4 loại frame truyền trong giao thức CAN

Trong giao thức **CAN (Controller Area Network)**, có 4 loại frame chính: **Data Frame**, **Remote Frame**, **Error Frame**, và **Overload Frame**. Mỗi loại frame có vai trò riêng trong quá trình truyền thông giữa các node trên mạng CAN. Dưới đây là giải thích chi tiết về từng loại, các trường hợp xảy ra và vai trò của chúng trong hệ thống.

1. Data Frame (Khung dữ liệu)

Giải thích:

Data Frame là khung phổ biến nhất được sử dụng trong giao thức **CAN (Controller Area Network)**. Nó được sử dụng để truyền dữ liệu thực tế giữa các node trên mạng CAN. **Data Frame** bao gồm các thông tin về địa chỉ của node gửi và nhận, kích thước dữ liệu, và chính dữ liệu cần truyền. Frame này giúp đảm bảo rằng dữ liệu sẽ được truyền đúng cách và bảo vệ khỏi lỗi bằng cách sử dụng mã kiểm tra CRC.

Cấu trúc của Data Frame:

1. **Start of Frame (SOF)**: 1 bit để báo hiệu bắt đầu của khung truyền.
2. **Arbitration Field**: Chứa địa chỉ của node gửi hoặc node nhận. Có thể là 11-bit (Identifier chuẩn) hoặc 29-bit (Identifier mở rộng), giúp phân biệt giữa các node. Trường này cũng chứa bit **RTR** để xác định kiểu khung là Data Frame hay Remote Frame.
3. **Control Field**: Chứa **DLC (Data Length Code)**, chỉ ra số byte dữ liệu trong khung, từ 0 đến 8 byte.
4. **Data Field**: Chứa dữ liệu thực tế cần truyền. Độ dài từ 0 đến 8 byte tùy thuộc vào giá trị của DLC.
5. **CRC Field (Cyclic Redundancy Check)**: Dùng để phát hiện lỗi trong quá trình truyền thông qua mạng.
6. **ACK Field (Acknowledgment)**: Node nhận sẽ gửi tín hiệu ACK để xác nhận rằng dữ liệu đã được nhận thành công.
7. **End of Frame (EOF)**: 7 bit kết thúc khung.

Khi nào xảy ra:

Data Frame xảy ra khi một node cần truyền dữ liệu đến các node khác trên mạng CAN. Các ứng dụng phổ biến của **Data Frame** bao gồm giao tiếp giữa các vi điều khiển, cảm biến, hoặc thiết bị điều khiển trong các hệ thống nhúng.

Ví dụ tình huống sử dụng:

- Một cảm biến nhiệt độ trên mạng CAN gửi giá trị nhiệt độ đến bộ điều khiển trung tâm.
- Một bộ điều khiển trong xe hơi gửi lệnh điều khiển đến các mô-tơ hoặc thiết bị truyền động (actuator) để thay đổi trạng thái cơ khí.

Ví dụ cụ thể: Gửi Data Frame từ STM32F103 sử dụng SPL

Trong ví dụ này, chúng ta sẽ cấu hình CAN trên **STM32F103** để gửi một **Data Frame** chứa 8 byte dữ liệu từ một node đến một node khác trên mạng CAN. Dưới đây là các bước cụ thể:

Bước 1: Cấu hình CAN trên STM32F103 sử dụng SPL

```
#include "stm32f10x.h"
#include "stm32f10x_can.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"

void CAN_Configuration(void) {
    CAN_InitTypeDef        CAN_InitStructure;
    CAN_FilterInitTypeDef  CAN_FilterInitStructure;
    GPIO_InitTypeDef       GPIO_InitStructure;

    // Enable CAN, GPIO clock
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

    // Configure CAN RX (PA11) and TX (PA12)
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;  // CAN RX
pin
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
```

```
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;  // CAN TX
pin
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    // CAN cell init
    CAN_InitStructure.CAN_TTCM = DISABLE;  // No
time-triggered communication mode
    CAN_InitStructure.CAN_ABOM = ENABLE;   // Automatic
bus-off management
    CAN_InitStructure.CAN_AWUM = ENABLE;   // Automatic
wake-up mode
    CAN_InitStructure.CAN_NART = DISABLE;  // No
automatic retransmission
    CAN_InitStructure.CAN_RFLM = DISABLE;  // No RX FIFO
locked mode
    CAN_InitStructure.CAN_TXFP = ENABLE;   // TX FIFO
priority
    CAN_InitStructure.CAN_Mode = CAN_Mode_Normal;  //
Normal mode
    CAN_InitStructure.CAN_SJW = CAN_SJW_1tq;  // 1 time
quantum for SJW
    CAN_InitStructure.CAN_BS1 = CAN_BS1_6tq;  // 6 time
quantum before sample point
    CAN_InitStructure.CAN_BS2 = CAN_BS2_8tq;  // 8 time
quantum after sample point
    CAN_InitStructure.CAN_Prescaler = 6;  // Prescaler
for 500 Kbps
    CAN_Init(CAN1, &CAN_InitStructure);

    // CAN filter init
    CAN_FilterInitStructure.CAN_FilterNumber = 0;
    CAN_FilterInitStructure.CAN_FilterMode =
CAN_FilterMode_IdMask;
```

```
    CAN_FilterInitStructure.CAN_FilterScale =
CAN_FilterScale_32bit;
    CAN_FilterInitStructure.CAN_FilterIdHigh = 0x0000;
    CAN_FilterInitStructure.CAN_FilterIdLow = 0x0000;
    CAN_FilterInitStructure.CAN_FilterMaskIdHigh =
0x0000;
    CAN_FilterInitStructure.CAN_FilterMaskIdLow = 0x0000;
    CAN_FilterInitStructure.CAN_FilterFIFOAssignment =
CAN_FIFO0;
    CAN_FilterInitStructure.CAN_FilterActivation =
ENABLE;
    CAN_FilterInit(&CAN_FilterInitStructure);
}
```

Bước 2: Gửi Data Frame

```
void CAN_TransmitData(uint8_t* data, uint8_t length) {
    CanTxMsg TxMessage;

    TxMessage.StdId = 0x123;  // 11-bit standard ID
    TxMessage.RTR = CAN_RTR_DATA;  // Data Frame
    TxMessage.IDE = CAN_ID_STD;  // Standard ID frame
    TxMessage.DLC = length;  // Number of data bytes

    // Copy data to be sent into TxMessage's data field
    for (int i = 0; i < length; i++) {
        TxMessage.Data[i] = data[i];
    }

    // Transmit the CAN message
    CAN_Transmit(CAN1, &TxMessage);

    // Wait for transmission to complete
    while (CAN_TransmitStatus(CAN1,
CAN_TransmitStatus_Complete) != CAN_TxStatus_Ok);
```

```
}
```

Bước 3: Main Program (ví dụ thực tế)

```
int main(void) {
    uint8_t dataToSend[8] = {0x01, 0x02, 0x03, 0x04,
0x05, 0x06, 0x07, 0x08};

    CAN_Configuration();  // Cấu hình CAN

    while (1) {
        CAN_TransmitData(dataToSend, 8);  // Gửi Data
Frame với 8 byte dữ liệu
        for (volatile int i = 0; i < 500000; i++);  //
Delay để tránh quá tải
    }
}
```

Tình huống sử dụng:

- **Data Frame** này có thể được dùng trong một mạng CAN của xe hơi, nơi mà bộ điều khiển trung tâm (ECU) gửi lệnh điều khiển các mô-tơ hoặc cảm biến trong hệ thống. Ví dụ, nếu đây là một hệ thống điều khiển ghế xe, lệnh điều khiển có thể yêu cầu mô-tơ di chuyển ghế đến vị trí mới.
- Trong ứng dụng công nghiệp, **Data Frame** có thể được sử dụng để một cảm biến gửi dữ liệu về nhiệt độ hoặc áp suất tới bộ điều khiển chính để thực hiện các hành động điều chỉnh.

2. Remote Frame (Khung yêu cầu)

Giải thích:

Remote Frame là một loại khung trong giao thức **CAN** được sử dụng để yêu cầu một node khác gửi dữ liệu qua **Data Frame**. Khác với **Data Frame**, **Remote Frame** không chứa dữ liệu thực tế trong trường **Data Field**, mà chỉ yêu

cầu node khác gửi lại một **Data Frame** có cùng định danh (Identifier). Một trong những đặc điểm quan trọng của **Remote Frame** là nó thiết lập bit **RTR (Remote Transmission Request)** để phân biệt với **Data Frame**.

Cấu trúc của Remote Frame:

1. **Start of Frame (SOF)**: 1 bit báo hiệu bắt đầu của khung.
2. **Arbitration Field**: Bao gồm định danh (11-bit hoặc 29-bit Identifier) và bit **RTR** được đặt thành 1 để báo đây là **Remote Frame**.
3. **Control Field**: Chứa **DLC (Data Length Code)**, chỉ ra số byte dữ liệu mà **Data Frame** sẽ chứa.
4. **CRC Field (Cyclic Redundancy Check)**: Được sử dụng để kiểm tra tính toàn vẹn của khung.
5. **ACK Field (Acknowledgment)**: Sử dụng để xác nhận rằng khung đã được nhận.
6. **End of Frame (EOF)**: 7 bit kết thúc khung.

Khi nào xảy ra:

Remote Frame xảy ra khi một node muốn yêu cầu dữ liệu từ một node khác trên mạng CAN mà không tự động nhận dữ liệu. Node gửi sẽ phát **Remote Frame**, sau đó node nhận sẽ trả lời bằng một **Data Frame** chứa dữ liệu được yêu cầu.

Ví dụ tình huống sử dụng:

- Một **node điều khiển trung tâm** muốn yêu cầu một **cảm biến** gửi dữ liệu hiện tại (ví dụ như dữ liệu nhiệt độ). **Node điều khiển** sẽ phát một **Remote Frame** với định danh tương ứng và cảm biến sẽ phản hồi lại bằng một **Data Frame** chứa dữ liệu.
- Trong hệ thống xe hơi, ECU có thể phát một **Remote Frame** để yêu cầu dữ liệu từ các cảm biến vị trí bánh xe để kiểm tra trạng thái vận hành.

Ví dụ cụ thể: Gửi Remote Frame từ STM32F103 sử dụng SPL

Trong ví dụ này, chúng ta sẽ cấu hình CAN trên **STM32F103** để gửi một **Remote Frame** nhằm yêu cầu một **Data Frame** từ một node khác. Dưới đây là các bước cấu hình cụ thể.

Bước 1: Cấu hình CAN trên STM32F103 sử dụng SPL

```
#include "stm32f10x.h"
#include "stm32f10x_can.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"

void CAN_Configuration(void) {
    CAN_InitTypeDef        CAN_InitStructure;
    CAN_FilterInitTypeDef  CAN_FilterInitStructure;
    GPIO_InitTypeDef       GPIO_InitStructure;

    // Enable CAN và GPIO clock
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

    // Configure CAN RX (PA11) và TX (PA12)
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;  // CAN RX
pin
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;  //
Input pull-up
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;  // CAN TX
pin
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;  //
Alternate function push-pull
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    // CAN cell init
    CAN_InitStructure.CAN_TTCM = DISABLE;  // No
time-triggered communication mode
    CAN_InitStructure.CAN_ABOM = ENABLE;   // Automatic
bus-off management
    CAN_InitStructure.CAN_AWUM = ENABLE;   // Automatic
wake-up mode
```

```
    CAN_InitStructure.CAN_NART = DISABLE;  // No
automatic retransmission
    CAN_InitStructure.CAN_RFLM = DISABLE;  // No RX FIFO
locked mode
    CAN_InitStructure.CAN_TXFP = ENABLE;   // TX FIFO
priority
    CAN_InitStructure.CAN_Mode = CAN_Mode_Normal;  // Chế
độ hoạt động bình thường
    CAN_InitStructure.CAN_SJW = CAN_SJW_1tq;  // 1 time
quantum for SJW
    CAN_InitStructure.CAN_BS1 = CAN_BS1_6tq;  // 6 time
quantum before sample point
    CAN_InitStructure.CAN_BS2 = CAN_BS2_8tq;  // 8 time
quantum after sample point
    CAN_InitStructure.CAN_Prescaler = 6;  // Prescaler
cho baudrate 500 Kbps
    CAN_Init(CAN1, &CAN_InitStructure);

    // CAN filter init
    CAN_FilterInitStructure.CAN_FilterNumber = 0;
    CAN_FilterInitStructure.CAN_FilterMode =
CAN_FilterMode_IdMask;
    CAN_FilterInitStructure.CAN_FilterScale =
CAN_FilterScale_32bit;
    CAN_FilterInitStructure.CAN_FilterIdHigh = 0x0000;
    CAN_FilterInitStructure.CAN_FilterIdLow = 0x0000;
    CAN_FilterInitStructure.CAN_FilterMaskIdHigh =
0x0000;
    CAN_FilterInitStructure.CAN_FilterMaskIdLow = 0x0000;
    CAN_FilterInitStructure.CAN_FilterFIFOAssignment =
CAN_FIFO0;
    CAN_FilterInitStructure.CAN_FilterActivation =
ENABLE;
    CAN_FilterInit(&CAN_FilterInitStructure);
}
```

Bước 2: Gửi Remote Frame

```
void CAN_TransmitRemoteRequest(uint16_t identifier,
uint8_t dlc) {
    CanTxMsg TxMessage;

    TxMessage.StdId = identifier;  // Định danh của node
mà bạn yêu cầu gửi Data Frame
    TxMessage.RTR = CAN_RTR_REMOTE;  // Thiết lập Remote
Frame
    TxMessage.IDE = CAN_ID_STD;  // Sử dụng chuẩn 11-bit
ID
    TxMessage.DLC = dlc;  // Data Length Code (số byte
dữ liệu mà bạn yêu cầu)

    // Gửi khung yêu cầu
    CAN_Transmit(CAN1, &TxMessage);

    // Đợi cho đến khi quá trình truyền hoàn tất
    while (CAN_TransmitStatus(CAN1,
CAN_TransmitStatus_Complete) != CAN_TxStatus_Ok);
}
```

Bước 3: Main Program (ví dụ thực tế)

```
int main(void) {
    CAN_Configuration();  // Cấu hình CAN

    while (1) {
        CAN_TransmitRemoteRequest(0x123, 8);  // Gửi
Remote Frame yêu cầu Data Frame với ID 0x123
        for (volatile int i = 0; i < 500000; i++);  //
Delay để tránh gửi quá nhanh
    }
}
```

Tình huống sử dụng:

- **Remote Frame** này có thể được sử dụng trong một mạng CAN công nghiệp để yêu cầu cảm biến gửi dữ liệu khi cần thiết. Ví dụ, một bộ điều khiển trung tâm có thể gửi **Remote Frame** với định danh của cảm biến nhiệt độ, yêu cầu nó gửi lại **Data Frame** chứa thông tin nhiệt độ hiện tại.
- Trong một hệ thống xe hơi, **Remote Frame** có thể được gửi bởi ECU để yêu cầu cảm biến vị trí của bánh xe gửi lại thông tin vị trí hiện tại, từ đó giúp điều khiển hệ thống phanh ABS hoặc điều chỉnh các thông số khác trong xe.

3. Error Frame (Khung lỗi)

Giải thích:

Error Frame là khung được tự động phát ra bởi một node CAN khi nó phát hiện lỗi trong quá trình giao tiếp. Khung này có mục đích thông báo cho các node khác trong mạng về việc phát hiện lỗi và yêu cầu quá trình truyền thông được khởi động lại. **Error Frame** không được người dùng gửi trực tiếp, mà phần cứng CAN sẽ tự động phát hiện và phát ra khi có lỗi.

Trong giao thức CAN, có hai loại **Error Frame**:

- **Active Error Frame**: Được phát ra bởi một node đang trong trạng thái **Active Error**, có thể can thiệp để sửa lỗi. Node này sẽ phát ra **6 bit Error Flag**.
- **Passive Error Frame**: Được phát ra bởi một node trong trạng thái **Passive Error**, khi nó đã gặp nhiều lỗi nhưng không thể sửa lỗi. **12 bit Error Flag** được gửi thay vì 6 bit.

Cấu trúc của Error Frame:

1. **Error Flag**: 6 hoặc 12 bit, phụ thuộc vào trạng thái lỗi (Active hoặc Passive).
2. **Error Delimiter**: 8 bit để phân tách Error Frame với các khung khác.

Khi nào xảy ra:

Error Frame xảy ra khi một node phát hiện một trong các lỗi sau trong quá trình truyền dữ liệu:

- **Bit Error**: Xảy ra khi một node phát hiện bit truyền ra không giống với bit nhận được.
- **CRC Error**: Xảy ra khi có lỗi trong quá trình kiểm tra mã CRC.
- **ACK Error**: Xảy ra khi node không nhận được tín hiệu ACK từ các node khác.
- **Form Error**: Xảy ra khi một trường trong khung không tuân theo định dạng đúng của giao thức CAN.
- **Stuff Error**: Xảy ra khi có nhiều hơn 5 bit giống nhau liên tiếp trong một khung (CAN sử dụng Bit stuffing để tránh điều này).

Khi bất kỳ lỗi nào được phát hiện, node phát ra **Error Frame** và các node khác trên mạng CAN sẽ tạm dừng quá trình giao tiếp và xử lý lại khung.

Ví dụ tình huống sử dụng:

- Một node phát hiện rằng tín hiệu ACK từ các node nhận không được phát đi đúng cách, gây ra lỗi **ACK Error**. Node sẽ phát **Error Frame** để cảnh báo các node khác và yêu cầu truyền lại dữ liệu.
- Một cảm biến trên mạng CAN bị lỗi truyền dữ liệu do môi trường bị nhiễu, gây ra lỗi **Bit Error** và phát **Error Frame** để các node khác biết rằng dữ liệu bị lỗi.

Ví dụ: Xử lý Error Frame trên STM32F103 sử dụng SPL

Trong ví dụ này, chúng ta sẽ cấu hình CAN trên **STM32F103** và cài đặt ngắt để xử lý các lỗi CAN. Khi phát hiện lỗi, **Error Frame** sẽ được tự động phát ra, và chúng ta sẽ xử lý sự kiện lỗi trong hàm ngắt.

Bước 1: Cấu hình CAN và kích hoạt ngắt lỗi

```
#include "stm32f10x.h"
#include "stm32f10x_can.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"
#include "misc.h"  // Cho ngắt

void CAN_Configuration(void) {
    CAN_InitTypeDef        CAN_InitStructure;
```

```
    CAN_FilterInitTypeDef  CAN_FilterInitStructure;
    GPIO_InitTypeDef       GPIO_InitStructure;
    NVIC_InitTypeDef       NVIC_InitStructure;

    // Enable CAN và GPIO clock
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

    // Configure CAN RX (PA11) và TX (PA12)
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;  // CAN RX
pin
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;  //
Input pull-up
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;  // CAN TX
pin
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;  //
Alternate function push-pull
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    // CAN cell init
    CAN_InitStructure.CAN_TTCM = DISABLE;  // No
time-triggered communication mode
    CAN_InitStructure.CAN_ABOM = ENABLE;   // Automatic
bus-off management
    CAN_InitStructure.CAN_AWUM = ENABLE;   // Automatic
wake-up mode
    CAN_InitStructure.CAN_NART = DISABLE;  // No
automatic retransmission
    CAN_InitStructure.CAN_RFLM = DISABLE;  // No RX FIFO
locked mode
    CAN_InitStructure.CAN_TXFP = ENABLE;   // TX FIFO
priority
    CAN_InitStructure.CAN_Mode = CAN_Mode_Normal;  //
```

```
Normal mode
    CAN_InitStructure.CAN_SJW = CAN_SJW_1tq;  // 1 time
quantum for SJW
    CAN_InitStructure.CAN_BS1 = CAN_BS1_6tq;  // 6 time
quantum before sample point
    CAN_InitStructure.CAN_BS2 = CAN_BS2_8tq;  // 8 time
quantum after sample point
    CAN_InitStructure.CAN_Prescaler = 6;  // Prescaler
cho baudrate 500 Kbps
    CAN_Init(CAN1, &CAN_InitStructure);

    // CAN filter init
    CAN_FilterInitStructure.CAN_FilterNumber = 0;
    CAN_FilterInitStructure.CAN_FilterMode =
CAN_FilterMode_IdMask;
    CAN_FilterInitStructure.CAN_FilterScale =
CAN_FilterScale_32bit;
    CAN_FilterInitStructure.CAN_FilterIdHigh = 0x0000;
    CAN_FilterInitStructure.CAN_FilterIdLow = 0x0000;
    CAN_FilterInitStructure.CAN_FilterMaskIdHigh =
0x0000;
    CAN_FilterInitStructure.CAN_FilterMaskIdLow = 0x0000;
    CAN_FilterInitStructure.CAN_FilterFIFOAssignment =
CAN_FIFO0;
    CAN_FilterInitStructure.CAN_FilterActivation =
ENABLE;
    CAN_FilterInit(&CAN_FilterInitStructure);

    // Kích hoạt ngắt CAN
    CAN_ITConfig(CAN1, CAN_IT_ERR, ENABLE);

    // Cấu hình ngắt trong NVIC
    NVIC_InitStructure.NVIC_IRQChannel =
USB_LP_CAN1_RX0_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority
= 0;
```

```
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
```

Bước 2: Xử lý lỗi trong ngắt CAN

```
void USB_LP_CAN1_RX0_IRQHandler(void) {
    // Kiểm tra xem có lỗi không
    if (CAN_GetITStatus(CAN1, CAN_IT_ERR) != RESET) {
        uint8_t error_code = CAN_GetLastErrorCode(CAN1);

        if (error_code == CAN_ErrorCode_BitRecessive) {
            // Xử lý lỗi bit error (node không gửi bit
đúng)
        } else if (error_code == CAN_ErrorCode_CRC) {
            // Xử lý lỗi CRC (kiểm tra CRC không thành
công)
        } else if (error_code == CAN_ErrorCode_Stuff) {
            // Xử lý lỗi stuff (nhiều hơn 5 bit giống
nhau liên tiếp)
        }
        // Xử lý các lỗi khác...

        // Xóa cờ lỗi
        CAN_ClearITPendingBit(CAN1, CAN_IT_ERR);
    }
}
```

Bước 3: Main Program

```
int main(void) {
```

```
    CAN_Configuration();  // Cấu hình CAN và kích hoạt
ngắt lỗi

    while (1) {
        // Chương trình chính
        // Mọi lỗi CAN sẽ được xử lý trong ngắt
    }
}
```

Tình huống sử dụng:

- Trong một mạng CAN công nghiệp, nếu một cảm biến nhiệt độ gặp phải lỗi **Bit Error** do nhiễu tín hiệu, nó sẽ phát ra **Error Frame** và chờ node khác gửi lại dữ liệu. Bộ điều khiển trung tâm có thể xử lý lỗi này và gửi lại khung dữ liệu đúng.
- Trong một hệ thống xe hơi, nếu tín hiệu ACK giữa các module CAN bị mất, lỗi **ACK Error** sẽ kích hoạt và **Error Frame** sẽ được phát ra. Node gặp lỗi sẽ cố gắng truyền lại khung dữ liệu.

Chi tiết về Active Error Frame và Passive Error Frame trong CAN

Trong giao thức **CAN**, việc phát hiện lỗi và xử lý lỗi là một trong những chức năng quan trọng để đảm bảo sự ổn định của hệ thống. Hai loại khung lỗi chính là **Active Error Frame** và **Passive Error Frame**, được phát ra khi một node phát hiện lỗi trong quá trình truyền hoặc nhận dữ liệu.

1. Active Error Frame:

- **Active Error Frame** được phát ra bởi một node đang trong trạng thái **Active Error**, tức là node đó vẫn còn khả năng phát hiện và sửa lỗi chủ động trong mạng CAN.
- Khi một node đang trong trạng thái **Active Error** phát hiện lỗi, nó sẽ phát ra một **Error Frame** với một **Error Flag** gồm **6 bit** liên tiếp có giá trị **0** (dominant bit). Điều này nhằm thông báo cho toàn bộ mạng rằng một lỗi đã xảy ra. Vì đây là một trạng thái chủ động, node này vẫn có quyền can

thiệp vào hoạt động của mạng và gửi lại các khung dữ liệu sau khi lỗi đã được giải quyết.

- **Active Error Frame** có thể được phát ra khi node vẫn còn trong trạng thái tin cậy, tức là node vẫn có thể giao tiếp và tham gia vào việc phát hiện và sửa lỗi.

Ví dụ:

- **Một cảm biến nhiệt độ** trong hệ thống công nghiệp phát hiện một **Bit Error** do tín hiệu CAN bị nhiễu. Node phát hiện lỗi này sẽ phát ra một **Active Error Frame** với **6 bit dominant**. Các node khác trên mạng sẽ tạm ngừng giao tiếp và xử lý lại quá trình truyền thông.

2. Passive Error Frame:

- **Passive Error Frame** được phát ra bởi một node đang trong trạng thái **Passive Error**, khi node này đã gặp quá nhiều lỗi và không thể tiếp tục can thiệp vào hoạt động của mạng.
- Khi một node trong trạng thái **Passive Error** phát hiện lỗi, nó sẽ phát ra **Error Flag** với **12 bit** (thay vì 6 bit trong **Active Error Frame**), bao gồm **6 bit dominant** và **6 bit recessive**. Điều này nhằm thông báo rằng node đó đã vượt qua giới hạn lỗi và không còn khả năng sửa lỗi chủ động.
- **Passive Error Frame** xảy ra khi một node đã tích lũy quá nhiều lỗi, vượt qua ngưỡng quy định và không thể tiếp tục sửa lỗi. Node này chỉ có thể phát ra các **Passive Error Frame** và không thể trực tiếp ảnh hưởng đến hoạt động của mạng bằng cách phát dominant bit.

Ví dụ:

- **Một mô-đun điều khiển động cơ** trong hệ thống xe hơi đã gặp quá nhiều lỗi do nhiễu sóng hoặc lỗi truyền thông lặp lại. Khi nó không còn khả năng sửa lỗi nữa, nó sẽ chuyển sang trạng thái **Passive Error**. Nếu lỗi tiếp tục xảy ra, nó sẽ phát ra **Passive Error Frame** với **12 bit Error Flag**, bao gồm cả dominant và recessive bit. Node này không còn quyền can thiệp vào giao tiếp mạng và chỉ có thể tiếp nhận thông tin mà không thể chủ động tham gia.

Chuyển đổi giữa các trạng thái Active và Passive

- **Error Counter**: Các node CAN quản lý một **Error Counter** để theo dõi số lượng lỗi mà chúng gặp phải. Mỗi khi một lỗi được phát hiện, giá trị của **Error Counter** sẽ tăng lên.
    - Khi **Error Counter** vượt qua ngưỡng **127**, node sẽ chuyển từ trạng thái **Active Error** sang **Passive Error**. Trong trạng thái này, node sẽ phát ra **Passive Error Frame** nếu phát hiện lỗi.
    - Nếu **Error Counter** tiếp tục tăng và vượt ngưỡng **255**, node sẽ chuyển sang trạng thái **Bus Off**, tức là node sẽ bị loại khỏi mạng CAN và không thể giao tiếp thêm cho đến khi được reset lại.

Tình huống sử dụng:

1. **Active Error Frame** có thể xảy ra trong những trường hợp ngắn hạn khi tín hiệu CAN bị nhiễu tạm thời, chẳng hạn như khi một dây cáp bị lỏng hoặc có nhiễu điện từ trong môi trường. Node phát hiện lỗi sẽ chủ động sửa lỗi bằng cách phát ra **Active Error Frame** và chờ cơ hội truyền lại dữ liệu.
2. **Passive Error Frame** xảy ra khi môi trường truyền thông có lỗi nghiêm trọng và kéo dài, như dây CAN bị đứt hoặc kết nối vật lý yếu. Node trong trạng thái này đã vượt qua giới hạn lỗi cho phép và chỉ có thể phát **Passive Error Frame**, không thể trực tiếp tham gia giao tiếp nữa.

Ví dụ thực tế về xử lý lỗi trong CAN với STM32F103 sử dụng SPL:

Trong mạng CAN sử dụng STM32F103, chúng ta có thể kiểm soát trạng thái lỗi thông qua **CAN Error Counter** và nhận dạng loại lỗi dựa trên trạng thái của node. Ví dụ sau đây minh họa cách giám sát và phát hiện **Active Error** và **Passive Error** bằng cách sử dụng các hàm SPL để đọc lỗi từ CAN.

Bước 1: Đọc và kiểm tra trạng thái lỗi

```
void check_CAN_error(void) {
    uint8_t error_code = CAN_GetLastErrorCode(CAN1);  // 
Lấy mã lỗi từ CAN

    if (error_code == CAN_ErrorCode_BitRecessive) {
        // Xử lý lỗi bit recessive (node phát hiện lỗi 
bit do không đồng bộ tín hiệu)
```

```
    } else if (error_code == CAN_ErrorCode_CRC) {
        // Xử lý lỗi CRC (kiểm tra CRC thất bại)
    } else if (error_code == CAN_ErrorCode_Stuff) {
        // Xử lý lỗi stuff (nhiều hơn 5 bit giống nhau
liên tiếp)
    }

    // Kiểm tra giá trị Error Counter
    uint8_t txErrorCounter =
CAN_GetLSBTransmitErrorCounter(CAN1);  // Lấy bộ đếm lỗi
truyền
    uint8_t rxErrorCounter =
CAN_GetReceiveErrorCounter(CAN1);  // Lấy bộ đếm lỗi nhận

    if (txErrorCounter > 127 || rxErrorCounter > 127) {
        // Node đang ở trạng thái Passive Error
        // Node đã gặp nhiều lỗi, không thể can thiệp
sửa lỗi
    } else if (txErrorCounter <= 127 && rxErrorCounter <=
127) {
        // Node đang ở trạng thái Active Error
        // Node vẫn có thể phát hiện và sửa lỗi chủ động
    }
}
```

Bước 2: Xử lý trong hàm ngắt khi phát hiện lỗi

```
void USB_LP_CAN1_RX0_IRQHandler(void) {
    if (CAN_GetITStatus(CAN1, CAN_IT_ERR) != RESET) {
        check_CAN_error();  // Gọi hàm kiểm tra và xử lý
lỗi
        CAN_ClearITPendingBit(CAN1, CAN_IT_ERR);  // Xóa
cờ lỗi
    }
}
```

Bước 3: Main Program

```
int main(void) {
    CAN_Configuration();  // Cấu hình CAN

    while (1) {
        // Kiểm tra và giám sát trạng thái lỗi
        check_CAN_error();
        for (volatile int i = 0; i < 500000; i++);  //
Delay để tránh kiểm tra quá thường xuyên
    }
}
```

4. Overload Frame (Khung quá tải)

Giải thích:

Overload Frame là một loại khung đặc biệt trong giao thức **CAN** được sử dụng để trì hoãn việc truyền dữ liệu khi một node trong mạng CAN cần thêm thời gian để xử lý. Khung này không chứa dữ liệu, mà chỉ báo hiệu rằng một node đang quá tải và cần thời gian trước khi tiếp tục giao tiếp. Mục tiêu của **Overload Frame** là ngăn không cho các khung khác được truyền quá nhanh, giúp node bị quá tải có đủ thời gian để xử lý các khung trước đó.

Overload Frame không phải do người dùng phát ra, mà được tự động phát ra bởi phần cứng CAN khi cần thiết. Node CAN sẽ phát ra **Overload Frame** khi một trong các điều kiện sau xảy ra:

- Node không thể xử lý tiếp dữ liệu do buffer đã đầy hoặc cần thêm thời gian xử lý dữ liệu.
- Node không thể nhận khung mới do có quá trình xử lý nội bộ cần hoàn thành trước.

Cấu trúc của Overload Frame:

1. **Overload Flag**: 6 bit **dominant** (bit 0) để báo hiệu trạng thái quá tải.

2. **Overload Delimiter**: 8 bit **recessive** (bit 1), để phân tách khung quá tải với các khung khác và báo hiệu kết thúc **Overload Frame**.

Các điều kiện có thể gây ra Overload Frame:

- **FIFO (First-In-First-Out) buffer** trong node nhận đầy và node cần thêm thời gian để xử lý dữ liệu đã nhận.
- **Tạm dừng nội bộ** trong node để hoàn tất việc xử lý một sự kiện trước khi nhận thêm khung dữ liệu mới.
- Node cần xử lý một yêu cầu cao cấp khác (như một yêu cầu từ phần mềm ứng dụng).

Khi nào xảy ra:

Overload Frame xảy ra khi một node cần thêm thời gian để xử lý khung dữ liệu đã nhận trước đó. Điều này ngăn các node khác truyền khung mới quá sớm, gây ra quá tải xử lý cho node đã phát ra **Overload Frame**.

- **Overload Frame** có thể được phát ra ngay sau **Intermission Field** (khoảng trống giữa các khung) hoặc ngay sau khi một khung dữ liệu đã được truyền hoàn tất, nếu node nhận không thể xử lý kịp thời.
- Trong thực tế, điều này thường xảy ra khi một node đang nhận nhiều dữ liệu từ nhiều nguồn khác nhau trên mạng CAN và cần tạm dừng để xử lý trước khi tiếp tục nhận thêm dữ liệu.

Ví dụ tình huống sử dụng:

- Trong một hệ thống **xe hơi**, module CAN trên vi điều khiển có thể phát ra **Overload Frame** khi phải nhận quá nhiều lệnh điều khiển từ nhiều bộ phận như cảm biến tốc độ, hệ thống phanh ABS và hệ thống điều hòa cùng một lúc. Việc phát **Overload Frame** giúp trì hoãn các lệnh mới cho đến khi module có thể xử lý xong lệnh trước đó.
- Trong hệ thống **công nghiệp**, một bộ điều khiển có thể phát ra **Overload Frame** khi phải xử lý một lượng lớn dữ liệu cảm biến cùng một lúc, chẳng hạn như từ các cảm biến nhiệt độ, áp suất, hoặc tốc độ, để ngăn chặn việc bị quá tải dữ liệu.

Ví dụ: Xử lý Overload Frame trong STM32F103 sử dụng SPL

Trong trường hợp **Overload Frame**, node phát tự động phát ra khung quá tải khi cần thêm thời gian xử lý dữ liệu. **Overload Frame** không được cấu hình hoặc kích hoạt bởi phần mềm mà được phần cứng CAN xử lý. Tuy nhiên, chúng ta có thể sử dụng ngắt để phát hiện các trạng thái CAN như **FIFO Full** hoặc các trạng thái khác có thể dẫn đến **Overload Frame**.

Bước 1: Cấu hình CAN và kích hoạt ngắt

```
#include "stm32f10x.h"
#include "stm32f10x_can.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"
#include "misc.h"  // Cho ngắt

void CAN_Configuration(void) {
    CAN_InitTypeDef        CAN_InitStructure;
    CAN_FilterInitTypeDef  CAN_FilterInitStructure;
    GPIO_InitTypeDef       GPIO_InitStructure;
    NVIC_InitTypeDef       NVIC_InitStructure;

    // Enable CAN và GPIO clock
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

    // Configure CAN RX (PA11) và TX (PA12)
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;  // CAN RX
pin
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;  //
Input pull-up
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;  // CAN TX
pin
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;  //
Alternate function push-pull
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
```

```
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    // CAN cell init
    CAN_InitStructure.CAN_TTCM = DISABLE;  // No
time-triggered communication mode
    CAN_InitStructure.CAN_ABOM = ENABLE;   // Automatic
bus-off management
    CAN_InitStructure.CAN_AWUM = ENABLE;   // Automatic
wake-up mode
    CAN_InitStructure.CAN_NART = DISABLE;  // No
automatic retransmission
    CAN_InitStructure.CAN_RFLM = ENABLE;   // RX FIFO
locked mode to avoid overflow
    CAN_InitStructure.CAN_TXFP = ENABLE;   // TX FIFO
priority
    CAN_InitStructure.CAN_Mode = CAN_Mode_Normal;  //
Normal mode
    CAN_InitStructure.CAN_SJW = CAN_SJW_1tq;  // 1 time
quantum for SJW
    CAN_InitStructure.CAN_BS1 = CAN_BS1_6tq;  // 6 time
quantum before sample point
    CAN_InitStructure.CAN_BS2 = CAN_BS2_8tq;  // 8 time
quantum after sample point
    CAN_InitStructure.CAN_Prescaler = 6;  // Prescaler
cho baudrate 500 Kbps
    CAN_Init(CAN1, &CAN_InitStructure);

    // CAN filter init
    CAN_FilterInitStructure.CAN_FilterNumber = 0;
    CAN_FilterInitStructure.CAN_FilterMode =
CAN_FilterMode_IdMask;
    CAN_FilterInitStructure.CAN_FilterScale =
CAN_FilterScale_32bit;
    CAN_FilterInitStructure.CAN_FilterIdHigh = 0x0000;
    CAN_FilterInitStructure.CAN_FilterIdLow = 0x0000;
    CAN_FilterInitStructure.CAN_FilterMaskIdHigh =
```

```
0x0000;
    CAN_FilterInitStructure.CAN_FilterMaskIdLow = 0x0000;
    CAN_FilterInitStructure.CAN_FilterFIFOAssignment =
CAN_FIFO0;
    CAN_FilterInitStructure.CAN_FilterActivation =
ENABLE;
    CAN_FilterInit(&CAN_FilterInitStructure);

    // Kích hoạt ngắt FIFO full để phát hiện trạng thái
có thể gây ra Overload Frame
    CAN_ITConfig(CAN1, CAN_IT_FF0, ENABLE);

    // Cấu hình ngắt trong NVIC
    NVIC_InitStructure.NVIC_IRQChannel =
USB_LP_CAN1_RX0_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority
= 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
```

Bước 2: Xử lý khi FIFO đầy, có khả năng phát ra Overload Frame

```
void USB_LP_CAN1_RX0_IRQHandler(void) {
    // Kiểm tra trạng thái FIFO Full
    if (CAN_GetITStatus(CAN1, CAN_IT_FF0) != RESET) {
        // Xử lý khi FIFO đầy
        // Lúc này phần cứng CAN sẽ tự động phát
Overload Frame để trì hoãn giao tiếp

        // Xóa cờ FIFO full
        CAN_ClearITPendingBit(CAN1, CAN_IT_FF0);
    }
}
```

Bước 3: Main Program

```
int main(void) {
    CAN_Configuration();  // Cấu hình CAN và kích hoạt
ngắt FIFO full

    while (1) {
        // Chương trình chính
        // Node sẽ tự động phát Overload Frame nếu cần
thêm thời gian xử lý dữ liệu
    }
}
```

Tình huống sử dụng:

- Trong một hệ thống **CAN công nghiệp**, nếu một node nhận nhiều dữ liệu từ các cảm biến hoặc bộ điều khiển khác nhau cùng một lúc, buffer nhận có thể bị đầy. Node này sẽ phát ra **Overload Frame** để báo cho các node khác rằng nó cần thêm thời gian để xử lý dữ liệu trước khi nhận thêm bất kỳ khung mới nào.
- Trong một hệ thống **xe hơi**, mô-đun điều khiển động cơ có thể gặp phải tình huống quá tải khi nhận nhiều dữ liệu điều khiển từ các bộ phận khác nhau cùng một lúc. **Overload Frame** sẽ được phát ra để trì hoãn các lệnh mới trong khi nó xử lý dữ liệu đã nhận trước đó.