CAN

参考资料

```
https://www.cnblogs.com/luoxiao23/p/11210592.html?tdsourcetag=s_pctim_aiomsg
```

硬件连接

两根线 `CAN_High` & `CAN_Low`

1. **闭环总线网络（常用）**

   CAN 物理层的形式主要有两种， 图 43-1 中的 CAN 通讯网络是一种遵循 ISO11898 标准的高速、短距离“闭环网络”，它的**总线最大长度为 40m，通信速度最高为 1Mbps**，总线的两端各要求有一个“120 欧”的电阻。

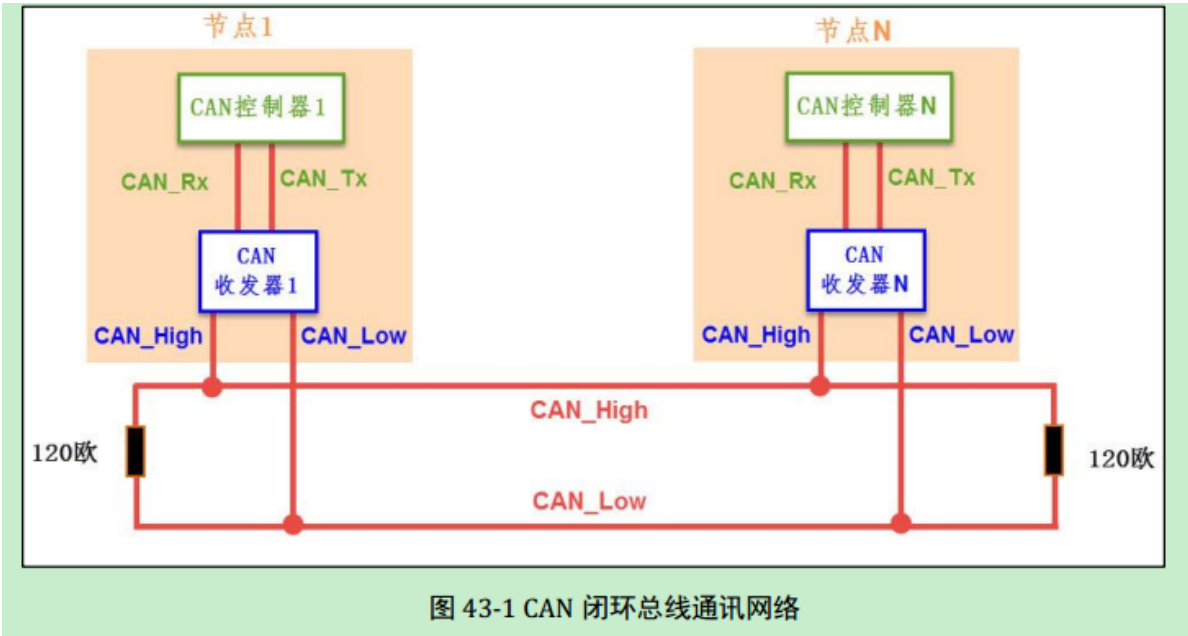


图 43-1 CAN 闭环总线通讯网络

2. 开环总线网络

   图 43-2 中的是遵循 ISO11519-2 标准的低速、远距离“开环网络”，**它的最大传输距离为 1km，最高通讯速率为 125kbps**，两根总线是独立的、不形成闭环，要求每根总线上各串联有一个“2.2 千欧”的电阻。

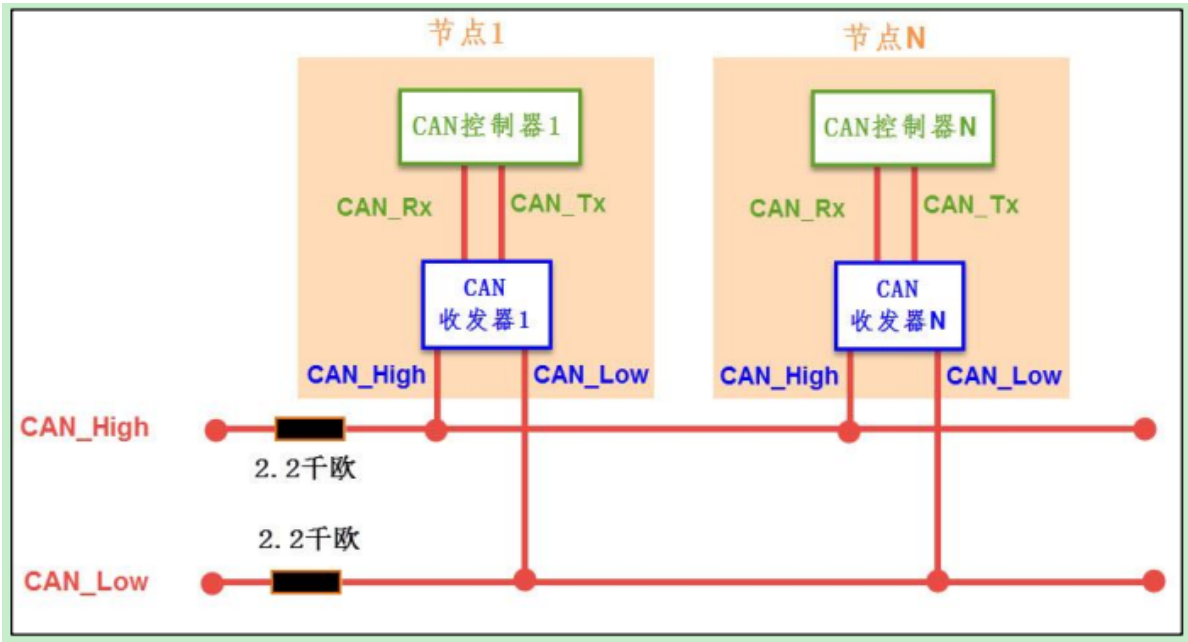


图 43-2 CAN 开环总线通讯网络

CAN 通讯节点由一个 CAN 控制器及 CAN 收发器组成，控制器与收发器之间通过CAN_Tx 及 CAN_Rx 信号线相连，收发器与 CAN 总线之间使用 CAN_High 及 CAN_Low信号线相连。其中 CAN_Tx 及 CAN_Rx 使用普通的类似 TTL 逻辑信号，而 CAN_High 及CAN_Low 是一对差分信号线，使用比较特别的差分信号。

协议特点

1. CAN属于异步通信 没有时钟信号线，连接在同一个总线网络中的各个节点会像串口异步通讯那样，节点间使用约定好的波特率进行通讯
2. 特别地， CAN 还会使用“位同步” 的方式来抗干扰、吸收误差，实现对总线电平信号进行正确的采样，确保通讯正常。

位时序

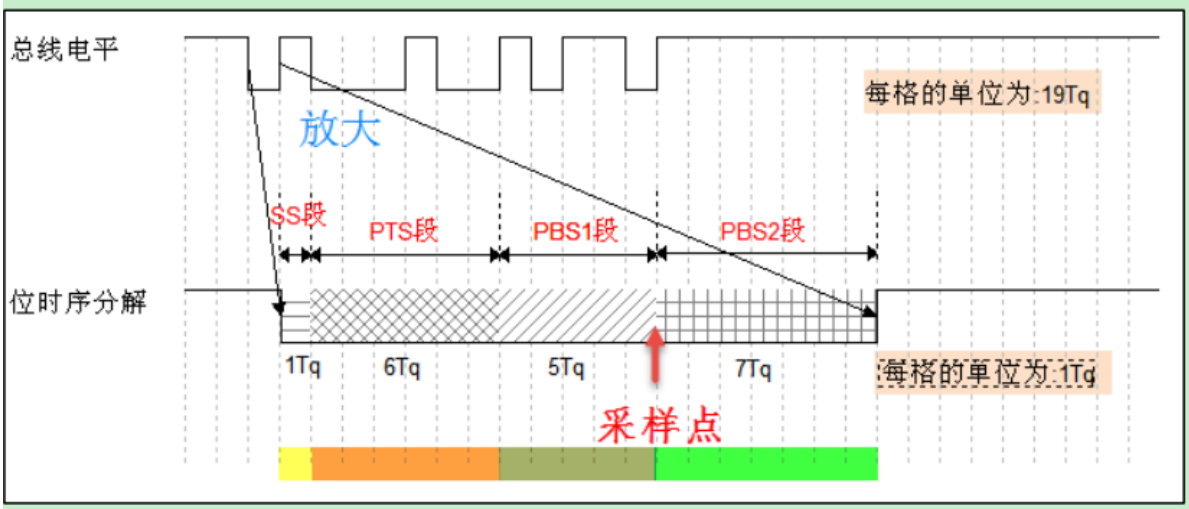


图 43-5 CAN 位时序分解图

为了实现位同步， CAN 协议把每一个数据位的时序分解成如图 43-5 所示的 SS 段、PTS 段、 PBS1 段、PBS2 段，**这四段的长度加起来即为一个 CAN 数据位的长度。**分解后最小的时间单位是 **Tq**，而一个完整的位由 8~25 个 Tq 组成。为方便表示， 图 43-5 中的高低电平直接代表信号逻辑 0 或逻辑 1(不是差分信号)。

波特率

总线上的各个通讯节点只要**约定好 1 个 Tq 的时间长度以及每一个数据位占据多少个Tq，**就可以确定 CAN 通讯的波特率。

例如，假设上图中的 1Tq=1us，而每个数据位由 19 个 Tq 组成，则传输一位数据需要时间 T =19us，从而**每秒可以传输的数据位个数**为：

$$1 * 10^6/19 = 52631.6(bps)$$

这个每秒可传输的数据位的个数即为通讯中的波特率。CAN通信需要注意总线上所有设备波特率设置要一致否则无法收到正确的消息

报文格式

在原始数据段的前面加上**传输起始标签、片选(识别)标签和控制标签，在数据的尾段加上 CRC 校验标签、应答标签和传输结束标签**，把这些内容按特定的格式打包好，就可以用一个通道表达各种信号了，各种各样的标签就如同 SPI 中各种通道上的信号，起到了协同传输的作用。当整个数据包被传输到其它设备时，只要这些设备按格式去解读，就能还原出原始数据，这样的报文就被称为 CAN 的“数据帧”。

为了更有效地控制通讯， CAN 一共规定了 5 种类型的帧，它们的类型及用途说明如表

帧	帧用途
数据帧	用于节点向外传送数据
遥控帧	用于向远端节点请求数据
错误帧	用于向远端节点通知校验错误，请求重新发送上一个数据
过载帧	用于通知远端节点：本节点尚未做好接收准备
帧间隔	用于将数据帧及遥控帧与前面的帧分离开来

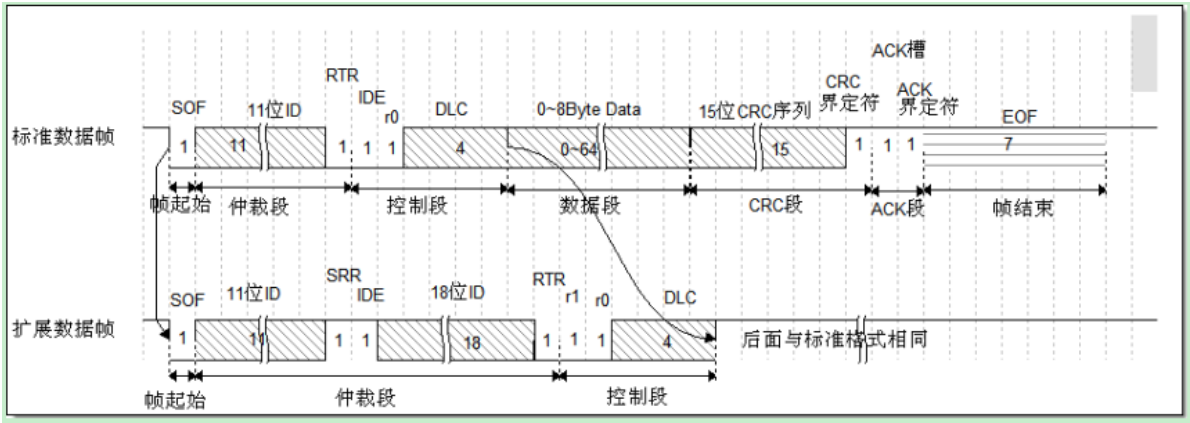


图 43-9 数据帧的结构

仲裁段的内容主要为本数据帧的 **ID 信息(标识符)**，数据帧具有**标准格式和扩展格式**两种，区别就在于 ID 信息的长度，**标准格式的 ID 为 11 位，扩展格式的 ID 为 29 位**，它在标准 ID 的基础上多出 18 位。

CAN ID

在 CAN 协议中， ID 起着重要的作用，它决定着数据帧发送的优先级，也决定着其它节点是否会接收这个数据帧。 CAN 协议不对挂载在它之上的节点分配优先级和地址，对总线的占有权是由信息的重要性决定的，即对于重要的信息，我们会给它打包上一个优先级高的 ID，使它能够及时地发送出去。也正因为它这样的优先级分配原则，使得 CAN 的扩展性大大加强，在总线上增加或减少节点并不影响其它设备。报文的优先级，是通过对 ID 的仲裁来确定的。根据前面对物理层的分析我们知道如果总线上同时出现显性电平和隐性电平，总线的状态会被置为显性电平， CAN 正是利用这个特性进行仲裁。

若两个节点同时竞争 CAN 总线的占有权，当它们发送报文时， 若首先出现隐性电平，则会失去对总线的占有权，进入接收状态。见图 43-10，在开始阶段，两个设备发送的电平一样，所以它们一直继续发送数据。到了图中箭头所指的时序处，节点单元 1 发送的为隐性电平，而此时节点单元 2 发送的为显性电平，由于总线的“线与”特性使它表达出显示电平，因此单元 2 竞争总线成功，这个报文得以被继续发送出去。

仲裁段 ID 的优先级也影响着接收设备对报文的反应。因为在 CAN 总线上数据是以广播的形式发送的，所有连接在 CAN 总线的节点都会收到所有其它节点发出的有效数据，因而我们的 CAN 控制器大多具有根据 ID 过滤报文的功能，它可以控制自己只接收某些 ID的报文。

RTR、IDE

仲裁段除了报文 ID 外，还有 RTR、 IDE 和 SRR位。

(1) RTR 位(Remote Transmission Request Bit)，译作远程传输请求位，它是用于区分**数据帧和遥控帧**的，当它为显性电平时表示数据帧，隐性电平时表示遥控帧。

(2) IDE 位(Identifier Extension Bit)，译作标识符扩展位，它是用于区分**标准格式与扩展格式**，当它为显性电平时表示标准格式，隐性电平时表示扩展格式。

(3) SRR 位(Substitute Remote Request Bit)，只存在于扩展格式，它用于替代标准格式中的RTR 位。由于扩展帧中的 SRR 位为隐性位， RTR 在数据帧为显性位，所以在两个 ID相同的标准格式报文与扩展格式报文中，标准格式的优先级较高。

DLC

在控制段中的 r1 和 r0 为保留位，默认设置为显性位。它最主要的是 DLC 段(Data Length Code)，译为**数据长度码**，它由 4 个数据位组成，用于表示本报文中的数据段含有多少个字节， DLC 段表示的数字为 0~8。

数据段

数据段为数据帧的核心内容，它是节点要发送的原始信息，由 0~8 个字节组成， MSB先行。

STM32中的CAN

位时序

STM32 外设定义的位时序与我们前面解释的 CAN 标准时序有一点区别

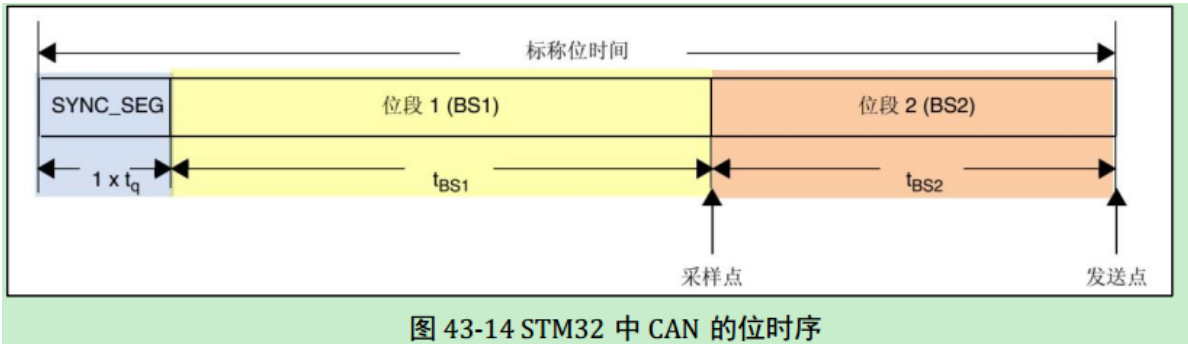


图 43-14 STM32 中 CAN 的位时序

STM32 的 CAN 外设位时序中只包含 3 段，分别是**同步段 SYNC_SEG、位段 BS1 及位段 BS2**，采样点位于 BS1 及 BS2 段的交界处。其中 SYNC_SEG 段固定长度为 1Tq，而BS1 及 BS2 段可以在位时序寄存器 CAN_BTR 设置它们的时间长度，它们可以在重新同步期间增长或缩短，该长度 SJW 也可在位时序寄存器中配置。

BS1 段时间：

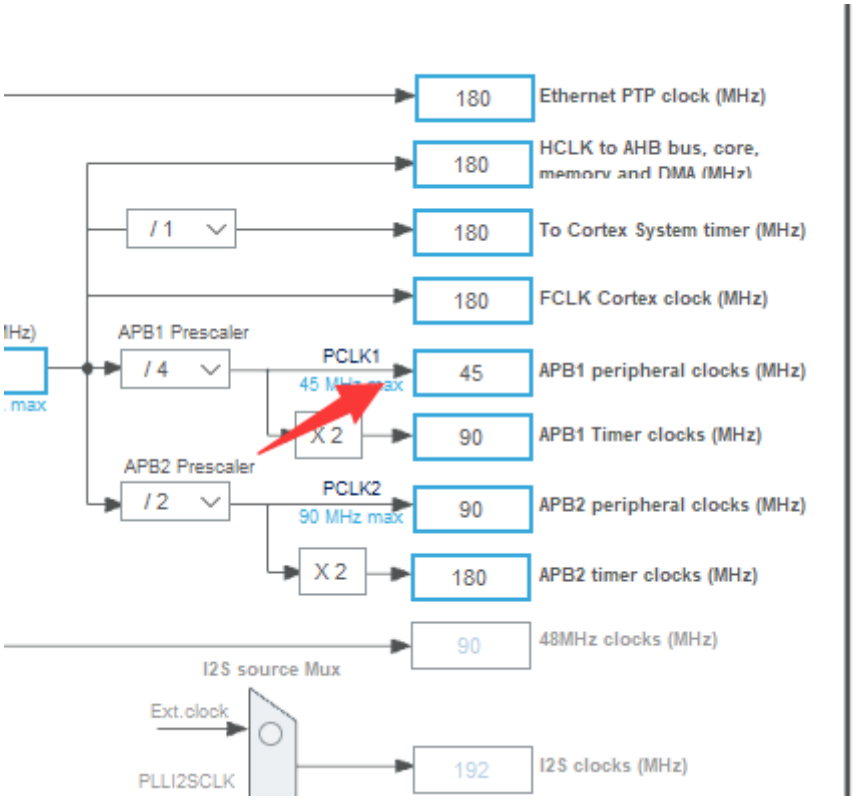
TS1=Tq x (TS1[3:0] + 1)，

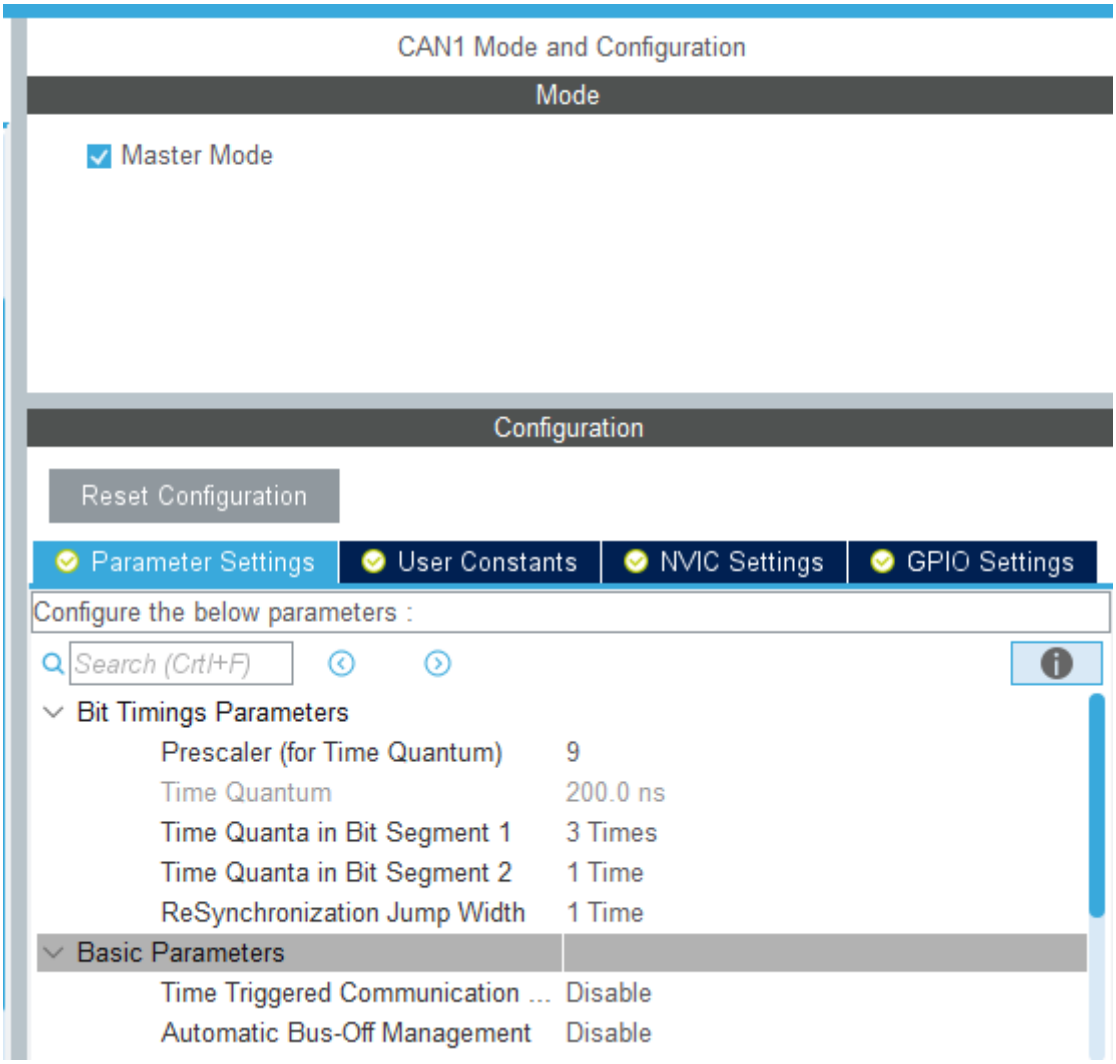
BS2 段时间：

TS2= Tq x (TS2[2:0] + 1)，

一个数据位的时间：

T1bit =1Tq+TS1+TS2 =1+ (TS1[3:0] + 1)+ (TS2[2:0] + 1)= N Tq





Tq = Prescaler /PCLK

其中的 PCLK 指 APB1 时钟，默认值为 45MHz。

CAN 通讯的波特率：

BaudRate = 1/((BS1+BS2+SJW)*Tq)

CAN发送邮箱

CAN 外设的发送邮箱，它一共有 3个发送邮箱，即最多可以缓存 3 个待发送的报文。每个发送邮箱中包含有标识符寄存器 CAN_TIxR、数据长度控制寄存器 CAN_TDTxR及 2 个数据寄存器 CAN_TDLxR、CAN_TDHxR，它们的功能见表

表 43-4 发送邮箱的寄存器

寄存器名	功能
标识符寄存器 CAN_TIxR	存储待发送报文的 ID、扩展 ID、IDE 位及 RTR 位
数据长度控制寄存器 CAN_TDTxR	存储待发送报文的 DLC 段
低位数据寄存器 CAN_TDLxR	存储待发送报文数据段的 Data0-Data3 这四个字节的内容
高位数据寄存器 CAN_TDHxR	存储待发送报文数据段的 Data4-Data7 这四个字节的内容

当我们要使用 CAN 外设发送报文时，把报文的各个段分解，按位置写入到这些寄存器中，并对标识符寄存器 CAN_TIxR 中的发送请求寄存器位 TMIDxR_TXRQ 置 1，即可把数据发送出去。

其中标识符寄存器 CAN_TIxR 中的 STDID 寄存器位比较特别。我们知道 CAN 的标准标识符的总位数为 11 位，而扩展标识符的总位数为 29 位的。当报文使用扩展标识符的时候，标识符寄存器 CAN_TIxR 中的 STDID[10:0]等效于 EXTID[18:28]位，它与 EXTID[17:0]共同组成完整的 29 位扩展标识符。

表 43-5 发送邮箱的寄存器

寄存器名	功能
标识符寄存器 CAN_RIxR	存储收到报文的 ID、扩展 ID、IDE 位及 RTR 位
数据长度控制寄存器 CAN_RDTxR	存储收到报文的 DLC 段
低位数据寄存器 CAN_RDLxR	存储收到报文数据段的 Data0-Data3 这四个字节的内容
高位数据寄存器 CAN_RDHxR	存储收到报文数据段的 Data4-Data7 这四个字节的内容

注意 当CAN发送邮箱内堆满消息后，就无法放入新的消息了，此时就无法发送CAN消息

CAN 接收 FIFO

CAN 外设的接收 FIFO，它一共有 2 个接收 FIFO，每个 FIFO 中有 3 个邮箱，即最多可以缓存 6 个接收到的报文。当接收到报文时， FIFO 的报文计数器会自增，而 STM32 内部读取 FIFO 数据之后，报文计数器会自减，我们通过状态寄存器可获知报文计数器的值，而通过前面主控制寄存器的 RFLM 位，可设置锁定模式，锁定模式下 FIFO 溢出时会丢弃新报文，非锁定模式下 FIFO 溢出时新报文会覆盖旧报文。

跟发送邮箱类似，每个接收 FIFO 中包含有标识符寄存器 CAN_RIxR、数据长度控制寄存器CAN_RDTxR 及 2 个数据寄存器 CAN_RDLxR、 CAN_RDHxR，它们的功能见表43-5。

通过中断或状态寄存器知道接收 FIFO 有数据后，我们再读取这些寄存器的值即可把接收到的报文加载到 STM32 的内存中。

注意 当在中断中知道FIFO有新数据但是不做处理 数据仍然还在FIFO中 这会导致接收FIFO被塞满 无法接收新消息

CAN过滤器(Filter)

CAN过滤器只存在于接收结构体，它存储了筛选器的编号，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。

CAN 的筛选器有多种工作模式，利用筛选器结构体可方便配置

```
/**

* @brief CAN filter init structure definition

* CAN 筛选器结构体

*/

typedef struct {

    uint16_t CAN_FilterIdHigh; /*CAN_FxR1 寄存器的高 16 位 */

    uint16_t CAN_FilterIdLow; /*CAN_FxR1 寄存器的低 16 位*/

    uint16_t CAN_FilterMaskIdHigh; /*CAN_FxR2 寄存器的高 16 位*/

    uint16_t CAN_FilterMaskIdLow; /*CAN_FxR2 寄存器的低 16 位 */

    uint16_t CAN_FilterFIFOAssignment; /*设置经过筛选后数据存储到哪个接收 FIFO
```

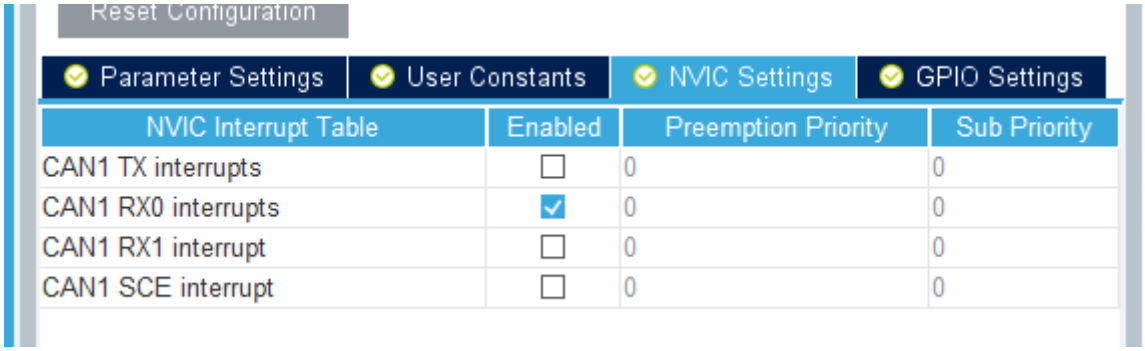
```
    */

    uint8_t CAN_FilterNumber; /*筛选器编号，范围 0-27*/

    uint8_t CAN_FilterMode; /*筛选器模式 */

    uint8_t CAN_FilterScale; /*设置筛选器的尺度 */

    FunctionalState CAN_FilterActivation; /*是否使能本筛选器*/

} CAN_FilterInitTypeDef;
```

最后附上CAN Filter的**祖传代码** 这里默认将收到的can消息原封不动地存入FIFO0中 如果需要滤去某些数据由我们自己在后续解码中处理

```
HAL_StatusTypeDef CANFilterInit(CAN_HandleTypeDef* hcan){
  CAN_FilterTypeDef  sFilterConfig;

  sFilterConfig.FilterBank = 0;
  sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
  sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
  sFilterConfig.FilterIdHigh = 0x0000;
  sFilterConfig.FilterIdLow = 0x0000;
  sFilterConfig.FilterMaskIdHigh = 0x0000;
  sFilterConfig.FilterMaskIdLow = 0x0000;
  sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0;
  sFilterConfig.FilterActivation = ENABLE;
  sFilterConfig.SlaveStartFilterBank = 14;


  if(HAL_CAN_ConfigFilter(hcan, &sFilterConfig) != HAL_OK)
  {
    Error_Handler();
  }

  if (HAL_CAN_Start(hcan) != HAL_OK)
  {
    Error_Handler();
  }

  if (HAL_CAN_ActivateNotification(hcan, CAN_IT_RX_FIFO0_MSG_PENDING) != HAL_OK)
  {
    Error_Handler();
  }

    return HAL_OK;
}
```

Cube配置

除了之前配置CAN波特率以外 还需要额外点上接收中断和是否使用DMA，接收中断注意只点RX0的即可对应FIFO0



HAL库函数

```
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)//FIFO0 接收中断回调函数
void HAL_CAN_RxFifo1MsgPendingCallback(CAN_HandleTypeDef *hcan)//FIFO1 接收中断回调函数

HAL_StatusTypeDef HAL_CAN_AddTxMessage(CAN_HandleTypeDef *hcan,
CAN_TxHeaderTypeDef *pHeader, uint8_t aData[], uint32_t *pTxMailbox)//向发送邮箱中添加一条can消息
HAL_StatusTypeDef HAL_CAN_GetRxMessage(CAN_HandleTypeDef *hcan, uint32_t RxFifo,
CAN_RxHeaderTypeDef *pHeader, uint8_t aData[])//从FIFOx中取出一条CAN消息
uint32_t HAL_CAN_GetTxMailboxesFreeLevel(CAN_HandleTypeDef *hcan);//获取当前发送邮箱状态
```