# CAN trong STM32F103 sử dụng SPL (Standard Peripheral Library)

**Mục lục:**



## 1. Giới thiệu về giao thức CAN

**Controller Area Network (CAN)** là một giao thức truyền thông **dung sai lỗi** được sử dụng phổ biến trong các hệ thống nhúng, đặc biệt là trong ngành ô tô và công nghiệp. Giao thức này cho phép các vi điều khiển và thiết bị truyền và nhận dữ liệu trên một mạng chung mà không cần đến máy chủ trung tâm. **CAN** có thể hoạt động ổn định trong môi trường có nhiễu, do nó có cơ chế kiểm tra và sửa lỗi thông qua **CRC (Cyclic Redundancy Check)** và các khung lỗi như **Error Frame** và **Overload Frame**.

### 2. Tính năng CAN trên STM32F103

Vi điều khiển **STM32F103** tích hợp sẵn hai bộ điều khiển CAN (CAN1 và CAN2). Các tính năng chính của CAN trong STM32F103 bao gồm:

- Hỗ trợ cả chế độ **Normal** và **Loopback**.
- **11-bit Standard ID** và **29-bit Extended ID**.
- Hỗ trợ lên đến **1 Mbps**.
- Cơ chế **Error Management** và **Automatic Bus-Off Management**.
- **FIFO buffers** cho cả **RX** và **TX**, giúp quản lý hiệu quả dữ liệu truyền và nhận.
- Bộ lọc **CAN Filters** để lọc và xử lý khung dữ liệu mong muốn.

### 3. Cấu hình CAN sử dụng SPL

### 3.1 Cấu hình clock

Trước tiên, cần kích hoạt clock cho **CAN1** và các chân GPIO cần thiết (PA11 và PA12 trên STM32F103):

#include "stm32f10x.h"

```
void CAN_Clock_Configuration(void) {
    // Kích hoạt clock cho CAN1 và GPIOA
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
}
```

### 3.2 Cấu hình GPIO cho CAN

CAN sử dụng **PA11** (RX) và **PA12** (TX) để giao tiếp:

```
void CAN_GPIO_Configuration(void) {
    GPIO_InitTypeDef GPIO_InitStructure;

    // Cấu hình PA11 (CAN RX) là Input Pull-up
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    // Cấu hình PA12 (CAN TX) là Alternate Function
Push-Pull
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}
```

### 3.3 Cấu hình CAN

Dưới đây là cấu hình cho **CAN** trong **Normal Mode** với tốc độ baudrate là **500 Kbps**:

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

    // Cấu hình CAN
    CAN_InitStructure.CAN_TTCM = DISABLE;  // 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;  // Receive FIFO Locked Mode
    CAN_InitStructure.CAN_TXFP = ENABLE;   // Transmit FIFO Priority
    CAN_InitStructure.CAN_Mode = CAN_Mode_Normal;  // Chế độ hoạt động bình thường

    // Cấu hình thời gian truyền (bit timing)
    CAN_InitStructure.CAN_SJW = CAN_SJW_1tq;  // Synchronization Jump Width = 1 time quantum
    CAN_InitStructure.CAN_BS1 = CAN_BS1_6tq;  // Bit Segment 1 = 6 time quanta
    CAN_InitStructure.CAN_BS2 = CAN_BS2_8tq;  // Bit Segment 2 = 8 time quanta
    CAN_InitStructure.CAN_Prescaler = 6;  // Tốc độ baudrate = 36 MHz / (Prescaler * 12) = 500 Kbps
    CAN_Init(CAN1, &CAN_InitStructure);
}
```

# 4. Cấu hình chi tiết các tùy chọn CAN trong STM32F103 sử dụng SPL

## 4.1 CAN_Mode (Chế độ hoạt động của CAN)

**CAN_Mode** xác định chế độ hoạt động của bộ điều khiển CAN. Mỗi chế độ phù hợp với các yêu cầu khác nhau trong việc giao tiếp hoặc kiểm tra.

- **CAN_Mode_Normal**: Chế độ hoạt động bình thường. Trong chế độ này, các node CAN sẽ giao tiếp với nhau như bình thường trên mạng CAN. Mỗi node có thể truyền và nhận dữ liệu từ các node khác.
  Ví dụ sử dụng **CAN_Mode_Normal**:

```
CAN_InitStructure.CAN_Mode = CAN_Mode_Normal;
```

- **Ứng dụng**: Sử dụng trong các hệ thống mạng CAN thực tế khi cần giao tiếp bình thường giữa các thiết bị như cảm biến, bộ điều khiển, và actuator.
- **CAN_Mode_LoopBack**: Chế độ vòng lặp, dùng để kiểm tra mà không cần một bus CAN thực. Khi sử dụng chế độ này, tất cả dữ liệu gửi đi sẽ được nhận lại bởi chính node phát ra. Không có giao tiếp thực sự xảy ra với các node khác trên bus.
  Ví dụ sử dụng **CAN_Mode_LoopBack**:

```
CAN_InitStructure.CAN_Mode = CAN_Mode_LoopBack;
```

- **Ứng dụng**: Chế độ này thích hợp cho việc kiểm tra và gỡ lỗi, đảm bảo rằng phần phát và phần nhận của bộ điều khiển CAN hoạt động đúng cách.
- **CAN_Mode_Silent**: Chế độ im lặng, dùng để theo dõi mà không ảnh hưởng đến mạng CAN. Node trong chế độ này chỉ có thể nhận dữ liệu từ các node khác trên bus, nhưng không thể truyền dữ liệu hoặc phát ra bất kỳ tín hiệu nào.
  Ví dụ sử dụng **CAN_Mode_Silent**:

```
CAN_InitStructure.CAN_Mode = CAN_Mode_Silent;
```

- **Ứng dụng**: Chế độ này thường được dùng trong việc nghe lén và giám sát mạng CAN, để đảm bảo không có ảnh hưởng nào từ node đang theo dõi.
- **CAN_Mode_Silent_LoopBack**: Kết hợp cả **LoopBack** và **Silent**. Node hoạt động trong chế độ vòng lặp nhưng không phát ra tín hiệu ra ngoài mạng CAN thực. Chế độ này thích hợp cho việc thử nghiệm mà không ảnh hưởng đến hoạt động của mạng CAN thực.
  Ví dụ sử dụng **CAN_Mode_Silent_LoopBack**:

```
CAN_InitStructure.CAN_Mode = CAN_Mode_Silent_LoopBack;
```

- **Ứng dụng**: Dùng để kiểm tra và mô phỏng mạng CAN mà không làm ảnh hưởng đến các node khác trên mạng thực.

### 4.2 CAN_TTCM (Time Triggered Communication Mode)

**CAN_TTCM** là chế độ giao tiếp theo thời gian, cho phép điều khiển giao tiếp CAN dựa trên các mốc thời gian cố định.

- **ENABLE**: Kích hoạt chế độ giao tiếp theo thời gian, giúp tổ chức các thông điệp truyền theo một lịch định sẵn.
  Ví dụ kích hoạt **TTCM**:

```
CAN_InitStructure.CAN_TTCM = ENABLE;
```

- **Ứng dụng**: Thường được sử dụng trong các hệ thống yêu cầu truyền dữ liệu theo lịch trình, ví dụ như trong các hệ thống điều khiển tự động hóa hoặc trong các hệ thống điều khiển xe hơi.
- **DISABLE**: Tắt chế độ này, CAN hoạt động bình thường mà không dựa trên mốc thời gian.
  Ví dụ tắt **TTCM**:

```
CAN_InitStructure.CAN_TTCM = DISABLE;
```

### 4.3 CAN_ABOM (Automatic Bus-Off Management)

**CAN_ABOM** cho phép node CAN tự động ngắt khỏi bus khi phát hiện quá nhiều lỗi.

- **ENABLE**: Kích hoạt chế độ tự động ngắt kết nối khỏi bus CAN nếu có lỗi **Bus-Off** (khi một node gặp quá nhiều lỗi và bị loại khỏi mạng). Sau khi bị ngắt kết nối, node sẽ tự động khởi động lại sau một khoảng thời gian nhất định và cố gắng tham gia lại mạng.
  Ví dụ kích hoạt **ABOM**:
  **CAN_InitStructure.CAN_ABOM = ENABLE;**
- **Ứng dụng**: Chế độ này thường được dùng trong các hệ thống có độ tin cậy cao, giúp node tự động phục hồi sau khi bị lỗi mà không cần can thiệp thủ công.
- **DISABLE**: Tắt chế độ này, node sẽ không tự động khởi động lại sau khi gặp lỗi **Bus-Off**. Người dùng phải can thiệp bằng phần mềm để khởi động lại node.
  Ví dụ tắt **ABOM**:

```
CAN_InitStructure.CAN_ABOM = DISABLE;
```

### 4.4 CAN_AWUM (Automatic Wake-Up Mode)

**CAN_AWUM** cho phép node tự động "thức dậy" từ chế độ ngủ khi nhận được một thông điệp trên bus CAN.

- **ENABLE**: Kích hoạt chế độ này. Node sẽ tự động thoát khỏi trạng thái ngủ khi có bất kỳ tín hiệu nào trên bus CAN.
  Ví dụ kích hoạt **AWUM**:

```
CAN_InitStructure.CAN_AWUM = ENABLE;
```

- **Ứng dụng**: Dùng trong các hệ thống yêu cầu tiết kiệm năng lượng. Khi không có dữ liệu truyền nhận, node sẽ chuyển sang chế độ **Sleep** để tiết kiệm điện và sẽ tự động thức dậy khi cần.
- **DISABLE**: Tắt chế độ này, node chỉ có thể thoát khỏi chế độ ngủ thông qua can thiệp của phần mềm.
  Ví dụ tắt **AWUM**:

```
CAN_InitStructure.CAN_AWUM = DISABLE;
```

### 4.5 CAN_NART (No Automatic Retransmission)

**CAN_NART** quyết định xem khung dữ liệu có tự động được truyền lại nếu không có tín hiệu **ACK** (Acknowledgment) từ các node khác hay không.

- **ENABLE**: Node sẽ không tự động truyền lại khung dữ liệu nếu không nhận được tín hiệu **ACK** từ các node khác.
  Ví dụ kích hoạt **NART**:
  **CAN_InitStructure.CAN_NART = ENABLE;**
- **Ứng dụng**: Dùng trong các hệ thống có yêu cầu cao về thời gian, nơi mà việc truyền lại khung dữ liệu có thể gây ra độ trễ không mong muốn.
- **DISABLE**: Node sẽ tự động truyền lại khung dữ liệu nếu không nhận được **ACK**.
  Ví dụ tắt **NART**:
  **CAN_InitStructure.CAN_NART = DISABLE;**

### 4.6 CAN_RFLM (Receive FIFO Locked Mode)

**CAN_RFLM** quyết định xem FIFO sẽ bị khóa hay ghi đè nếu dữ liệu không được đọc kịp thời.

- **ENABLE**: Khi FIFO đã đầy, không có khung dữ liệu mới nào được nhận vào cho đến khi dữ liệu cũ được đọc.
  Ví dụ kích hoạt **RFLM**:

```
CAN_InitStructure.CAN_RFLM = ENABLE;
```

- **Ứng dụng**: Dùng trong các hệ thống yêu cầu kiểm soát chặt chẽ việc xử lý dữ liệu và không được bỏ qua bất kỳ dữ liệu nào.
- **DISABLE**: Dữ liệu mới sẽ ghi đè lên dữ liệu cũ nếu FIFO đầy.
  Ví dụ tắt **RFLM**:

```
CAN_InitStructure.CAN_RFLM = DISABLE;
```

### 4.7 CAN_TXFP (Transmit FIFO Priority)

**CAN_TXFP** quyết định độ ưu tiên của các khung dữ liệu trong FIFO.

- **ENABLE**: Khung dữ liệu có độ ưu tiên cao hơn sẽ được truyền trước, không tuân theo thứ tự nhập vào FIFO.
  Ví dụ kích hoạt **TXFP**:

```
CAN_InitStructure.CAN_TXFP = ENABLE;
```

- **Ứng dụng**: Dùng trong các hệ thống mà một số khung dữ liệu cần được ưu tiên truyền đi trước, ngay cả khi được thêm vào sau.
- **DISABLE**: Khung dữ liệu sẽ được truyền theo thứ tự **FIFO** (First-In-First-Out), tức là khung nào vào trước sẽ được truyền trước.
  Ví dụ tắt **TXFP**:

```
CAN_InitStructure.CAN_TXFP = DISABLE;
```

## 5. Gửi và nhận dữ liệu CAN

### 5.1 Gửi dữ liệu CAN:

Sử dụng hàm CAN_Transmit để gửi một **Data Frame** với tối đa 8 byte dữ liệu.

Ví dụ gửi một khung dữ liệu với 8 byte:

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

    TxMessage.StdId = 0x123;  // Sử dụng định danh 11-bit
    TxMessage.RTR = CAN_RTR_DATA;  // Data Frame
    TxMessage.IDE = CAN_ID_STD;  // Standard ID
    TxMessage.DLC = length;  // Số byte dữ liệu

    // Copy dữ liệu vào trường dữ liệu của khung
    for (int i = 0; i < length; i++) {
        TxMessage.Data[i] = data[i];
    }

    CAN_Transmit(CAN1, &TxMessage);  // Truyền khung dữ
```

```
liệu

    // Chờ cho đến khi khung được truyền thành công
    while (CAN_TransmitStatus(CAN1,
CAN_TransmitStatus_Complete) != CAN_TxStatus_Ok);
}
```

### 5.2 Nhận dữ liệu CAN:

Sử dụng hàm CAN_Receive để nhận một **Data Frame** từ FIFO buffer.

Ví dụ nhận một khung dữ liệu:

```
void CAN_ReceiveData(uint8_t* data, uint8_t* length) {
    CanRxMsg RxMessage;

    CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);  // Nhận
từ FIFO 0

    // Copy dữ liệu nhận được
    *length = RxMessage.DLC;  // Độ dài dữ liệu
    for (int i = 0; i < RxMessage.DLC; i++) {
        data[i] = RxMessage.Data[i];
    }
}
```

## 6. Cấu hình bộ lọc CAN

Bộ lọc CAN cho phép node chỉ nhận những khung dữ liệu có định danh phù hợp với yêu cầu, giúp giảm thiểu gánh nặng xử lý khi có nhiều khung dữ liệu khác nhau trên bus CAN.

### 6.1 Cấu hình bộ lọc:

**CAN_FilterInitTypeDef** được sử dụng để cấu hình bộ lọc. Có thể thiết lập các bộ lọc với chế độ lọc theo **ID Mask** hoặc **ID List**, với khả năng lọc theo định danh **11-bit** (Standard ID) hoặc **29-bit** (Extended ID).

- **ID Mask**: Node chỉ nhận khung dữ liệu khi định danh khung và định danh của bộ lọc khớp với nhau dựa trên mặt nạ.
- **ID List**: Node chỉ nhận các khung có định danh khớp hoàn toàn với một trong các giá trị trong danh sách định danh của bộ lọc.

Ví dụ cấu hình bộ lọc với **ID Mask**:

```
void CAN_FilterConfiguration(void) {
    CAN_FilterInitTypeDef CAN_FilterInitStructure;

    // Sử dụng bộ lọc số 0, chế độ ID Mask, định danh
11-bit
    CAN_FilterInitStructure.CAN_FilterNumber = 0;
    CAN_FilterInitStructure.CAN_FilterMode =
CAN_FilterMode_IdMask;
    CAN_FilterInitStructure.CAN_FilterScale =
CAN_FilterScale_32bit;
    CAN_FilterInitStructure.CAN_FilterIdHigh = 0x321 <<
5;  // Định danh Standard ID 0x321
    CAN_FilterInitStructure.CAN_FilterIdLow = 0x0000;
    CAN_FilterInitStructure.CAN_FilterMaskIdHigh =
0xFFE0;  // Mặt nạ chỉ so sánh 11-bit ID
    CAN_FilterInitStructure.CAN_FilterMaskIdLow = 0x0000;
    CAN_FilterInitStructure.CAN_FilterFIFOAssignment =
CAN_FIFO0;  // Đưa khung vào FIFO0
    CAN_FilterInitStructure.CAN_FilterActivation =
ENABLE;

    // Khởi tạo bộ lọc
    CAN_FilterInit(&CAN_FilterInitStructure);
}
```

### 7. Sử dụng ngắt trong CAN

Ngắt giúp phản hồi tức thời khi có các sự kiện như nhận dữ liệu mới, hoàn thành việc truyền, hoặc gặp lỗi trong quá trình giao tiếp CAN. STM32F103 hỗ trợ nhiều loại ngắt CAN, bao gồm:

- Ngắt khi có dữ liệu mới trong **FIFO**.
- Ngắt khi **FIFO** đầy.
- Ngắt khi một khung dữ liệu được truyền thành công.
- Ngắt khi có lỗi trong quá trình truyền hoặc nhận dữ liệu.

**7.1 Kích hoạt ngắt:**

Ví dụ dưới đây cấu hình ngắt CAN khi có dữ liệu mới vào FIFO0:

```
void CAN_Interrupt_Configuration(void) {
    NVIC_InitTypeDef NVIC_InitStructure;

    // Kích hoạt ngắt CAN 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);

    // Kích hoạt ngắt nhận dữ liệu mới vào FIFO0
    CAN_ITConfig(CAN1, CAN_IT_FMP0, ENABLE);
}
```

**7.2 Hàm xử lý ngắt:**

Khi có dữ liệu mới vào **FIFO0**, hàm xử lý ngắt sẽ đọc khung dữ liệu này.

```
void USB_LP_CAN1_RX0_IRQHandler(void) {
    if (CAN_GetITStatus(CAN1, CAN_IT_FMP0) != RESET) {
        CanRxMsg RxMessage;
```

```
        // Nhận dữ liệu từ FIFO0
        CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);

        // Xử lý dữ liệu nhận được...
        // Ví dụ: Lưu trữ dữ liệu hoặc phản hồi lại
khung dữ liệu này.

        // Xóa cờ ngắt
        CAN_ClearITPendingBit(CAN1, CAN_IT_FMP0);
    }
}
```

### 8. Ví dụ thực tế

Dưới đây là ví dụ thực tế về việc cấu hình và sử dụng CAN trên **STM32F103** sử dụng **SPL**:

**Bước 1: Cấu hình CAN và bộ lọc**

```
int main(void) {
    // Khởi tạo Clock và GPIO cho CAN
    CAN_Clock_Configuration();
    CAN_GPIO_Configuration();

    // Cấu hình CAN
    CAN_Configuration();

    // Cấu hình bộ lọc CAN
    CAN_FilterConfiguration();

    // Cấu hình ngắt cho CAN
    CAN_Interrupt_Configuration();

    while (1) {
        // Chương trình chính có thể thực hiện các tác
```

```
vụ khác
        // Việc nhận và xử lý dữ liệu sẽ do ngắt CAN xử
lý
    }
}
```

**Bước 2: Xử lý dữ liệu trong ngắt**

```
void USB_LP_CAN1_RX0_IRQHandler(void) {
    if (CAN_GetITStatus(CAN1, CAN_IT_FMP0) != RESET) {
        CanRxMsg RxMessage;

        // Nhận dữ liệu từ FIFO0
        CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);

        // Ví dụ xử lý dữ liệu: hiển thị dữ liệu trên
terminal qua UART
        printf("Received CAN Data: ");
        for (int i = 0; i < RxMessage.DLC; i++) {
            printf("%02X ", RxMessage.Data[i]);
        }
        printf("\n");

        // Xóa cờ ngắt
        CAN_ClearITPendingBit(CAN1, CAN_IT_FMP0);
    }
}
```