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

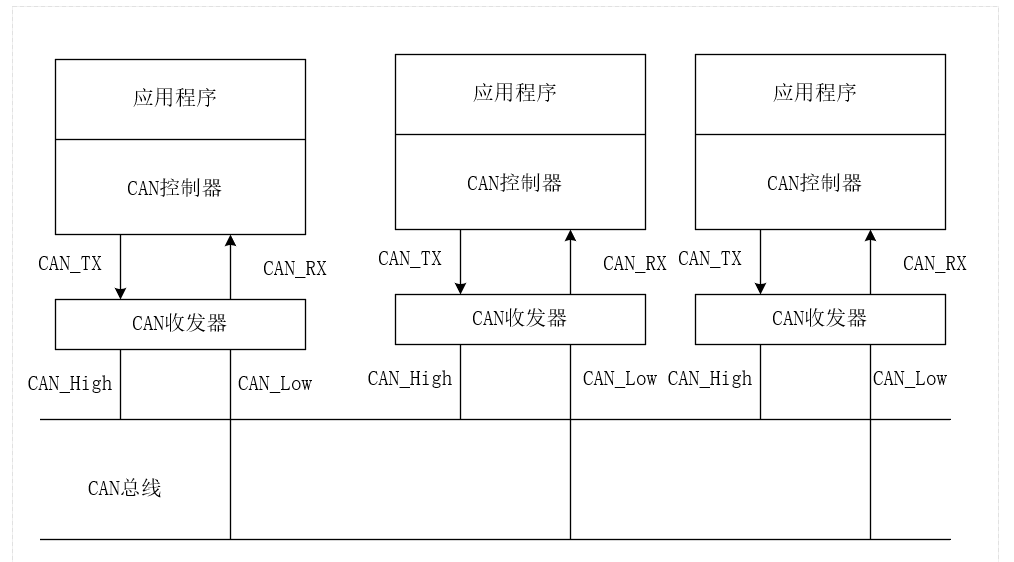
20.5 CAN错误帧，过载帧结构以及帧间隔

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。

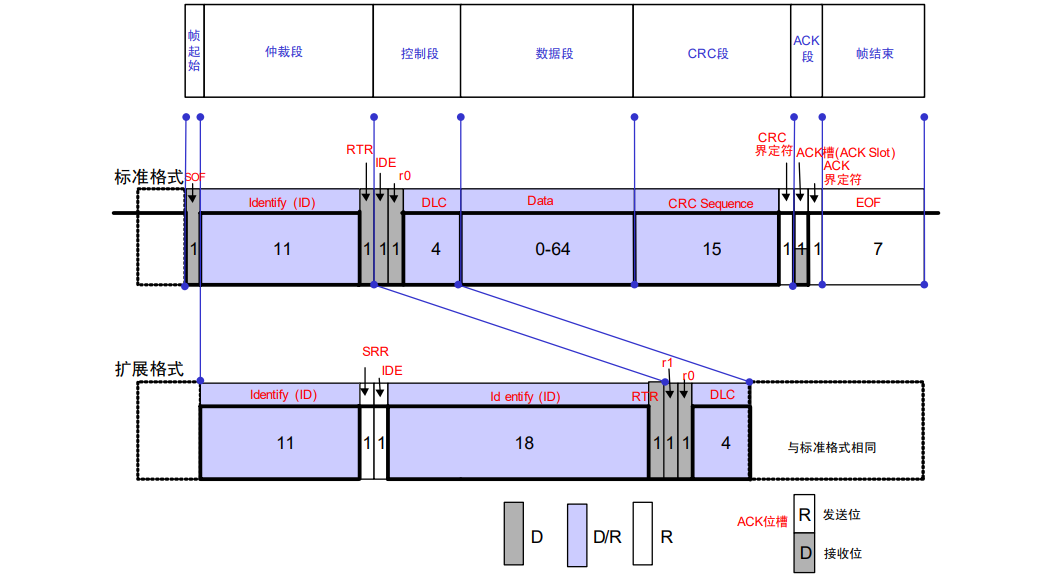
CAN\_High、CAN\_Low的差分电压与逻辑值0、1的关系：

| 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 PORTA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。

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 NewState);

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\*RxMessage);

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。

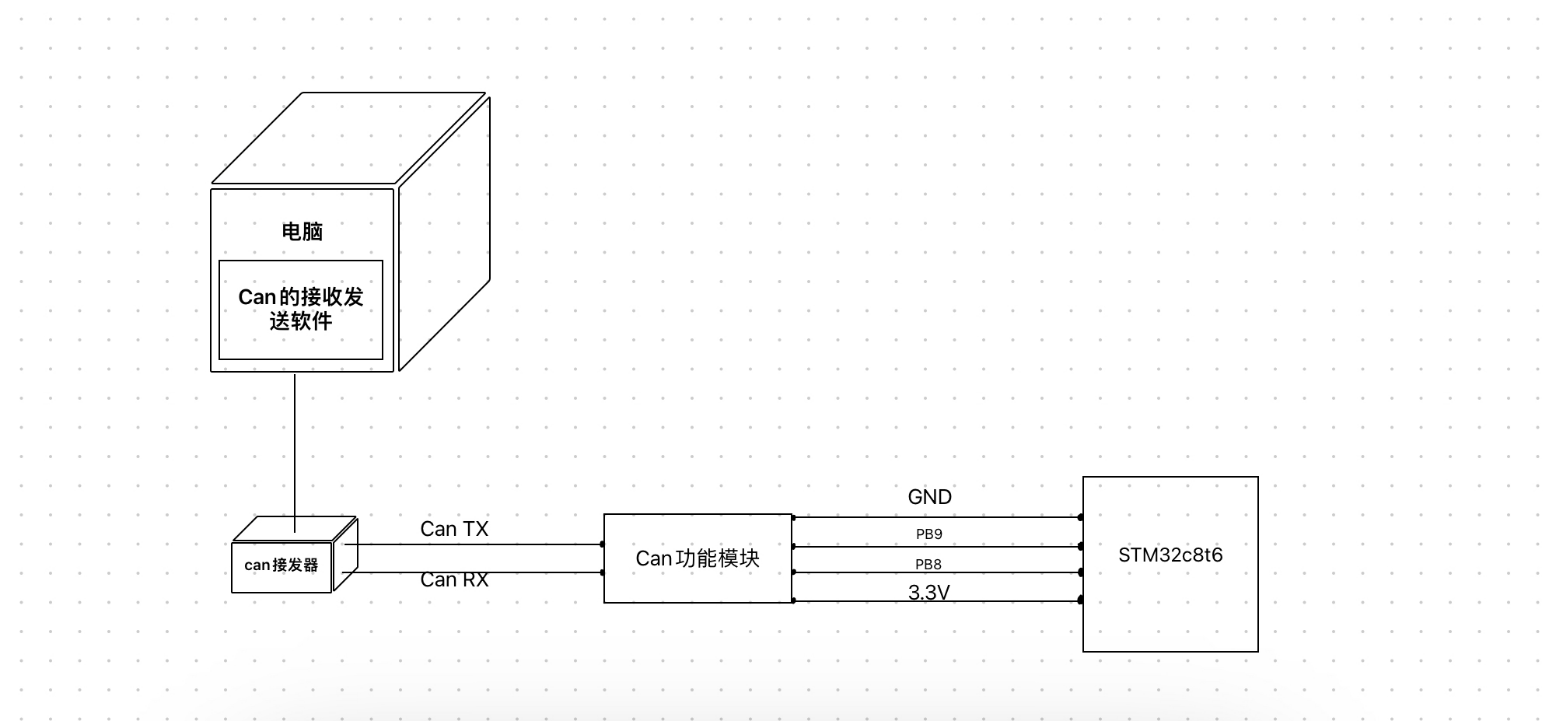
## 20.7.6 CAN 状态获取

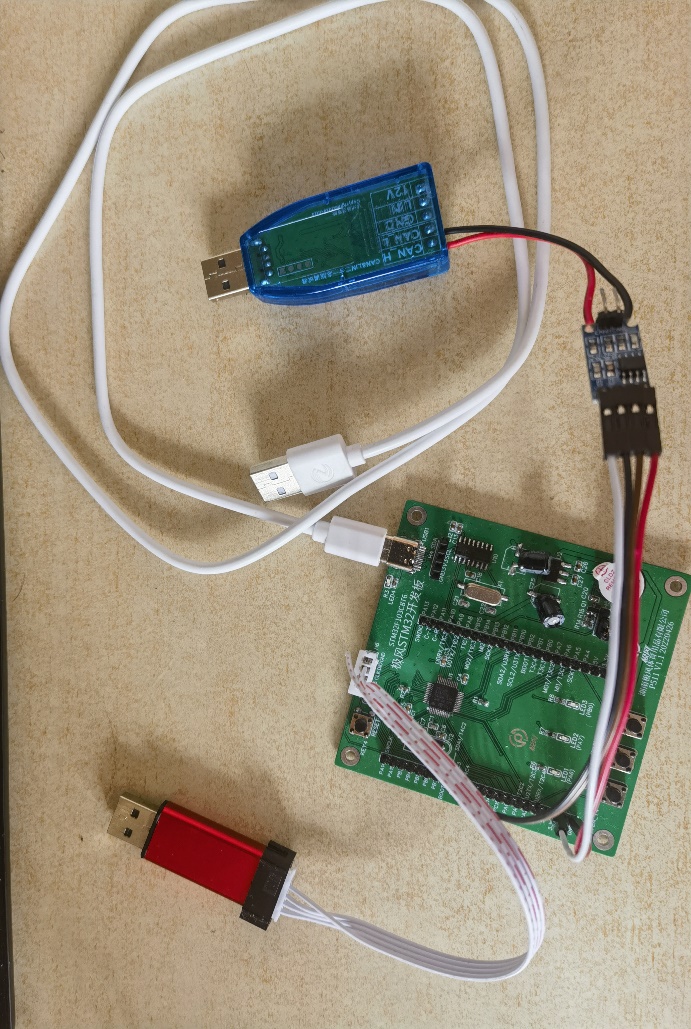
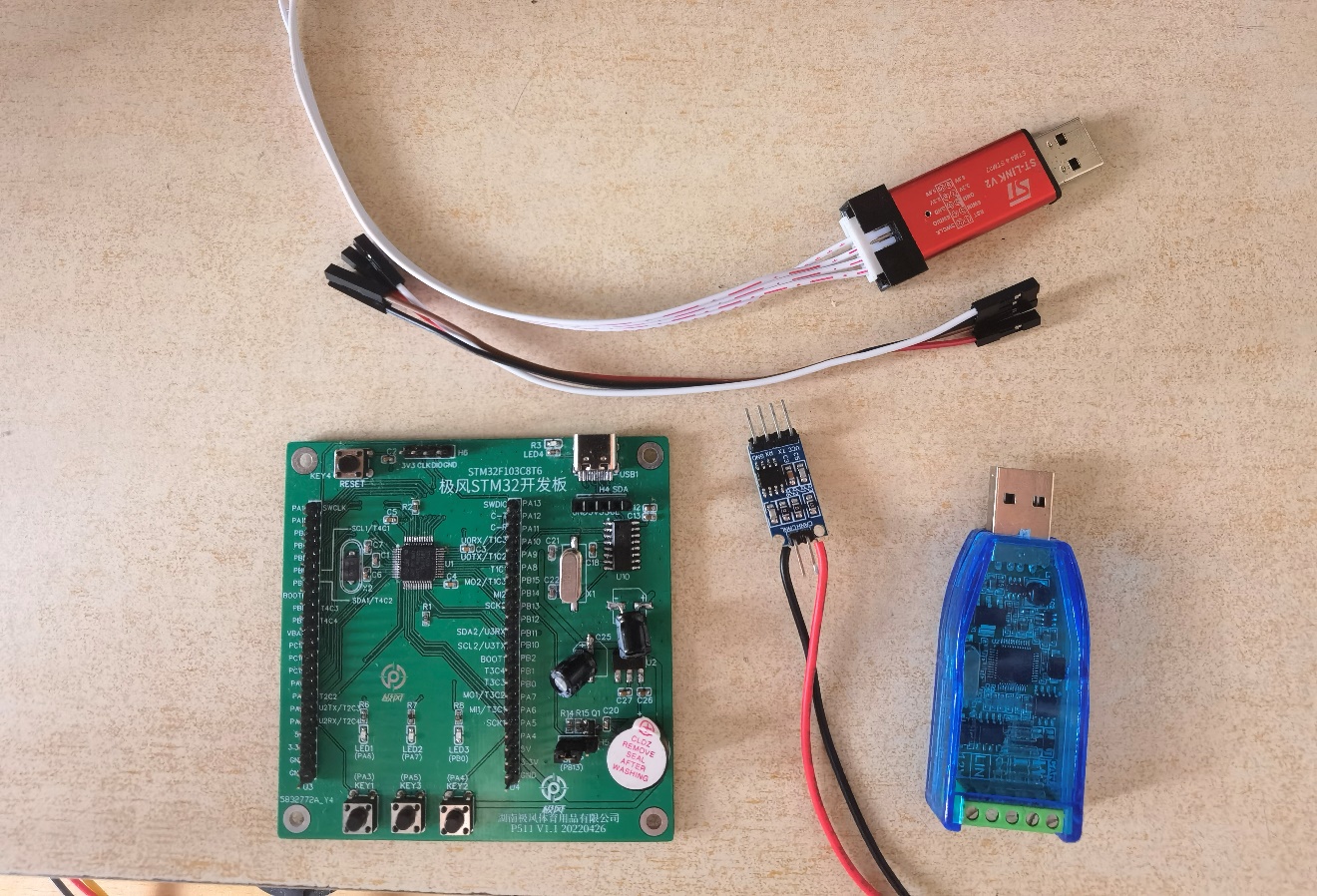
当使用 CAN 进行数据传输时， 我们会通过获取 CAN 状态标志来确认是否传输完成，比如说在接收报文时，通过检测标志位获知接收 FIFO 的状态，若收到报文，可调用库函数 CAN\_Receive 把接收 FIFO 中的内容读取到预先定义的接收类型结构体中，然后再访问该结构体即可利用报文了。

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。

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。



1. 当按下KEYX时，点亮对应的LED，并返回KEY的序号X。

