第二十章 CAN 总线协议

本章我们将向大家介绍如何利用 STM32F103C8T6 实现 CAN 通信实验。本章分为一下部分：

20.1 CAN 简介

20.2 CAN 数据报文帧结构

20.3 总线仲裁

20.4 文填充

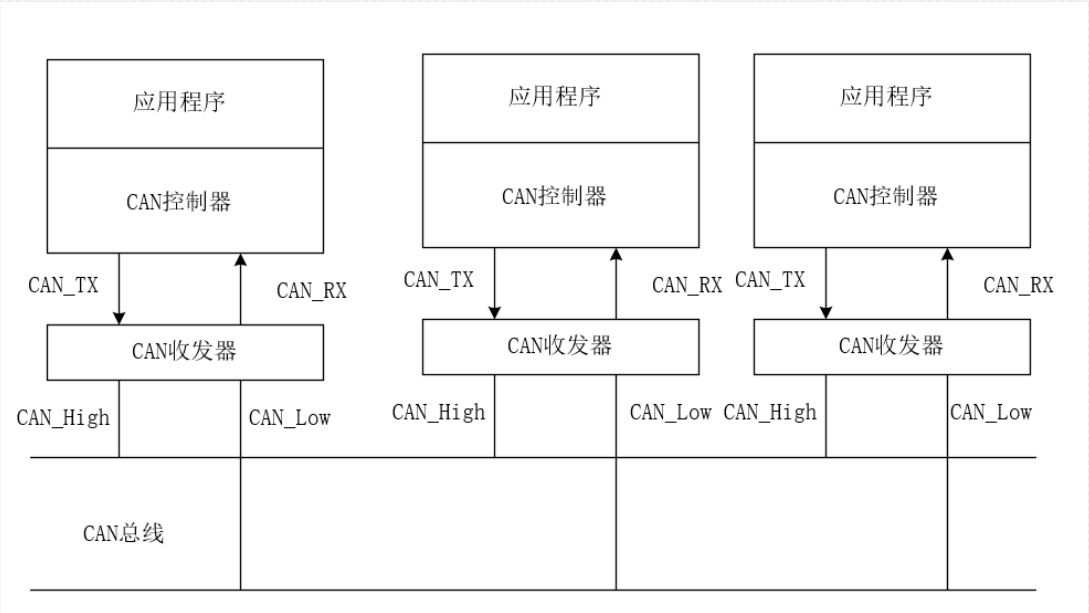
20.5 CAN 错误帧，过载帧结构以及帧间隔

20.6 CAN 总线错误类型

20.7 CAN 的代码示例与操作演示

20.1 CAN 简介

CAN 是控制器域网 (Controller Area Network, CAN) 的简称。CAN 属于总线式串行通信网络。



1. CAN 节点通过 CAN_High、CAN_Low 两根线接入 CAN 总线网络。
2. CAN 收发器将 CAN 控制器的 CAN_TX 输出的逻辑值 0、1 转换为差分信号
3. 通过 CAN_High、CAN_Low 向总线输出；同时将总线上的差分信号转换为逻辑值 0、1。
4. 通过 CAN_RX 传输给 CAN 控制器。

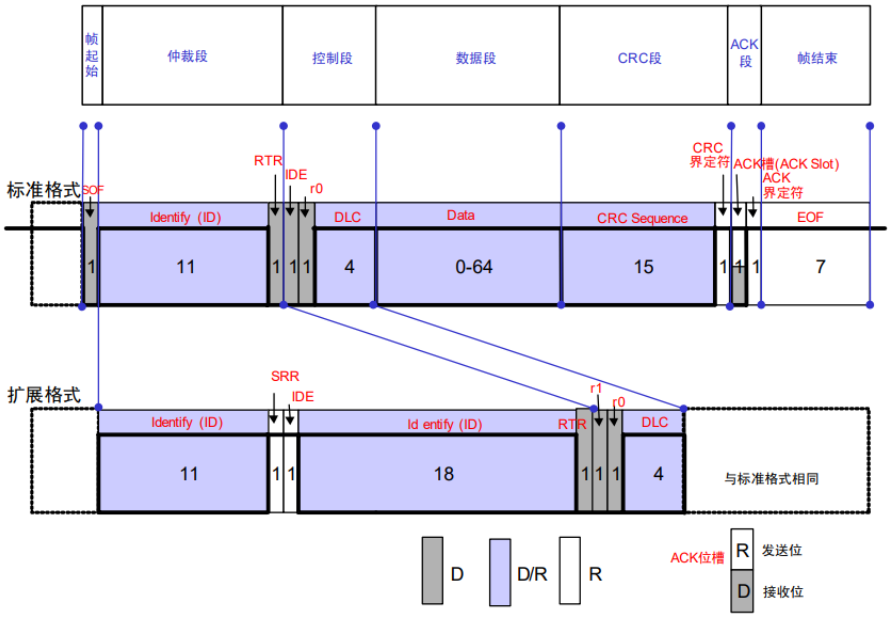
CAN_High、CAN_Low 的差分电压与逻辑值 0、1 的关系：

逻辑值	CAN_High	CAN_Low	差分电压（CAN_High-CAN_Low)
0	3.5V	1.5V	2V
1	2.5V	2.5V	0V

20.2 CAN 数据报文帧结构

CAN 数据报文分**标准和扩展**两种类型。

标准数据报文结构：帧起始(SOF)、仲裁段、控制段、数据段、CRC 段、ACK 段、帧结束(EOF)



- **帧起始**占 1 位，值为 0，表示一帧数据报文的开始，帧结束占 7 位，值都为 1，表示一帧数据报文的结束。
- **仲裁段**包括 11 位的标识符 ID 和 1 位 RTR
  - 每个节点发送的数据报文使用独立的标识符 ID，接收方可以通过标识符 ID 判断数据报文来自哪个节点；
  - RTR 指示该数据报文是数据帧还是遥控帧，RTR 值为 0 时表示数据帧，值为 1 时表示遥控帧，遥控帧不含数据段，节点可以通过发送遥控帧来请求其它节点发送数据。
  - 当多个节点同时发送数据报文时，节点通过仲裁段来参与总线的竞争，竞争获胜的一方可继续发送数据报文。
- **控制段**包括 1 位 IDE、1 位 r0、4 位的 DLC。
  - IDE 指示该数据报文属于**标准数据报文还是扩展数据报文**，IDE 值为 **0 时表示标准数据报文**，值为 **1 时表示扩展数据报文**。DLC 指示数据段的字节数（最大值应为 8），r0 值为 0。

- 实际要传输的数据放在**数据段**，占 0 ~ 8 个字节，即一帧数据报文最多只能携带 8 个字节的数据内容。数据段的字节数通过 DLC 指定。
- **CRC 段**包含 **15 位的 CRC 和 1 位 CRC 界定符**。
  - CRC 的**计算范围包括帧起始、仲裁段、控制端、数据段**，接收节点在接收数据报文时**也会计算 CRC**，与数据报文中的 CRC 进行比较，**如果不一致会报 CRC 错误。CRC 界定符的值为 1**。
- **ACK 段**包括 1 位 ACK 槽和 1 位 ACK 界定符。
  - **发送节点在 ACK 槽输出 1**，接收节点**如果正确接收到数据报文，会在 ACK 槽输出 0 作为应答**。ACK 界定符的值为 1。
- **帧结束**是表示该该帧的结束的段。由 7 个位的隐性位构成。

20.3 总线仲裁

- 当多个节点同时发送数据报文时，节点通过仲裁段来参与总线的竞争。
- **节点向总线发送数据的同时会检测总线的电平状态。**

20.4 位填充

- 节点发送数据报文时，在数据报文的 SOF 至 CRC 段内，如果出现连续 5 个位的 0，要在下一个位插入 1， 如果出现连续 5 个位的 1，则要在下一个位插入 0 。
- 节点接收数据报文时，要去掉填充位，即在数据报文的 SOF～CRC 段内，如果发现连续 5 个位的 0 或连续 5 个位的 1，要删除下一个位，即去掉填充位。

20.5 CAN 错误帧，过载帧结构以及帧间隔

- 在数据报文传输过程中，节点检测出总线错误后会发送错误帧。
- 错误帧由错误标志和错误界定符构成。
- 错误标志分主动错误标志（连续 6 个位的 0）和被动错误标志（连续 6 个位的 1）。
- 处于主动错误状态的节点检测出错误时会发送带主动错误标志的错误帧，处于被动错误状态的节点在检测出错误时会发送带被动错误标志的错误帧。

- 错误界定符是连续 8 个位的 1。节点发送完错误标志后，从检测到总线的第一个 1 开始的连续 8 个位的 1 作为错误帧的界定符。
- 发送数据报文的节点检测出错误
  - 该节点停止数据报文的发送，转为发送错误帧。错误帧的错误标志会破坏原数据报文的结构，其它节点可能在错误标志的任何一位检测出错误，然后在下一位开始发送错误帧，因此错误标志会出现重叠部分。
- 接收数据报文的节点检测出错误
  - 如果该节点处于主动错误状态，会发送带主动错误标志的错误帧。**主动错误标志会破坏总线上正在传输的数据报文的结构，其它节点可能在主动错误标志的任何一位检测出错误，然后在下一位开始发送错误帧，因此错误标志会出现重叠部分。**
  - 如果该节点处于被动错误状态，会发送带被动错误标志的错误帧。**被动错误标志对总线上正在传输的数据报文没有影响。**
- 帧间隔用于将数据帧或遥控帧与前面的帧（可以是数据帧、遥控帧、错误帧、过载帧）分离开来，帧间隔是连续 3 个位的 1。处于被动错误状态的节点，在输出帧间隔时需要在帧间隔之后插入“延迟传送”（连续 8 个位的 1）。

20.6 CAN 总线错误类型

位错误、ACK 错误、填充错误、CRC 错误、格式错误。

1. 位错误
   - 节点发送数据报文，输出 0 时检测到总线为 1，或输出 1 时检测到总线为 0，该节点检测出一个位错误。但以下两种情况除外：
     - 在数据报文的仲裁段，节点输出 1 时检测到总线为 0，说明该节点竞争总线失败，不视为位错误；
     - 在数据报文的 ACK 槽，发送节点输出 1 时检测到总线为 0，说明有接收节点正确接收到数据报文并输出 0 来做出应答，不视为位错误。
2. ACK 错误：
   - 节点发送数据报文时，在 ACK 槽检测到总线为 1（即没有接收节点正确接收到数据报文并输出 0 来做出应答），该节点检测出一个位错误。

3. 填充错误:

   – 节点发送数据报文，在数据报文的 SOF ~ CRC 段内检测到连续 6 个位的 0 或连续 6 个位的 1，该节点检测出一个填充错误。

   – 节点接收数据报文，在数据报文的 SOF ~ CRC 段内检测到连续 6 个位的 0 或连续 6 个位的 1，该节点检测出一个填充错误。

4. CRC 错误

   – 节点接收数据报文，检测到计算的 CRC 与数据报文中的 CRC 不一致，该接收节点检测出一个 CRC 错误。

   – 节点发送错误帧或过载帧，在错误界定符或过载界定符中检测到 0，该节点检测出一个格式错误。

20.7 CAN 的代码示例与操作演示

编写代码流程：

20.7.1 使能 CAN 时钟

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);    //使能 PORT A 时钟
RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);     //使能 CAN1 时钟
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11; //PA11
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;              //上拉输入模式
GPIO_Init(GPIOA, &GPIO_InitStructure);
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12; //PA12
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //复用推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;//IO 口速度为 50MHz
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

20.7.2 设置 CAN 工作模式、波特率等

```
// CAN 的初始化函数

uint8_t CAN_Init(CAN_TypeDef* CANx, CAN_InitTypeDef*CAN_InitStruct);

typedef struct
{
uint16_t CAN_Prescaler;
uint8_t CAN_Mode;
uint8_t CAN_SJW;
uint8_t CAN_BS1;
uint8_t CAN_BS2;
```

```
FunctionalState CAN_TTCM;
FunctionalState CAN_ABOM;
FunctionalState CAN_AWUM;
FunctionalState CAN_NART;
FunctionalState CAN_RFLM;
FunctionalState CAN_TXFP;
} CAN_InitTypeDef;
```

CAN_InitTypeDef 结构体介绍：

- CAN_Prescaler：用于设置 CAN 外设的时钟分频，它可控制时间片 tq 的时间长度，这里设置的值最终会加 1 后再写入 BRP 寄存器位。
- CAN_Mode：用于设置 CAN 的工作模式，可设置为正常模式(CAN_Mode_Normal)、回环模式(CAN_Mode_LoopBack)、静默模式(CAN_Mode_Silent)以及回环静默模式(CAN_Mode_Silent_LoopBack)。

本实验使用到的只有正常模式和回环模式。

- CAN_SJW：用于置 SJW 的极限长度，即 CAN 重新同步时单次可增加或缩短的最大长度，它可以被配置为 1-4tq(CAN_SJW_1/2/3/4tq)。
- CAN_BS1：用于设置 CAN 位时序中的 BS1 段的长度，它可以被配置为 1-16 个 tq 长度(CAN_BS1_1/2/3…16tq)。
- CAN_BS2：用于设置 CAN 位时序中的 BS2 段的长度，它可以被配置为 1-8 个 tq 长度(CAN_BS2_1/2/3…8tq)。
- CAN_TTCM：用于设置是否使用时间触发功能，ENABLE 为使能，DISABLE 为失能。时间触发功能在某些 CAN 标准中会使用到。
- CAN_ABOM：用于设置是否使用自动离线管理(ENABLE/DISABLE)，使用自动离线管理可以在节点出错离线后适时自动恢复，不需要软件干预。
- CAN_AWUM：用于设置是否使用自动唤醒功能(ENABLE/DISABLE)，使能自动唤醒功能后它会在监测到总线活动后自动唤醒。
- CAN_NART：用于设置是否使用自动重传功能(ENABLE/DISABLE)，使用自动重传功能时，会一直发送报文直到成功为止。
- CAN_RFLM：用于设置是否使用锁定接收 FIFO(ENABLE/DISABLE)，锁定接收 FIFO 后，若 FIFO 溢出时会丢弃新数据，否则在 FIFO 溢出时以新数据覆盖旧数据。
- CAN_TXFP：用于设置发送报文的优先级判定方法(ENABLE/DISABLE)，使能时，以报文存入发送邮箱的先后顺序来发送，否则按照报文 ID 的优先级来发送。

$$CAN\,波特率 = \frac{\text{Fpclk1}}{(\text{TBS1} + \text{TBS2} + 1) * \text{BP1}}$$

就可以算出波特率。本章实验我们初始化配置 CAN 为正常工作模式，波特率为 500Kbps，配置代码如下：

```
CAN_InitTypeDef CAN_InitStructure;
//CAN 单元设置
CAN_InitStructure.CAN_TTCM=DISABLE; //非时间触发通信模式
CAN_InitStructure.CAN_ABOM=DISABLE; //软件自动离线管理
CAN_InitStructure.CAN_AWUM=DISABLE;//睡眠模式通过软件唤醒(清除 CAN->MCR 的 SLEEP 位)
CAN_InitStructure.CAN_NART=ENABLE; //使用报文自动传送
CAN_InitStructure.CAN_RFLM=DISABLE; //报文不锁定,新的覆盖旧的
CAN_InitStructure.CAN_TXFP=DISABLE; //优先级由报文标识符决定

CAN_InitStructure.CAN_Mode= CAN_Mode_Normal; //模式设置
CAN_InitStructure.CAN_SJW=CAN_SJW_1tq; //重新同步跳跃宽度(Tsjw)为 tsjw+1 个时间单位 CAN_SJW_1tq~CAN_SJW_4tq
CAN_InitStructure.CAN_BS1=CAN_BS1_7tq; //Tbs1 范 围 CAN_BS1_1tq~CAN_BS1_16tq
CAN_InitStructure.CAN_BS2=CAN_BS2_6tq;//Tbs2 范 围 CAN_BS2_1tq ~CAN_BS2_8tq
CAN_InitStructure.CAN_Prescaler=6; //分频系数(Fdiv)为 brp+1

CAN_Init(CAN1, &CAN_InitStructure); // 初始化 CAN1
```

20.7.3 设置 CAN 筛选器

设置好 CAN 的工作模式及波特率后，我们还需要通过 CAN_FMR 寄存器设置 CAN 筛选器，库函数中提供了 CAN_FilterInit()函数来完成这一步骤。

函数原型是：

```
void CAN_FilterInit(CAN_FilterInitTypeDef* CAN_FilterInitStruct);

typedef struct
{
  uint16_t CAN_FilterIdHigh;
  uint16_t CAN_FilterIdLow;
  uint16_t CAN_FilterMaskIdHigh;
  uint16_t CAN_FilterMaskIdLow;
  uint16_t CAN_FilterFIFOAssignment;
  uint8_t CAN_FilterNumber;
  uint8_t CAN_FilterMode;
  uint8_t CAN_FilterScale;
```

```
  FunctionalState CAN_FilterActivation;
} CAN_FilterInitTypeDef;
```

- CAN_FilterIdHigh：用于存储要筛选的 ID，若筛选器工作在 32 位模式，它存储的是所筛选 ID 的高 16 位；若筛选器工作在 16 位模式，它存储的就是一个完整的要筛选的 ID。
- CAN_FilterIdLow：同上一个成员一样，它也是用于存储要筛选的 ID，若筛选器工作在 32 位模式， 它存储的是所筛选 ID 的低 16 位；若筛选器工作在 16 位模式，它存储的就是一个完整的要筛选的 ID。
- CAN_FilterMaskIdHigh：用于存储要筛选的 ID 或掩码。
- CAN_FilterMaskIdHigh 存储的内容分两种情况，当筛选器工作在标识符列表模式时，它的功能与 CAN_FilterIdHigh 相同，都是存储要筛选的 ID；而当筛选器工作在掩码模式时，它存储的是 CAN_FilterIdHigh 成员对应的掩码，与 CAN_FilterIdLow 组成一组筛选器。
- CAN_FilterMaskIdLow：同上一个成员一样，它也是用于存储要筛选的 ID 或掩码，只不过这里对应存储 CAN_FilterIdLow 的成员。
- CAN_FilterFIFOAssignment：用于设置当报文通过筛选器的匹配后，该报文会被存储到哪一个接收 FIFO，它的可选值为 FIFO0 FIFO1(CAN_Filter_FIFO0/1)。
- CAN_FilterNumber：用于设置筛选器的编号， 即使用的是哪个筛选器。CAN 一共有 28 个筛选器，所以它的可输入参数范围为 0-27。
- CAN_FilterMode：用于设置筛选器的工作模式，可以设置为列表模式(CAN_FilterMode_IdList)及掩码模式(CAN_FilterMode_IdMask)。
- CAN_FilterScale：用于设置筛选器的位宽，可以设置为 32 位(CAN_FilterScale_32bit)及 16 位长(CAN_FilterScale_16bit)。
- CAN_FilterActivation：用于设置是否激活这个筛选器(ENABLE/DISABLE)。

```
CAN_FilterInitTypeDef CAN_FilterInitStructure;//配置过滤器
CAN_FilterInitStructure.CAN_FilterNumber=0; //过滤器 0
CAN_FilterInitStructure.CAN_FilterMode=CAN_FilterMode_IdMask;
CAN_FilterInitStructure.CAN_FilterScale=CAN_FilterScale_32bit;//32 位
CAN_FilterInitStructure.CAN_FilterIdHigh=0x0000;//32 位 ID
CAN_FilterInitStructure.CAN_FilterIdLow=0x0000;
CAN_FilterInitStructure.CAN_FilterMaskIdHigh=0x0000;//32 位 MASK
CAN_FilterInitStructure.CAN_FilterMaskIdLow=0x0000;
CAN_FilterInitStructure.CAN_FilterFIFOAssignment=CAN_Filter_FIFO;//过滤器 0 关联到 FIFO0
```

```
CAN_FilterInitStructure.CAN_FilterActivation=ENABLE;//激活过滤器 0
CAN_FilterInit(&CAN_FilterInitStructure); //滤波器初始化
```

20.7.4 选择 CAN 中断类型，开启中断

配置好上述 3 个步骤后，CAN 就可以开始工作了，如果要开启 CAN 的接收中断， 我们还需要选择它的中断类型并使能。配置 CAN 中断类型及使能的库函数是：

```
void CAN_ITConfig(CAN_TypeDef* CANx, uint32_t CAN_IT,FunctionalState NewSta
te);
```

比如我们使用 FIFO0 消息挂号允许中断，那么此参数可以设置为 CAN_IT_FMP0。使能中断后还需要设置它的中断优先级，即初始化 NVIC。当产生中断后就会进去 CAN 接收中断内进行处理， 所以需要编写一个 CAN 接收中断函数，如 USB_LP_CAN1_RX0_IRQHandler。

示例代码：

```
void CAN_ITConfig_Init(void)
{
	CAN_ITConfig(CAN1,CAN_IT_FMP0,ENABLE);// FIFO0 消息挂号中断允许.
	CAN_ITConfig(CAN1,CAN_IT_TME,ENABLE);//发送邮箱空中断允许
	CAN_ITConfig(CAN1,CAN_IT_BOF,ENABLE);//错误打断，Bus-off 中断允
许

	NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);//配置优先级分组:抢占优
先级和子优先级。

	NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN1_RX0_IRQn;
	NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2;
	NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
	NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;

	NVIC_Init(&NVIC_InitStructure);//初始化 NVIC
}
```

再编写中断函数：

```
void USB_LP_CAN1_RX0_IRQHandler(void)
{
		int i=0;
		CanRxMsg RxMessage;
		RxMessage.StdId=0x00;
		RxMessage.ExtId=0x00;
		RxMessage.IDE=0;
		RxMessage.RTR=0;
```

```
                RxMessage.FMI=0;
                for(i=0;i<8;i++){
                        RxMessage.Data[i]=0;
                }
                CAN_Receive(CAN1, 0, &RxMessage);//can 接收
                printf("id=%d \r\n",RxMessage.StdId);
                printf("data[0]=%d \r\n",RxMessage.Data[0]);
                printf("data[1]=%d \r\n",RxMessage.Data[1]);
                printf("data[2]=%d \r\n",RxMessage.Data[2]);
                printf("data[3]=%d \r\n",RxMessage.Data[3]);
                printf("data[4]=%d \r\n",RxMessage.Data[4]);
                printf("data[5]=%d \r\n",RxMessage.Data[5]);
                printf("data[6]=%d \r\n",RxMessage.Data[6]);
                printf("data[7]=%d \r\n",RxMessage.Data[7]);
}
```

20.7.5 CAN 发送和接收消息

发送消息的函数是：

```
uint8_t CAN_Transmit(CAN_TypeDef* CANx, CanTxMsg* TxMessage);

typedef struct
{
  uint32_t StdId;
  uint32_t ExtId;
  uint8_t IDE;
  uint8_t RTR;
  uint8_t DLC;
  uint8_t Data[8];
} CanTxMsg;
```

- StdId：用于存储报文的 11 位标准标识符，范围是 0-0x7FF。
- ExtId：用于存储报文的 29 位扩展标识符， 范围是 0-0x1FFFFFFF。ExtId 与 StdId 这两个成员哪一个有效要根据下面的 IDE 位配置。
- IDE：用于存储扩展标志 IDE 位的值，其值可配置为 CAN_ID_STD 和
- CAN_ID_EXT。如果为 CAN_ID_STD 时表示本报文是标准帧，使用 StdId 成员存储报文 ID。如果为 CAN_ID_EXT 时表示本报文是扩展帧，使用 ExtId 成员存储报文 ID。
- RTR：用于存储报文类型标志 RTR 位的值，当它的值为宏 CAN_RTR_Data 时表示本报文是数据帧；当它的值为宏 CAN_RTR_Remote 时表示本报文是

遥控帧，由于遥控帧没有数据段，所以当报文是遥控帧时，下面的 Data[8] 成员的内容是无效的。

- DLC：用于存储数据帧数据段的长度，其值范围是 0-8，当报文是遥控帧时 DLC 值为 0。
- Data[8]：用于存储数据帧中数据段的数据。

当我们需要发送报文时，就需要对此结构体进行初始化，然后调用该函数发送出去，例如：

```
CanTxMsg TxMessage;
TxMessage.StdId=0x12; // 标准标识符为 0
TxMessage.ExtId=0x12; // 设置扩展标示符（29 位）
TxMessage.IDE=0; // 使用扩展标识符
TxMessage.RTR=0; // 消息类型为数据帧，一帧 8 位
TxMessage.DLC=8; // 发送两帧信息
for(i=0;i<8;i++)
TxMessage.Data[8]=msg[i]; // 第一帧信息
CAN_Transmit(CAN1, &TxMessage);
```

接收消息的函数是：

```
void CAN_Receive(CAN_TypeDef* CANx, uint8_t FIFONumber, CanRxMsg*RxMess
age);

typedef struct
{
  uint32_t StdId;
  uint32_t ExtId;
  uint8_t IDE;
  uint8_t RTR;
  uint8_t DLC;
  uint8_t Data[8];
  uint8_t FMI;
} CanRxMsg;
```

前面几个成员和 CanTxMsg 结构体内是一样的。

- FMI，它用于存储筛选器的编号，表示本报文是经过哪个筛选器存储进接收 FIFO 的，可以用它简化软件处理。

20.7.6 CAN 状态获取

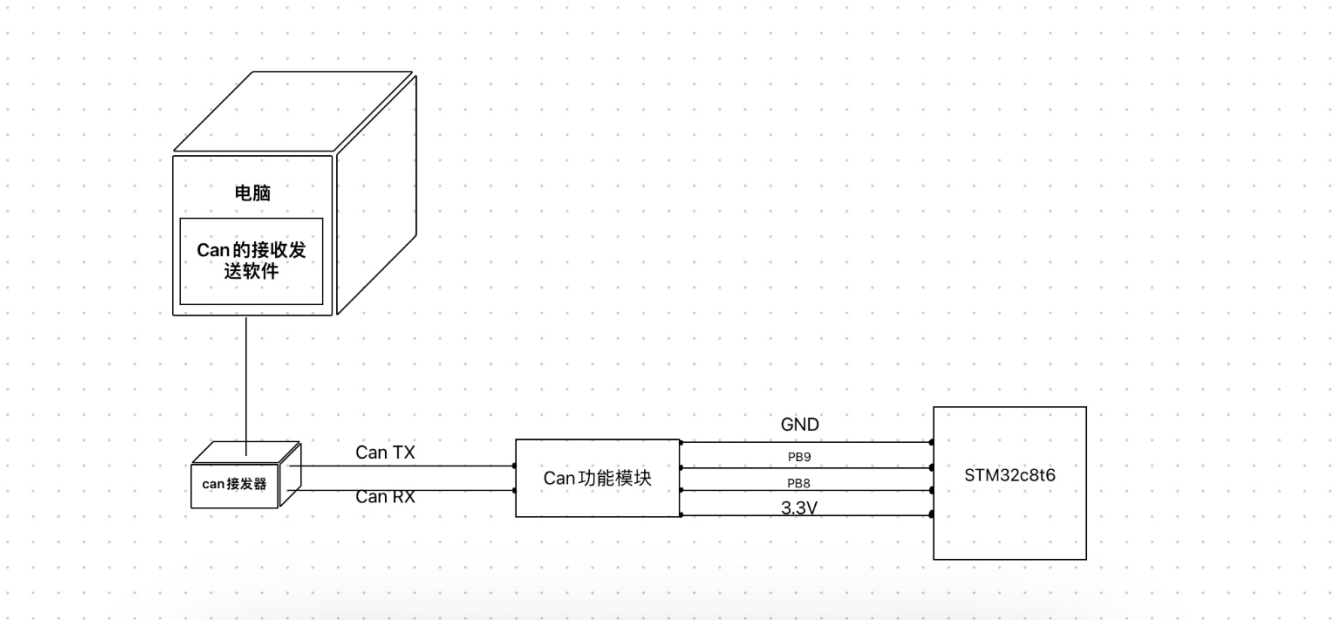
当使用 CAN 进行数据传输时， 我们会通过获取 CAN 状态标志来确认是否传输完成，比如说在接收报文时，通过检测标志位获知接收 FIFO 的状态，若收到报

文，可调用库函数 CAN_Receive 把接收 FIFO 中的内容读取到预先定义的接收类型结构体中，然后再访问该结构体即可利用报文了。

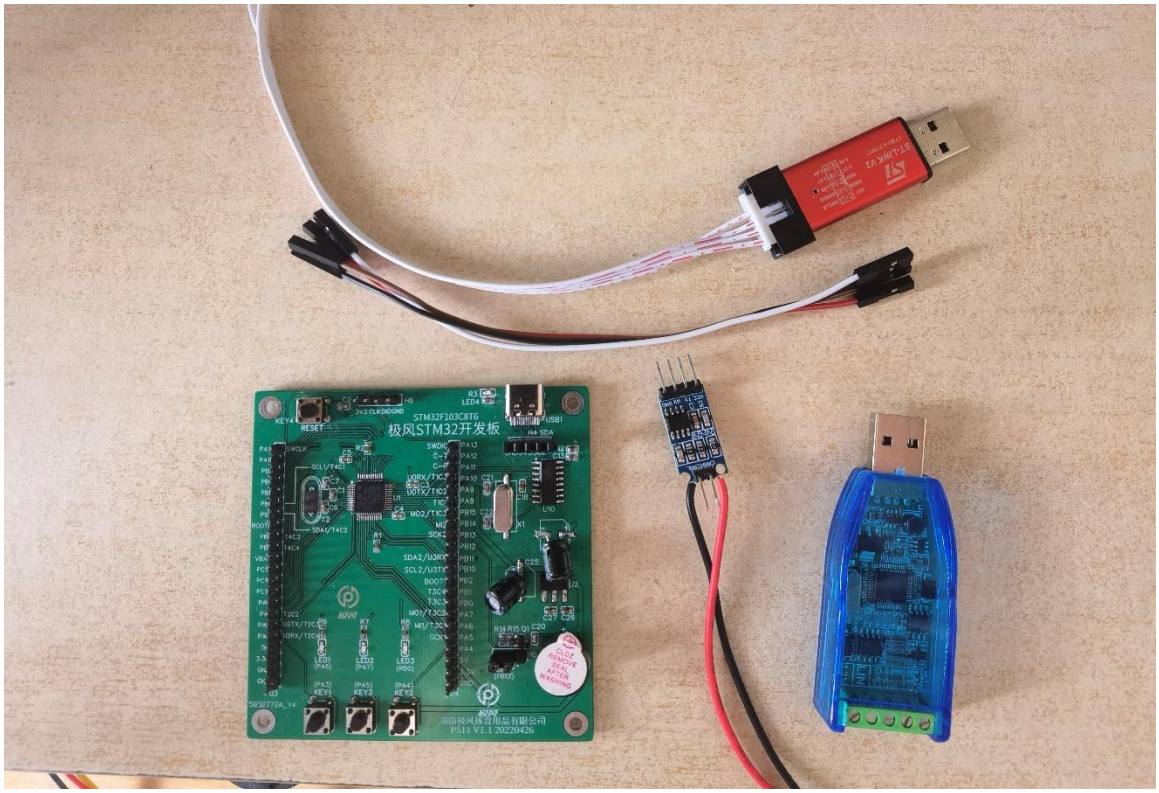
库函数中提供了很多获取 CAN 状态标志的函数，如 CAN_TransmitStatus()函数， CAN_MessagePending()函数， CAN_GetFlagStatus()函数等等，大家可以根据需要来调用。

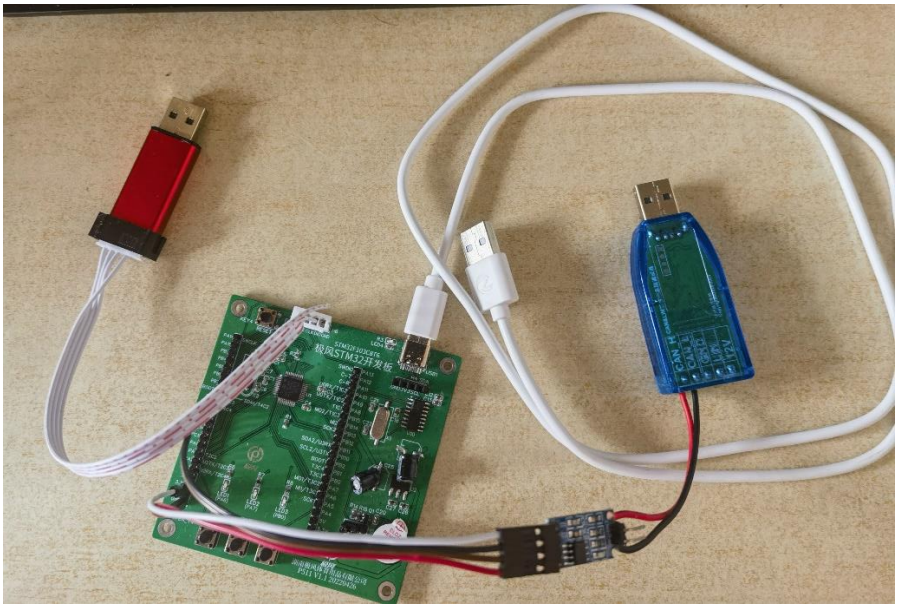
20.7.8 CAN 实验演示

CAN 示例代码的连接模型图：



主要使用设备及元件：电脑，极风 STM32F103C8T6 开发板，TJA1050 CAN 模块，CAN 总线调试器，ST-LINK 调试器，杜邦线若干。


极风STM32开发板
STM32F103C8T6


极风STM32开发板

示例代码实验现象：

1. 当电脑发送 0X 时，控制 LEDX 亮起 300ms，后返回亮起的 LED 序号 X。

2. 当按下 KEYX 时，点亮对应的 LED，并返回 KEY 的序号 X。