**PROJECT REPORT**

ON

**SOFTWARE DEVELOPMENT**

**Table of Content**

1. Develop firmware for ECUs using C/C++.

2. Implement CAN, LIN, and FlexRay protocol stacks.

3. Write device drivers for sensors and actuators.

4. Integrate error handling and diagnostic mechanisms.

**1. Develop firmware for ECUs using C/C++.**

What are the Developments required for this code and

**Required Tools and Environment**

 **Integrated Development Environment (IDE)**: Examples include STM32CubeIDE, Keil uVision, or Eclipse with the appropriate plugins.

 **Compiler**: GCC for ARM (arm-none-eabi-gcc) or the compiler provided by your chosen IDE.

 **Debugger**: JTAG/SWD debugger (e.g., ST-Link, J-Link).

**Microcontroller and Hardware:**

* **Development Board**: An STM32, NXP, or similar microcontroller development board with CAN, LIN, and ADC capabilities.
* **Peripheral Components**: Sensors, actuators, and communication modules for testing.

**Libraries and Middleware:**

* **HAL (Hardware Abstraction Layer)**: Provided by the microcontroller manufacturer (e.g., STM32 HAL).
* **Communication Protocol Stacks**: CAN, LIN, FlexRay libraries.
* **RTOS (Real-Time Operating System)**: Optional, but useful for managing tasks (e.g., FreeRTOS).

**Development Steps:**

1. **Setup Development Environment:**
   * Install your chosen IDE and configure it for your target microcontroller.
   * Set up the compiler and debugger settings.
2. **Initialize the Project:**
   * Create a new project in the IDE and select the appropriate microcontroller.
   * Generate initialization code using STM32CubeMX or another configuration tool if available.
3. **Develop Initialization Code:**
   * Initialize system clocks, GPIOs, and communication peripherals (CAN, LIN, FlexRay).
   * Example for CAN initialization:

#include <stdint.h>

#include <stdbool.h>

#include <stdio.h>

#include "stm32f4xx\_hal.h"

// CAN handle structure

CAN\_HandleTypeDef hcan;

// Configuration structure for CAN initialization parameters

typedef struct {

uint32\_t Prescaler;

uint32\_t Mode;

uint32\_t SyncJumpWidth;

uint32\_t TimeSeg1;

uint32\_t TimeSeg2;

bool TimeTriggeredMode;

bool AutoBusOff;

bool AutoWakeUp;

bool AutoRetransmission;

bool ReceiveFifoLocked;

bool TransmitFifoPriority;

} CAN\_Config;

// Function to initialize the CAN peripheral

void CAN\_Init(CAN\_Config \*config) {

hcan.Instance = CAN1; // Use CAN1 peripheral

// Apply configuration settings

hcan.Init.Prescaler = config->Prescaler;

hcan.Init.Mode = config->Mode;

hcan.Init.SyncJumpWidth = config->SyncJumpWidth;

hcan.Init.TimeSeg1 = config->TimeSeg1;

hcan.Init.TimeSeg2 = config->TimeSeg2;

hcan.Init.TimeTriggeredMode = config->TimeTriggeredMode ? ENABLE : DISABLE;

hcan.Init.AutoBusOff = config->AutoBusOff ? ENABLE : DISABLE;

hcan.Init.AutoWakeUp = config->AutoWakeUp ? ENABLE : DISABLE;

hcan.Init.AutoRetransmission = config->AutoRetransmission ? ENABLE : DISABLE;

hcan.Init.ReceiveFifoLocked = config->ReceiveFifoLocked ? ENABLE : DISABLE;

hcan.Init.TransmitFifoPriority = config->TransmitFifoPriority ? ENABLE : DISABLE;

// Initialize the CAN peripheral

if (HAL\_CAN\_Init(&hcan) != HAL\_OK) {

printf("CAN initialization failed!\n");

} else {

printf("CAN initialized successfully.\n");

}

}

// Example usage

int main(void) {

// Define the configuration settings for CAN initialization

CAN\_Config can\_config = {

.Prescaler = 16,

.Mode = CAN\_MODE\_NORMAL,

.SyncJumpWidth = CAN\_SJW\_1TQ,

.TimeSeg1 = CAN\_BS1\_1TQ,

.TimeSeg2 = CAN\_BS2\_1TQ,

.TimeTriggeredMode = false,

.AutoBusOff = false,

.AutoWakeUp = false,

.AutoRetransmission = true,

.ReceiveFifoLocked = false,

.TransmitFifoPriority = false

};

// Initialize the CAN peripheral with the provided settings

CAN\_Init(&can\_config);

// Main application loop

while (1) {

// Your main application code here

}

return 0;

}

### Explanation:

1. **CAN\_Config Structure**: We define a CAN\_Config structure to hold the initialization parameters, making it easier to modify and understand each parameter's purpose.
2. **CAN\_Init Function**: The CAN\_Init function takes a pointer to a CAN\_Config structure, applies the settings, and initializes the CAN peripheral.
3. **Example Usage**: The main function demonstrates how to use the CAN\_Init function with a specific configuration. This makes it simple for anyone to customize the CAN initialization parameters.

**Implement Communication Protocols:**

* Develop functions to transmit and receive data using CAN, LIN, and FlexRay.
* Example for CAN transmission and reception:

#include <stdint.h>

#include <stdbool.h>

#include <stdio.h>

#include "stm32f4xx\_hal.h"

// CAN handle structure

CAN\_HandleTypeDef hcan;

// Structure to hold CAN messages

typedef struct {

uint32\_t id; // CAN message ID

uint8\_t data[8]; // CAN message data

uint8\_t length; // Data length code (DLC)

} CAN\_Message;

// Function to transmit a CAN message

void CAN\_Transmit(CAN\_Message \*msg) {

CAN\_TxHeaderTypeDef TxHeader;

uint32\_t TxMailbox;

// Fill the transmission header with message information

TxHeader.StdId = msg->id; // Standard identifier

TxHeader.ExtId = 0; // No extended identifier

TxHeader.RTR = CAN\_RTR\_DATA; // Data frame (not a remote frame)

TxHeader.IDE = CAN\_ID\_STD; // Standard ID (not extended)

TxHeader.DLC = msg->length; // Data length code

// Add the message to the transmit mailbox and check for errors

if (HAL\_CAN\_AddTxMessage(&hcan, &TxHeader, msg->data, &TxMailbox) != HAL\_OK) {

printf("Error sending CAN message\n");

} else {

printf("CAN message sent\n");

}

}

// Function to receive a CAN message

bool CAN\_Receive(CAN\_Message \*msg) {

CAN\_RxHeaderTypeDef RxHeader;

uint8\_t data[8];

// Check if a message is available in the receive FIFO

if (HAL\_CAN\_GetRxMessage(&hcan, CAN\_RX\_FIFO0, &RxHeader, data) == HAL\_OK) {

msg->id = RxHeader.StdId; // Get the message ID

msg->length = RxHeader.DLC; // Get the data length code

for (int i = 0; i < RxHeader.DLC; i++) {

msg->data[i] = data[i]; // Copy the data

}

printf("Received CAN message with ID 0x%03X\n", msg->id);

return true; // Message received successfully

}

return false; // No message received

}

// Example usage

int main(void) {

// Initialize the CAN peripheral

CAN\_Init();

// Create a CAN message to transmit

CAN\_Message tx\_msg = {0x100, {0x01, 0x02, 0x03}, 3};

CAN\_Transmit(&tx\_msg);

// Receive a CAN message

CAN\_Message rx\_msg;

if (CAN\_Receive(&rx\_msg)) {

// Process the received message (if any)

}

// Main application loop

while (1) {

// Your main application code here

}

return 0;

}

**Explanation of This Code:**

#### 1. ****Including Necessary Headers****:

c

#include <stdint.h>

#include <stdbool.h>

#include <stdio.h>

#include "stm32f4xx\_hal.h"

* **stdint.h**: Provides fixed-width integer types (e.g., uint32\_t, uint8\_t).
* **stdbool.h**: Provides Boolean type and values (true, false).
* **stdio.h**: Provides input and output functions (e.g., printf).
* **stm32f4xx\_hal.h**: Includes STM32 HAL library functions for hardware abstraction.

#### 2. ****Defining CAN Handle Structure****:

c

CAN\_HandleTypeDef hcan;

* **hcan**: This structure is used to manage and configure the CAN peripheral. It is an instance of CAN\_HandleTypeDef provided by the HAL library.

#### 3. ****Defining CAN Message Structure****:

c

typedef struct {

uint32\_t id; // CAN message ID

uint8\_t data[8]; // CAN message data

uint8\_t length; // Data length code (DLC)

} CAN\_Message;

* **CAN\_Message**: Custom structure to store CAN message information, including message ID, data, and data length code (DLC).

#### 4. ****CAN Transmit Function****:

c

void CAN\_Transmit(CAN\_Message \*msg) {

CAN\_TxHeaderTypeDef TxHeader;

uint32\_t TxMailbox;

// Fill the transmission header with message information

TxHeader.StdId = msg->id;

TxHeader.ExtId = 0;

TxHeader.RTR = CAN\_RTR\_DATA;

TxHeader.IDE = CAN\_ID\_STD;

TxHeader.DLC = msg->length;

// Add the message to the transmit mailbox and check for errors

if (HAL\_CAN\_AddTxMessage(&hcan, &TxHeader, msg->data, &TxMailbox) != HAL\_OK) {

printf("Error sending CAN message\n");

} else {

printf("CAN message sent\n");

}

}

* **CAN\_Transmit**: Function to transmit a CAN message.
  + **TxHeader**: CAN transmission header structure, filled with message details (ID, data type, ID type, DLC).
  + **TxMailbox**: Variable to store the mailbox number used for transmission.
  + **HAL\_CAN\_AddTxMessage**: HAL function to add the message to the CAN transmit mailbox. It returns a status indicating success or failure.
  + **printf**: Outputs the result of the transmission attempt.

#### 5. ****CAN Receive Function****:

c

bool CAN\_Receive(CAN\_Message \*msg) {

CAN\_RxHeaderTypeDef RxHeader;

uint8\_t data[8];

// Check if a message is available in the receive FIFO

if (HAL\_CAN\_GetRxMessage(&hcan, CAN\_RX\_FIFO0, &RxHeader, data) == HAL\_OK) {

msg->id = RxHeader.StdId;

msg->length = RxHeader.DLC;

for (int i = 0; i < RxHeader.DLC; i++) {

msg->data[i] = data[i];

}

printf("Received CAN message with ID 0x%03X\n", msg->id);

return true; // Message received successfully

}

return false; // No message received

}

* **CAN\_Receive**: Function to receive a CAN message.
  + **RxHeader**: CAN reception header structure, stores received message details.
  + **HAL\_CAN\_GetRxMessage**: HAL function to retrieve a message from the CAN receive FIFO.
  + **printf**: Outputs the received message details.

#### 6. ****Main Function****:

int main(void) {

// Initialize the CAN peripheral

CAN\_Init();

// Create a CAN message to transmit

CAN\_Message tx\_msg = {0x100, {0x01, 0x02, 0x03}, 3};

CAN\_Transmit(&tx\_msg);

// Receive a CAN message

CAN\_Message rx\_msg;

if (CAN\_Receive(&rx\_msg)) {

// Process the received message (if any)

}

// Main application loop

while (1) {

// Your main application code here

}

return 0;

}

* **CAN\_Init**: Placeholder function to initialize the CAN peripheral (not shown in provided code).
* **tx\_msg**: CAN message to be transmitted, initialized with ID 0x100, data {0x01, 0x02, 0x03}, and length 3.
* **CAN\_Transmit**: Transmits the tx\_msg.
* **CAN\_Receive**: Receives a CAN message into rx\_msg and processes it (processing code not shown).
* **while (1)**: Infinite loop representing the main application loop.

1. **Overview**:
   * The provided code demonstrates basic CAN communication using the STM32 HAL library. It includes functions for initializing, transmitting, and receiving CAN messages.
2. **Initialization**:
   * The CAN\_Init function (not provided) is called to initialize the CAN peripheral before any communication occurs.
3. **Transmission**:
   * The CAN\_Transmit function takes a CAN\_Message structure, fills a CAN\_TxHeaderTypeDef with message details, and uses HAL\_CAN\_AddTxMessage to transmit the message. If the transmission is successful, it prints "CAN message sent"; otherwise, it prints an error message.
4. **Reception**:
   * The CAN\_Receive function checks for a message in the receive FIFO using HAL\_CAN\_GetRxMessage. If a message is available, it fills the provided CAN\_Message structure with the received data and prints the message ID. It returns true if a message was received, or false otherwise.

**Example Usage:**

* Initializes the CAN peripheral.
* Creates a CAN message and transmits it.
* Checks for and processes any received CAN messages.
* Contains the main application loop where additional code can be added.

**Develop Sensor and Actuator Drivers:**

* Implement drivers for reading sensors and controlling actuators.
* Example for ADC reading:

void ADC\_Init(void) {

// Configure ADC resolution, input channels, etc.

}

uint16\_t ADC\_Read(uint8\_t channel) {

// Trigger ADC conversion and wait for result

return 0; // Dummy value

}

**Implement Control Logic:**

* Develop the main control logic to handle sensor readings, protocol communication, and actuator control.
* Example main loop:

int main(void) {

System\_Init();

Peripheral\_Init();

while (1) {

// Main ECU loop

// Handle sensor reading, protocol communication, and diagnostics

CAN\_Example();

LIN\_Example();

Sensor\_Read();

}

return 0;

}

**Explanation of this code**

 int main(void) {

System\_Init();

Peripheral\_Init();

* System\_Init(): This function initializes the overall system. It typically sets up the microcontroller, configures clocks, and prepares the environment for peripheral initialization.
* Peripheral\_Init(): This function initializes the peripherals connected to the system. This might include setting up communication interfaces like CAN (Controller Area Network), LIN (Local Interconnect Network), sensors, and other hardware components.

 **Main Loop**:

while (1) {

* while (1): This creates an infinite loop, ensuring that the ECU (Electronic Control Unit) runs continuously. The loop keeps the system operational, allowing it to handle tasks repeatedly.

 **Core Tasks within the Loop**:

c

// Main ECU loop

// Handle sensor reading, protocol communication, and diagnostics

CAN\_Example();

LIN\_Example();

Sensor\_Read();

* **CAN\_Example()**: This function likely handles communication using the CAN protocol. It might send and receive CAN messages, process incoming data, and ensure proper communication with other nodes in the CAN network.
* **LIN\_Example()**: Similar to CAN\_Example(), this function manages LIN protocol communication. It handles data transmission and reception over the LIN bus.
* **Sensor\_Read()**: This function reads data from various sensors connected to the ECU. It might include acquiring temperature, pressure, or other relevant sensor data and processing it for further use.

 **Return Statement**:

}

return 0;

* return 0;: This line is technically redundant since the infinite loop will never allow the program to reach this point. However, it signifies successful program termination if the loop were to exit.

**Integrating Error Handling and Diagnostic Mechanics**

### 1. Error Handling

#### A. ****General Error Handling Principles****

* **Fail Gracefully**: Ensure your application can continue running or shut down safely if an error occurs.
* **Provide Informative Error Messages**: Errors should be clear and provide actionable information for debugging.
* **Log Errors**: Maintain a log of errors to aid in diagnosing issues and tracking patterns.

#### B. ****Techniques in Code****

* **Using Try-Catch Blocks**:

try {

// Code that might throw an exception

} catch (const std::exception& e) {

printf("Error: %s\n", e.what());

// Handle exception

}

* **Error Codes and Messages**:

enum ErrorCode {

SUCCESS,

ERROR\_INIT,

ERROR\_TRANSMIT,

ERROR\_RECEIVE

};

const char\* getErrorMessage(ErrorCode code) {

switch (code) {

case SUCCESS: return "Success";

case ERROR\_INIT: return "Initialization Error";

case ERROR\_TRANSMIT: return "Transmission Error";

case ERROR\_RECEIVE: return "Reception Error";

default: return "Unknown Error";

}

}

### 2. Diagnostic Mechanisms

#### A. ****Self-Tests****

* **Routine Checks**: Implement self-test routines that check the health of various components on startup and periodically.
* **Example**:

bool runSelfTests() {

bool success = true;

success &= checkSensor();

success &= checkActuator();

// Additional checks...

return success;

}

#### B. ****Health Monitoring****

* **System Metrics**: Monitor metrics such as CPU usage, memory usage, and response times to assess system health.
* **Example**:

void monitorHealth() {

uint32\_t cpu\_usage = getCPUUsage();

uint32\_t memory\_usage = getMemoryUsage();

printf("CPU Usage: %u%%, Memory Usage: %uKB\n", cpu\_usage, memory\_usage);

}

#### C. ****Error Reporting****

* **Automatic Reports**: Implement automatic error reporting to collect data on issues and improve the system.
* **Example**:

void reportError(ErrorCode code) {

printf("Reporting Error: %s\n", getErrorMessage(code));

// Send error report to server or log it

}

### Integration Example:

Here’s how you can integrate error handling and diagnostic mechanisms into your main application code:

int main(void) {

// Initialize the CAN peripheral

if (!CAN\_Init()) {

reportError(ERROR\_INIT);

return -1;

}

// Create a CAN message to transmit

CAN\_Message tx\_msg = {0x100, {0x01, 0x02, 0x03}, 3};

if (!CAN\_Transmit(&tx\_msg)) {

reportError(ERROR\_TRANSMIT);

}

// Self-test and health monitoring

if (!runSelfTests()) {

reportError(ERROR\_INIT);

}

monitorHealth();

// Main application loop

while (1) {

// Receive a CAN message

CAN\_Message rx\_msg;

if (CAN\_Receive(&rx\_msg)) {

// Process the received message (if any)

} else {

reportError(ERROR\_RECEIVE);

}

// Additional application code...

}

return 0;

**The total Entire of this code in CAN, LIN, FlexRay protocols:**

#include <stdint.h>

#include <stdbool.h>

#include <stdio.h>

// ---------- CAN Protocol Stack ----------

void CAN\_Init() {

    printf("CAN protocol initialized.\n");

}

void CAN\_Transmit(uint32\_t id, uint8\_t \*data, uint8\_t length) {

    printf("CAN Transmit: ID=0x%X, Data=", id);

    for (uint8\_t i = 0; i < length; i++) {

        printf(" 0x%X", data[i]);

    }

    printf("\n");

}

bool CAN\_Receive(uint32\_t \*id, uint8\_t \*data, uint8\_t \*length) {

    \*id = 0x123; // Example ID

    \*length = 3;

    data[0] = 0xAA;

    data[1] = 0xBB;

    data[2] = 0xCC;

    printf("CAN Received: ID=0x%X, Data=0x%X 0x%X 0x%X\n", \*id, data[0], data[1], data[2]);

    return true; // Indicate that a message was received

}

// ---------- LIN Protocol Stack ----------

void LIN\_Init() {

    printf("LIN protocol initialized.\n");

}

void LIN\_Transmit(uint8\_t id, uint8\_t \*data, uint8\_t length) {

    printf("LIN Transmit: ID=0x%X, Data=", id);

    for (uint8\_t i = 0; i < length; i++) {

        printf(" 0x%X", data[i]);

    }

    printf("\n");

}

bool LIN\_Receive(uint8\_t \*id, uint8\_t \*data, uint8\_t \*length) {

    \*id = 0x01; // Example ID

    \*length = 2;

    data[0] = 0x11;

    data[1] = 0x22;

    printf("LIN Received: ID=0x%X, Data=0x%X 0x%X\n", \*id, data[0], data[1]);

    return true; // Indicate that a frame was received

}

// ---------- FlexRay Protocol Stack ----------

void FlexRay\_Init() {

    printf("FlexRay protocol initialized.\n");

}

void FlexRay\_Transmit(uint16\_t slot\_id, uint8\_t \*data, uint8\_t length) {

    printf("FlexRay Transmit: Slot ID=%d, Data=", slot\_id);

    for (uint8\_t i = 0; i < length; i++) {

        printf(" 0x%X", data[i]);

    }

    printf("\n");

}

bool FlexRay\_Receive(uint16\_t \*slot\_id, uint8\_t \*data, uint8\_t \*length) {

    \*slot\_id = 256; // Example Slot ID

    \*length = 4;

    data[0] = 0x55;

    data[1] = 0x66;

    data[2] = 0x77;

    data[3] = 0x88;

    printf("FlexRay Received: Slot ID=%d, Data=0x%X 0x%X 0x%X 0x%X\n",

           \*slot\_id, data[0], data[1], data[2], data[3]);

    return true; // Indicate that a message was received

}

// ---------- Main Function for Testing ----------

int main() {

    // CAN Test

    CAN\_Init();

    uint8\_t can\_data[] = {0x01, 0x02, 0x03};

    CAN\_Transmit(0x100, can\_data, 3);

    uint32\_t can\_rx\_id;

    uint8\_t can\_rx\_data[8];

    uint8\_t can\_rx\_length;

    if (CAN\_Receive(&can\_rx\_id, can\_rx\_data, &can\_rx\_length)) {

        // Process received CAN message

    }

    // LIN Test

    LIN\_Init();

    uint8\_t lin\_data[] = {0x10, 0x20};

    LIN\_Transmit(0x01, lin\_data, 2);

    uint8\_t lin\_rx\_id;

    uint8\_t lin\_rx\_data[8];

    uint8\_t lin\_rx\_length;

    if (LIN\_Receive(&lin\_rx\_id, lin\_rx\_data, &lin\_rx\_length)) {

        // Process received LIN frame

    }

    // FlexRay Test

    FlexRay\_Init();

    uint8\_t flexray\_data[] = {0xDE, 0xAD, 0xBE, 0xEF};

    FlexRay\_Transmit(256, flexray\_data, 4);

    uint16\_t flexray\_rx\_slot\_id;

    uint8\_t flexray\_rx\_data[8];

    uint8\_t flexray\_rx\_length;

    if (FlexRay\_Receive(&flexray\_rx\_slot\_id, flexray\_rx\_data, &flexray\_rx\_length)) {

        // Process received FlexRay message

    }

    return 0;

}

**The output of this code and Explanation:**

CAN protocol initialized.

CAN Transmit: ID=0x100, Data= 0x1 0x2 0x3

CAN Received: ID=0x123, Data=0xAA 0xBB 0xCC

LIN protocol initialized.

LIN Transmit: ID=0x1, Data= 0x10 0x20

LIN Received: ID=0x1, Data=0x11 0x22

FlexRay protocol initialized.

FlexRay Transmit: Slot ID=256, Data= 0xDE 0xAD 0xBE 0xEF

FlexRay Received: Slot ID=256, Data=0x55 0x66 0x77 0x88

#include <stdint.h>

#include <stdbool.h>

#include <stdio.h>

These headers provide standard integer types, boolean values, and standard input/output functions.

2. **CAN Protocol Functions**:

**CAN\_Init**: Initializes the CAN protocol and prints a confirmation.

void CAN\_Init() { printf("CAN protocol initialized.\n");

**CAN\_Transmit**: Transmits a CAN message with a given ID and data, printing the transmitted data.

void CAN\_Transmit(uint32\_t id, uint8\_t \*data, uint8\_t length) {

printf("CAN Transmit: ID=0x%X, Data=", id);

for (uint8\_t i = 0; i < length; i++) { printf(" 0x%X", data[i]);

}

printf("\n");

* **CAN\_Receive**: Receives a CAN message and prints the received data.

c

bool CAN\_Receive(uint32\_t \*id, uint8\_t \*data, uint8\_t \*length) {

\*id = 0x123; // Example ID

\*length = 3;

data[0] = 0xAA;

data[1] = 0xBB;

data[2] = 0xCC;

printf("CAN Received: ID=0x%X, Data=0x%X 0x%X 0x%X\n", \*id, data[0], data[1], data[2]);

return true; // Indicate that a message was received

}

#### 3. ****LIN Protocol Functions****:

* **LIN\_Init**: Initializes the LIN protocol and prints a confirmation.

c

void LIN\_Init() {

printf("LIN protocol initialized.\n");

}

* **LIN\_Transmit**: Transmits a LIN message with a given ID and data, printing the transmitted data.

c

void LIN\_Transmit(uint8\_t id, uint8\_t \*data, uint8\_t length) {

printf("LIN Transmit: ID=0x%X, Data=", id);

for (uint8\_t i = 0; i < length; i++) {

printf(" 0x%X", data[i]);

}

printf("\n");

}

* **LIN\_Receive**: Receives a LIN message and prints the received data.

c

bool LIN\_Receive(uint8\_t \*id, uint8\_t \*data, uint8\_t \*length) {

\*id = 0x01; // Example ID

\*length = 2;

data[0] = 0x11;

data[1] = 0x22;

printf("LIN Received: ID=0x%X, Data=0x%X 0x%X\n", \*id, data[0], data[1]);

return true; // Indicate that a frame was received

}

#### 4. ****FlexRay Protocol Functions****:

* **FlexRay\_Init**: Initializes the FlexRay protocol and prints a confirmation.

c

void FlexRay\_Init() {

printf("FlexRay protocol initialized.\n");

}

* **FlexRay\_Transmit**: Transmits a FlexRay message with a given slot ID and data, printing the transmitted data.

c

void FlexRay\_Transmit(uint16\_t slot\_id, uint8\_t \*data, uint8\_t length) {

printf("FlexRay Transmit: Slot ID=%d, Data=", slot\_id);

for (uint8\_t i = 0; i < length; i++) {

printf(" 0x%X", data[i]);

}

printf("\n");

}

* **FlexRay\_Receive**: Receives a FlexRay message and prints the received data.

c

bool FlexRay\_Receive(uint16\_t \*slot\_id, uint8\_t \*data, uint8\_t \*length) {

\*slot\_id = 256; // Example Slot ID

\*length = 4;

data[0] = 0x55;

data[1] = 0x66;

data[2] = 0x77;

data[3] = 0x88;

printf("FlexRay Received: Slot ID=%d, Data=0x%X 0x%X 0x%X 0x%X\n", \*slot\_id, data[0], data[1], data[2], data[3]);

return true; // Indicate that a message was received

}

#### 5. ****Main Function****:

* Tests the CAN, LIN, and FlexRay protocols by initializing them, transmitting, and receiving messages.

c

int main() {

// CAN Test

CAN\_Init();

uint8\_t can\_data[] = {0x01, 0x02, 0x03};

CAN\_Transmit(0x100, can\_data, 3);

uint32\_t can\_rx\_id;

uint8\_t can\_rx\_data[8];

uint8\_t can\_rx\_length;

if (CAN\_Receive(&can\_rx\_id, can\_rx\_data, &can\_rx\_length)) {

// Process received CAN message

}

// LIN Test

LIN\_Init();

uint8\_t lin\_data[] = {0x10, 0x20};

LIN\_Transmit(0x01, lin\_data, 2);

uint8\_t lin\_rx\_id;

uint8\_t lin\_rx\_data[8];

uint8\_t lin\_rx\_length;

if (LIN\_Receive(&lin\_rx\_id, lin\_rx\_data, &lin\_rx\_length)) {

// Process received LIN frame

}

// FlexRay Test

FlexRay\_Init();

uint8\_t flexray\_data[] = {0xDE, 0xAD, 0xBE, 0xEF};

FlexRay\_Transmit(256, flexray\_data, 4);

uint16\_t flexray\_rx\_slot\_id;

uint8\_t flexray\_rx\_data[8];

uint8\_t flexray\_rx\_length;

if (FlexRay\_Receive(&flexray\_rx\_slot\_id, flexray\_rx\_data, &flexray\_rx\_length)) {

// Process received FlexRay message

}

return 0;

}