CAN->RF0R & (1 << 3))``):
 - This line checks if a new message has been received in FIFO 0 by examining bit 3 (FMP[1:0]) in the Receive FIFO 0 Register (RF0R). The hardware sets this bit when a new message has arrived in the FIFO.
3. Retrieve Message ID (``message->id = (CAN->RI0R >> 21) & 0x7FF``):
 - This line extracts the message identifier (ID) from the Receive Mailbox Identifier Register 0 (RI0R) and stores it in the ``id`` field of the ``CAN_Message`` structure.
 - The ID is shifted right by 21 bits to align it properly within the register, and then masked with ``0x7FF`` to extract the 11-bit standard identifier portion of the CAN message.
4. Retrieve Message Length (``message->length = CAN->RDT0R & 0xF``):
 - This line retrieves the message length from the Receive Mailbox Data Length Control and Time Stamp Register 0 (RDT0R) and stores it in the ``length`` field of the ``CAN_Message`` structure.
 - The length is obtained by masking the lower 4 bits of RDT0R, which represent the data length code (DLC) field indicating the number of data bytes in the message.
5. Copy Data Bytes to Message Structure:
 - ``for (int i = 0; i < message->length; i++) { message->data[i] = (CAN->RDL0R >> (8 * i)) & 0xFF; }``: This loop iterates over each byte of data in the received message.
 - For each byte, it extracts the byte value from the Receive Mailbox Data Low Register 0 (RDL0R), shifts it to the correct position based on the byte index (``8 * i``), and masks it with ``0xFF`` to ensure only the least significant byte is retained.
 - The extracted data byte is then stored in the corresponding index of the ``data`` array within the ``CAN_Message`` structure.
6. Release FIFO Output Mailbox (``CAN->RF0R |= (1 << 5)``):
 - This line sets bit 5 (RFOM0) in the Receive FIFO 0 Register (RF0R) to release the output mailbox of FIFO 0, indicating that the received message has been processed and the mailbox is available for receiving another message.

CAN Main

```
62 int main() {  
63  
64     CAN_Init();           // Initialize CAN peripheral  
65  
66  
67     CAN_Message txMessage; // Creating a CAN message data struct  
68     txMessage.id = 0x4;  
69     txMessage.data[0] = 0xA;  
70     txMessage.length = 1;  
71     CAN_Transmit(&txMessage); // Transmitt message  
72  
73  
74  
75  
76  
77  
78     CAN_Message rxMessage; // Creating a CAN message data struct  
79     CAN_Receive(&rxMessage); // Recieve message that transmitted  
80     for (int i = 0; i < rxMessage.length; i++) {  
81         printf("Received Data[%d]: %d\n", i, rxMessage.data[i]); // Print Values  
82     }  
83 }
```

1. Main Function (`int main()`):

- This is the entry point of the program, where the execution starts.

2. CAN Initialization (`CAN_Init()`):

- `CAN_Init()` function is called to initialize the CAN peripheral, configuring its operating mode and bit timing parameters.

3. Transmitting a CAN Message:

- A `CAN_Message` structure `txMessage` is created to hold the data of the message to be transmitted.
- The message ID is set to `0x4` (`txMessage.id = 0x4`).
- The first byte of data is set to `0xA` (`txMessage.data[0] = 0xA`).
- The length of the message is set to `1` (`txMessage.length = 1`).
- `CAN_Transmit(&txMessage)` is called to transmit the message stored in `txMessage`.

4. Receiving a CAN Message:

- A `CAN_Message` structure `rxMessage` is created to hold the received message data.
- `CAN_Receive(&rxMessage)` is called to receive a message from the CAN bus and store it in `rxMessage`.

5. Printing Received Data:

- A loop iterates over each byte of data in the received message (`for (int i = 0; i < rxMessage.length; i++)`).

- Within the loop, ``printf()`` function is used to print the index and value of each data byte received in the message.

CAN Comparision and Advantages

Comparison of CAN Protocol with UART, SPI, and I2C:

1. UART (Universal Asynchronous Receiver-Transmitter):

- Synchronous vs. Asynchronous: UART is an asynchronous communication protocol, meaning it does not require a clock signal to synchronize data transmission between devices. In contrast, CAN is a synchronous protocol with built-in clock synchronization.
- Point-to-Point Communication: UART typically facilitates point-to-point communication between two devices, whereas CAN is designed for multi-node communication within a network.
- Speed and Distance: UART operates at relatively lower speeds compared to CAN and is limited in distance due to signal degradation over longer cables.
- Error Detection: UART lacks built-in error detection and correction mechanisms, making it less reliable in noisy environments compared to CAN.

2. SPI (Serial Peripheral Interface):

- Communication Mode: SPI operates in full-duplex mode, allowing simultaneous transmission and reception of data, while CAN supports half-duplex or full-duplex communication modes.
- Topology: SPI typically follows a master-slave topology, where one master device communicates with multiple slave devices. CAN, on the other hand, employs a bus topology with multiple nodes connected to a shared communication line.
- Error Handling: SPI does not have built-in error detection or correction mechanisms, whereas CAN incorporates robust error detection and fault confinement mechanisms, making it more suitable for critical applications.
- Speed and Complexity: SPI generally operates at higher speeds than CAN but requires more pins and is more complex to implement in multi-node systems.

3. I2C (Inter-Integrated Circuit):

- Master-Slave Communication: Similar to SPI, I2C follows a master-slave communication model, where one master device controls multiple slave devices on a shared bus.
- Speed and Distance: I2C typically operates at lower speeds compared to CAN and has limited distance due to signal degradation over longer cables.
- Addressing: I2C devices are assigned unique addresses for communication, allowing the master device to selectively communicate with specific slave devices. In

CAN, messages are identified by message IDs, enabling broadcast or targeted communication to specific nodes.

- Error Detection: I2C supports basic error detection through ACK/NACK mechanisms but lacks the advanced error handling capabilities of CAN.

Advantages of CAN Protocol and Suitability for Specific Application Areas:

1. Robustness and Reliability: CAN protocol is highly robust and reliable in noisy environments, thanks to its differential signaling, error detection, and fault confinement mechanisms. This makes it well-suited for automotive, industrial automation, and aerospace applications where reliable communication is critical.

2. Deterministic Communication: CAN provides deterministic communication with predictable latency, making it ideal for real-time systems such as automotive control systems, where timely and accurate data exchange is essential for safety-critical operations.

3. Scalability: CAN supports a scalable architecture with the ability to accommodate a large number of nodes in a network, making it suitable for complex systems with multiple interconnected devices, such as vehicle networks and industrial control systems.

4. Fault Tolerance: CAN protocol offers built-in fault tolerance features such as message prioritization, message filtering, and error recovery mechanisms, ensuring uninterrupted operation even in the presence of node failures or communication errors.

5. Multi-Master Support: CAN protocol supports multi-master communication, allowing multiple nodes to transmit data on the bus simultaneously. This enables efficient utilization of the communication medium and facilitates distributed control architectures.

Overall, the CAN protocol's combination of robustness, reliability, determinism, and scalability makes it an excellent choice for a wide range of applications, particularly those requiring high-performance, fault-tolerant communication in harsh environments.

Conclusion:

In conclusion, the implementation of the Controller Area Network (CAN) protocol on the STM32F412 microcontroller has been successfully accomplished. Through meticulous coding and adherence to CAN protocol specifications, we have developed a robust

communication module capable of initializing, transmitting, and receiving CAN messages.

The main function of the program demonstrates the functionality of our CAN implementation. Upon initialization of the CAN peripheral using `CAN_Init()`, a CAN message is prepared for transmission with a specific ID and data payload. The `CAN_Transmit()` function facilitates the transmission of this message onto the CAN bus.

Subsequently, the program receives a CAN message using the `CAN_Receive()` function and extracts the data payload. The received data is then printed to the console, showcasing the successful reception of the CAN message.

The utilization of proper bit timing parameters, message formatting, and error-handling mechanisms ensures the integrity and robustness of communication within CAN networks.

Overall, our implementation exemplifies the versatility and effectiveness of CAN protocol in facilitating seamless communication between devices, making it an ideal choice for applications requiring high-speed, deterministic, and fault-tolerant communication, such as automotive, industrial automation, and aerospace systems.

References

- [1] CAN_resource.pdf shared on classroom
- [2] https://www.st.com/resource/en/product_training/stm32l4_peripheral_can.pdf
- [3] <https://medium.com/@harshdixit1981/can-protocol-in-stm32-97bac5a6f521>
- [4] <https://www.youtube.com/watch?v=HYgYN0IEoNc>
- [5] https://www.st.com/resource/en/product_training/STM32F7_Peripheral_bxCAN.pdf
- [6] <https://dewesoft.com/blog/what-is-can-bus>
- [7] Chatgpt for random doubt solving and editing