第39章　CAN—通讯实验

本章参考资料：《STM32 参考手册》、《STM32F103CDE 增强型系列数据手册》、库帮助文档《stm32f10x_stdperiph_lib_um.chm》。

若对 CAN 通讯协议不了解，可先阅读《CAN 总线入门》、《CAN-bus 规范》文档内容学习。

关于实验板上的 CAN 收发器可查阅《TJA1050》文档了解。

39.1 CAN 协议简介

CAN 是控制器局域网络(Controller Area Network)的简称，它是由研发和生产汽车电子产品著称的德国 BOSCH 公司开发的，并最终成为国际标准（ISO11519），是国际上应用最广泛的现场总线之一。

CAN 总线协议已经成为汽车计算机控制系统和嵌入式工业控制局域网的标准总线，并且拥有以 CAN 为底层协议专为大型货车和重工机械车辆设计的 J1939 协议。近年来，它具有的高可靠性和良好的错误检测能力受到重视，被广泛应用于汽车计算机控制系统和环境温度恶劣、电磁辐射强及振动大的工业环境。

39.1.1 CAN 物理层

与 I2C、SPI 等具有时钟信号的同步通讯方式不同，CAN 通讯并不是以时钟信号来进行同步的，它是一种异步通讯，只具有 CAN_High 和 CAN_Low 两条信号线，共同构成一组差分信号线，以差分信号的形式进行通讯。

1. 闭环总线网络

CAN 物理层的形式主要有两种，图 39-1 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。

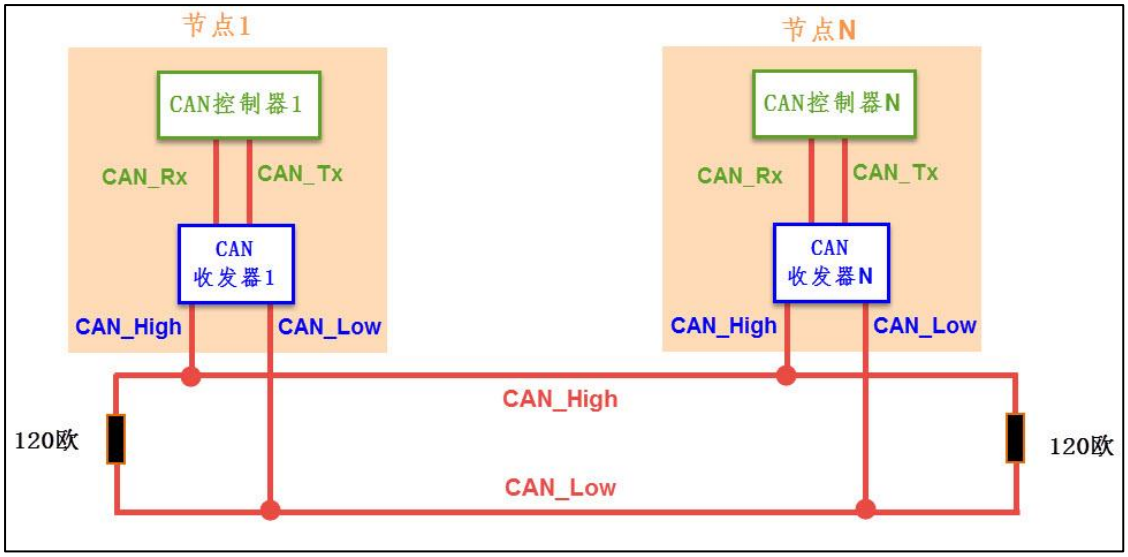

图 39-1 CAN 闭环总线通讯网络

2. 开环总线网络

图 39-2 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。

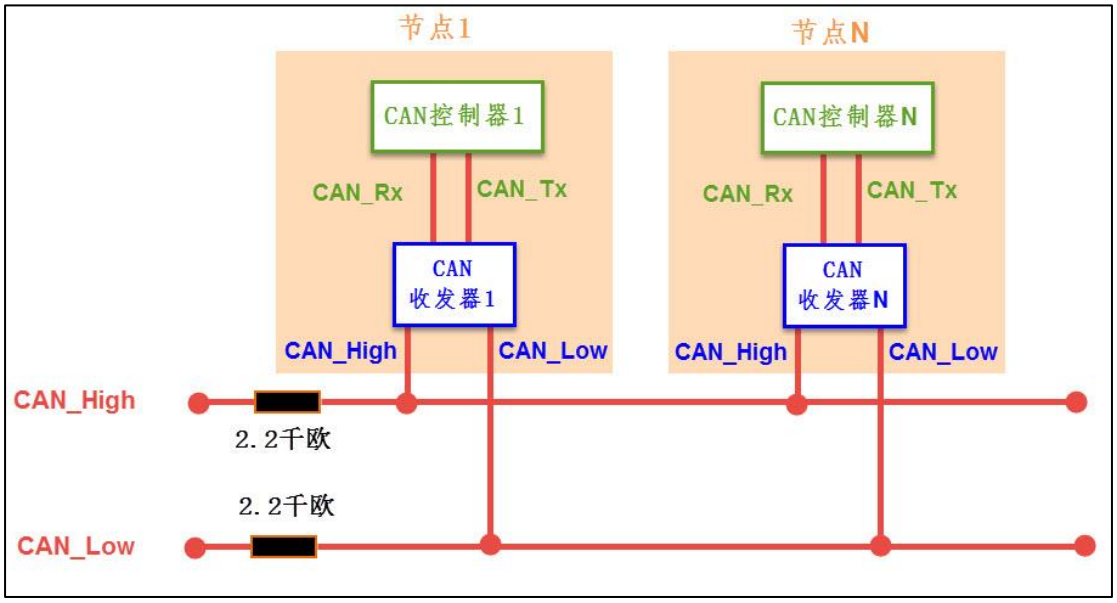

图 39-2 CAN 开环总线通讯网络

3. 通讯节点

从 CAN 通讯网络图可了解到，CAN 总线上可以挂载多个通讯节点，节点之间的信号经过总线传输，实现节点间通讯。由于 CAN 通讯协议不对节点进行地址编码，而是对数据内容进行编码的，所以网络中的节点个数理论上不受限制，只要总线的负载足够即可，可以通过中继器增强负载。

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，下一小节再详细说明。

当 CAN 节点需要发送数据时，控制器把要发送的二进制编码通过 CAN_Tx 线发送到收发器，然后由收发器把这个普通的逻辑电平信号转化成差分信号，通过差分线 CAN_High 和 CAN_Low 线输出到 CAN 总线网络。而通过收发器接收总线上的数据到控制器时，则是相反的过程，收发器把总线上收到的 CAN_High 及 CAN_Low 信号转化成普通的逻辑电平信号，通过 CAN_Rx 输出到控制器中。

例如，STM32 的 CAN 片上外设就是通讯节点中的控制器，为了构成完整的节点，还要给它外接一个收发器，在我们实验板中使用型号为 TJA1050 的芯片作为 CAN 收发器。CAN 控制器与 CAN 收发器的关系如同 TTL 串口与 MAX3232 电平转换芯片的关系，MAX3232 芯片把 TTL 电平的串口信号转换成 RS-232 电平的串口信号，CAN 收发器的作用则是把 CAN 控制器的 TTL 电平信号转换成差分信号(或者相反)。

4. 差分信号

差分信号又称差模信号，与传统使用单根信号线电压表示逻辑的方式有区别，使用差分信号传输时，需要两根信号线，这两个信号线的振幅相等，相位相反，通过两根信号线的电压差值来表示逻辑 0 和逻辑 1。见图 39-3，它使用了 V+与 V-信号的差值表达出了图下方的信号。

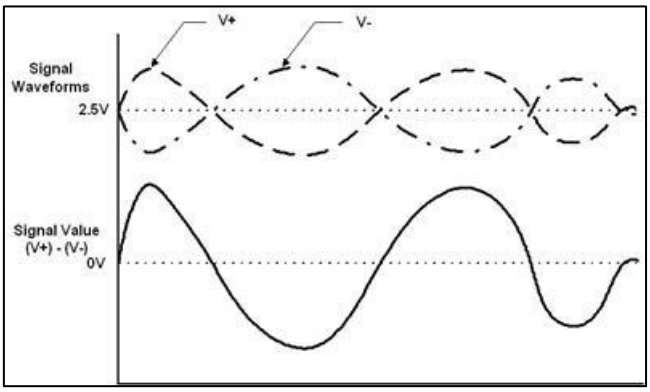

图 39-3 差分信号

相对于单信号线传输的方式，使用差分信号传输具有如下优点：

- 抗干扰能力强，当外界存在噪声干扰时，几乎会同时耦合到两条信号线上，而接收端只关心两个信号的差值，所以外界的共模噪声可以被完全抵消。
- 能有效抑制它对外部的电磁干扰，同样的道理，由于两根信号的极性相反，他们对外辐射的电磁场可以相互抵消，耦合的越紧密，泄放到外界的电磁能量越少。
- 时序定位精确，由于差分信号的开关变化是位于两个信号的交点，而不像普通单端信号依靠高低两个阈值电压判断，因而受工艺，温度的影响小，能降低时序上的误差，同时也更适合于低幅度信号的电路。

由于差分信号线具有这些优点，所以在 USB 协议、485 协议、以太网协议及 CAN 协议的物理层中，都使用了差分信号传输。

5. CAN 协议中的差分信号

CAN 协议中对它使用的 CAN_High 及 CAN_Low 表示的差分信号做了规定，见表 39-1 及图 39-4。以高速 CAN 协议为例，当表示逻辑 1 时(隐性电平)，CAN_High 和 CAN_Low 线上的电压均为 2.5v，即它们的电压差 V_H-V_L=0V；而表示逻辑 0 时(显性电平)，CAN_High 的电平为 3.5V，CAN_Low 线的电平为 1.5V，即它们的电压差为 V_H-V_L=2V。例如，当 CAN 收发器从 CAN_Tx 线接收到来自 CAN 控制器的低电平信号时(逻辑 0)，它会使 CAN_High 输出 3.5V，同时 CAN_Low 输出 1.5V，从而输出显性电平表示逻辑 0。

表 39-1 CAN 协议标准表示的信号逻辑

信号	ISO11898(高速)						ISO11519-2(低速)					
	隐性(逻辑 1)			显性(逻辑 0)			隐性(逻辑 1)			显性(逻辑 0)		
	最小值	典型值	最大值	最小值	典型值	最大值	最小值	典型值	最大值	最小值	典型值	最大值
CAN_High（V）	2.0	2.5	3.0	2.75	3.5	4.5	1.6	1.75	1.9	3.85	4.0	5.0
CAN_Low（V）	2.0	2.5	3.0	0.5	1.5	2.25	3.10	3.25	3.4	0	1.0	1.15
High-Low 电位差 (V)	-0.5	0	0.05	1.5	2.0	3.0	-0.3	-1.5	-	0.3	3.0	-

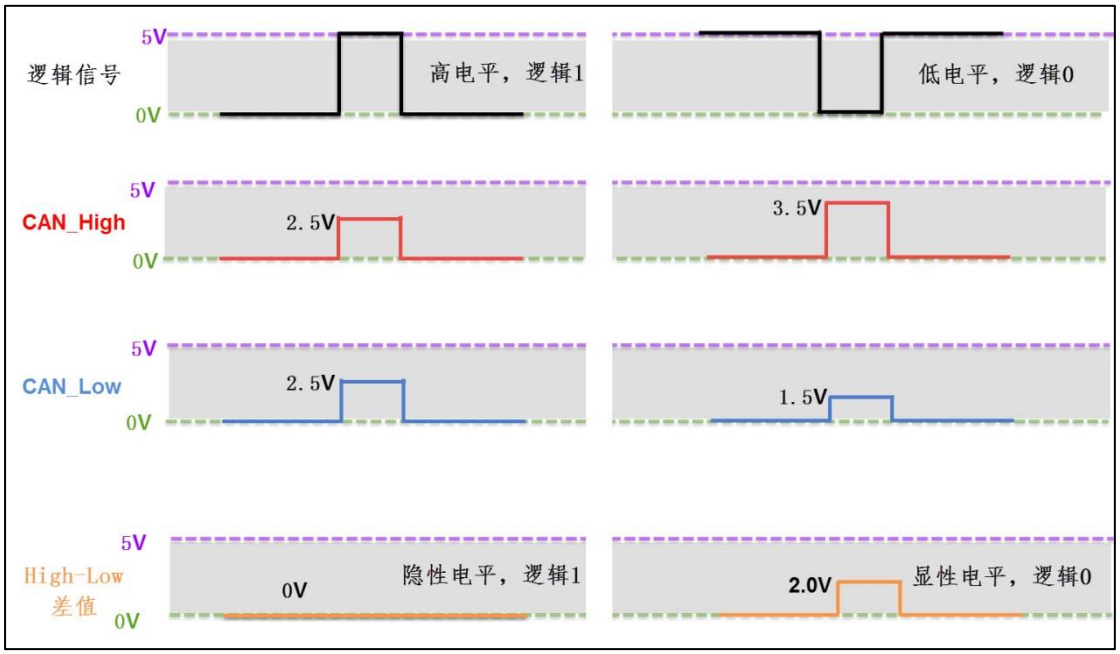

图 39-4 CAN 的差分信号（高速）

在 CAN 总线中，必须使它处于隐性电平(逻辑 1)或显性电平(逻辑 0)中的其中一个状态。假如有两个 CAN 通讯节点，在同一时间，一个输出隐性电平，另一个输出显性电平，类似 I2C 总线的“线与”特性将使它处于显性电平状态，显性电平的名字就是这样来的，即可以认为显性具有优先的意味。

由于 CAN 总线协议的物理层只有 1 对差分线，在一个时刻只能表示一个信号，所以对通讯节点来说，CAN 通讯是半双工的，收发数据需要分时进行。在 CAN 的通讯网络中，因为共用总线，在整个网络中同一时刻只能有一个通讯节点发送信号，其余的节点在该时刻都只能接收。

39.1.2 协议层

以上是 CAN 的物理层标准，约定了电气特性，以下介绍的协议层则规定了通讯逻辑。

1. CAN 的波特率及位同步

由于 CAN 属于异步通讯，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，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。

位时序分解

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 39-5 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 39-5 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)。

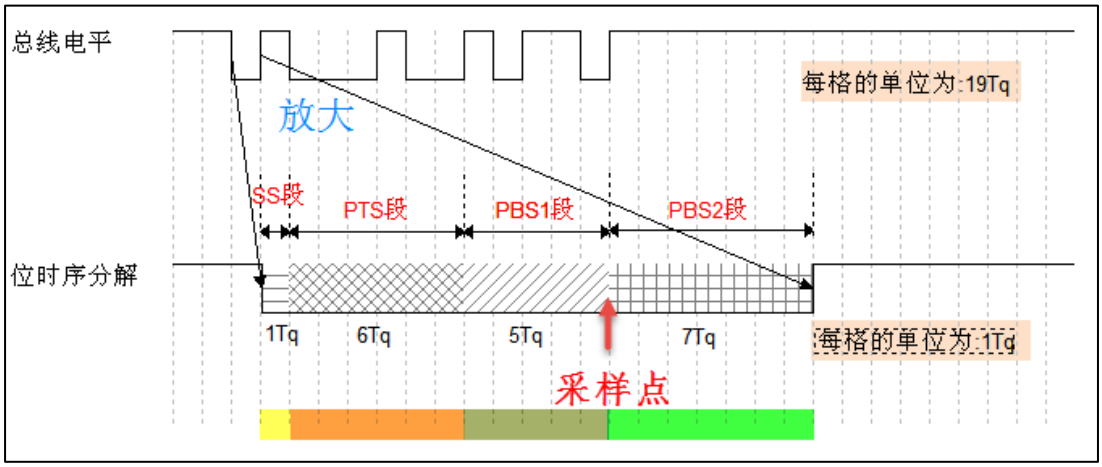

图 39-5 CAN 位时序分解图

该图中表示的 CAN 通讯信号每一个数据位的长度为 19Tq，其中 SS 段占 1Tq，PTS 段占 6Tq，PBS1 段占 5Tq，PBS2 段占 7Tq。信号的采样点位于 PBS1 段与 PBS2 段之间，通过控制各段的长度，可以对采样点的位置进行偏移，以便准确地采样。

各段的作用如介绍下：

- SS 段(SYNC SEG)

  SS 译为同步段，若通讯节点检测到总线上信号的跳变沿被包含在 SS 段的范围之内，则表示节点与总线的时序是同步的，当节点与总线同步时，采样点采集到的总线电平即可被确定为该位的电平。SS 段的大小固定为 1Tq。

- PTS 段(PROP SEG)

PTS 译为传播时间段，这个时间段是用于补偿网络的物理延时时间。是总线上输入比较器延时和输出驱动器延时总和的两倍。PTS 段的大小可以为 1~8Tq。

- PBS1 段(PHASE SEG1)，

  PBS1 译为相位缓冲段，主要用来补偿边沿阶段的误差，它的时间长度在重新同步的时候可以加长。PBS1 段的初始大小可以为 1~8Tq。

- PBS2 段(PHASE SEG2)

  PBS2 这是另一个相位缓冲段，也是用来补偿边沿阶段误差的，它的时间长度在重新同步时可以缩短。PBS2 段的初始大小可以为 2~8Tq。

通讯的波特率

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_{1bit} =19us，从而每秒可以传输的数据位个数为：

$$1x10^6/19 = 52631.6 \text{ (bps)}$$

这个每秒可传输的数据位的个数即为通讯中的波特率。

同步过程分析

波特率只是约定了每个数据位的长度，数据同步还涉及到相位的细节，这个时候就需要用到数据位内的 SS、PTS、PBS1 及 PBS2 段了。

根据对段的应用方式差异，CAN 的数据同步分为硬同步和重新同步。其中硬同步只是当存在“帧起始信号”时起作用，无法确保后续一连串的位时序都是同步的，而重新同步方式可解决该问题，这两种方式具体介绍如下：

(1) 硬同步

若某个 CAN 节点通过总线发送数据时，它会发送一个表示通讯起始的信号(即下一小节介绍的帧起始信号)，该信号是一个由高变低的下降沿。而挂载到 CAN 总线上的通讯节点在不发送数据时，会时刻检测总线上的信号。

见图 39-6，可以看到当总线出现帧起始信号时，某节点检测到总线的帧起始信号不在节点内部时序的 SS 段范围，所以判断它自己的内部时序与总线不同步，因而这个状态的采样点采集得的数据是不正确的。所以节点以硬同步的方式调整，把自己的位时序中的 SS 段平移至总线出现下降沿的部分，获得同步，同步后采样点就可以采集得正确数据了。

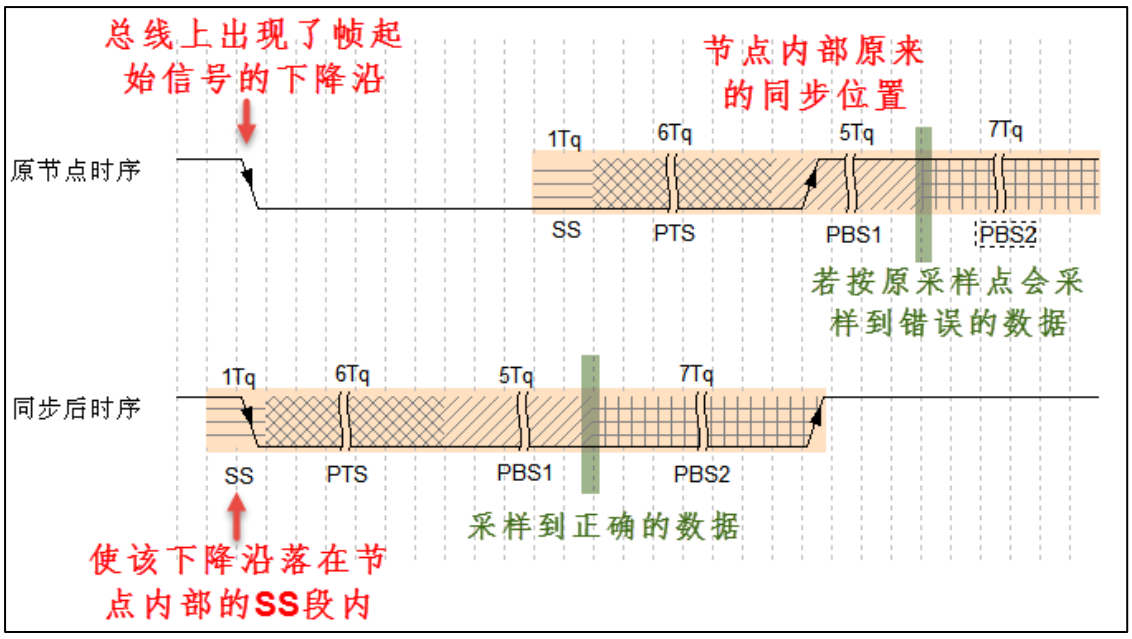

图 39-6 硬同步过程图

(2) 重新同步

前面的硬同步只是当存在帧起始信号时才起作用，如果在一帧很长的数据内，节点信号与总线信号相位有偏移时，这种同步方式就无能为力了。因而需要引入重新同步方式，它利用普通数据位的高至低电平的跳变沿来同步(帧起始信号是特殊的跳变沿)。重新同步与硬同步方式相似的地方是它们都使用 SS 段来进行检测，同步的目的都是使节点内的 SS 段把跳变沿包含起来。

重新同步的方式分为超前和滞后两种情况，以总线跳变沿与 SS 段的相对位置进行区分。第一种相位超前的情况如图 39-7，节点从总线的边沿跳变中，检测到它内部的时序比总线的时序相对超前 2Tq，这时控制器在下一个位时序中的 PBS1 段增加 2Tq 的时间长度，使得节点与总线时序重新同步。

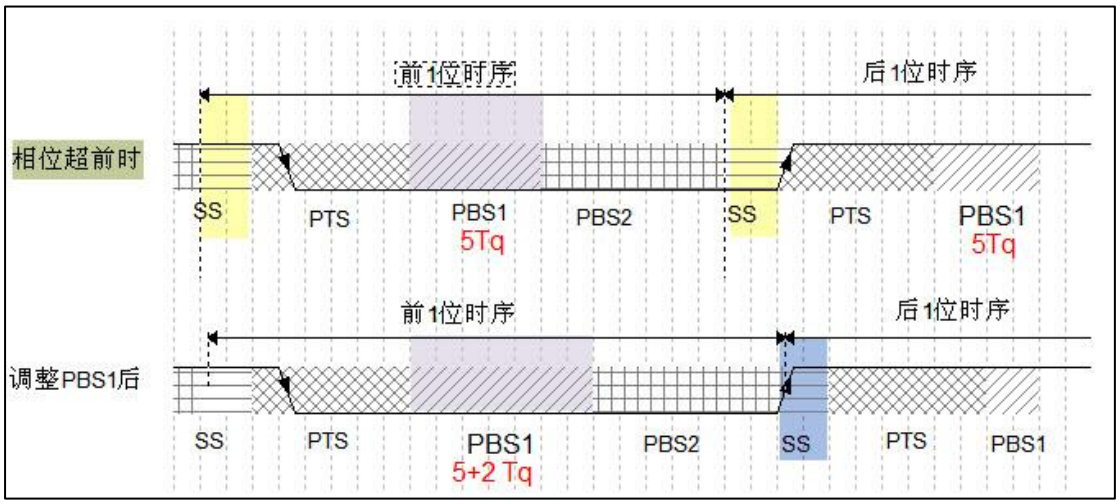

图 39-7 相位超前时的重新同步

第二种相位滞后的情况如图 39-8，节点从总线的边沿跳变中，检测到它的时序比总线的时序相对滞后 2Tq，这时控制器在前一个位时序中的 PBS2 段减少 2Tq 的时间长度，获得同步。

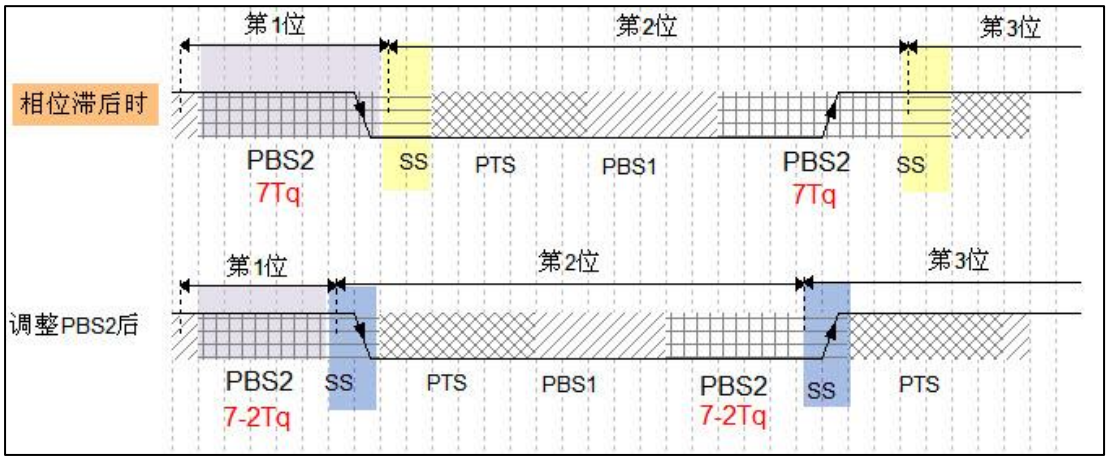

图 39-8 相位滞后时的重新同步

在重新同步的时候，PBS1 和 PBS2 中增加或减少的这段时间长度被定义为“重新同步补偿宽度 SJW (reSynchronization Jump Width)”。一般来说 CAN 控制器会限定 SJW 的最大值，如限定了最大 SJW=3Tq 时，单次同步调整的时候不能增加或减少超过 3Tq 的时间长度，若有需要，控制器会通过多次小幅度调整来实现同步。当控制器设置的 SJW 极限值较大时，可以吸收的误差加大，但通讯的速度会下降。

2. CAN 的报文种类及结构

在 SPI 通讯中，片选、时钟信号、数据输入及数据输出这 4 个信号都有单独的信号线，I2C 协议包含有时钟信号及数据信号 2 条信号线，异步串口包含接收与发送 2 条信号线，这些协议包含的信号都比 CAN 协议要丰富，它们能轻易进行数据同步或区分数据传输方向。而 CAN 使用的是两条差分信号线，只能表达一个信号，简洁的物理层决定了 CAN 必然要配上一套更复杂的协议，如何用一个信号通道实现同样、甚至更强大的功能呢？CAN 协议给出的解决方案是对数据、操作命令(如读/写)以及同步信号进行打包，打包后的这些内容称为报文。

报文的种类

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 39-2。

表 39-2 帧的种类及其用途

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

数据帧的结构

数据帧是在 CAN 通讯中最主要、最复杂的报文，我们来了解它的结构，见图 39-9。

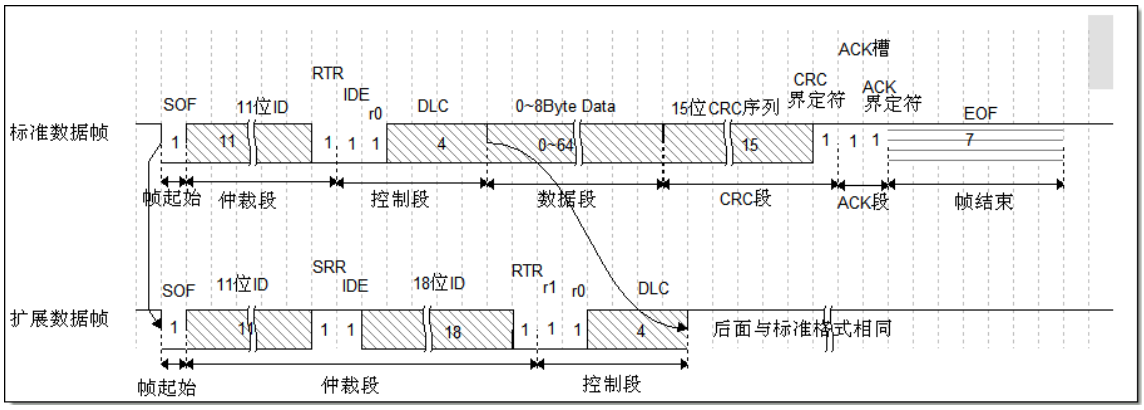

图 39-9 数据帧的结构

数据帧以一个显性位(逻辑 0)开始，以 7 个连续的隐性位(逻辑 1)结束，在它们之间，分别有仲裁段、控制段、数据段、CRC 段和 ACK 段。

- 帧起始

SOF 段(Start Of Frame)，译为帧起始，帧起始信号只有一个数据位，是一个显性电平，它用于通知各个节点将有数据传输，其它节点通过帧起始信号的电平跳变沿来进行硬同步。

- 仲裁段

当同时有两个报文被发送时，总线会根据仲裁段的内容决定哪个数据包能被传输，这也是它名称的由来。

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。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 39-10，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。

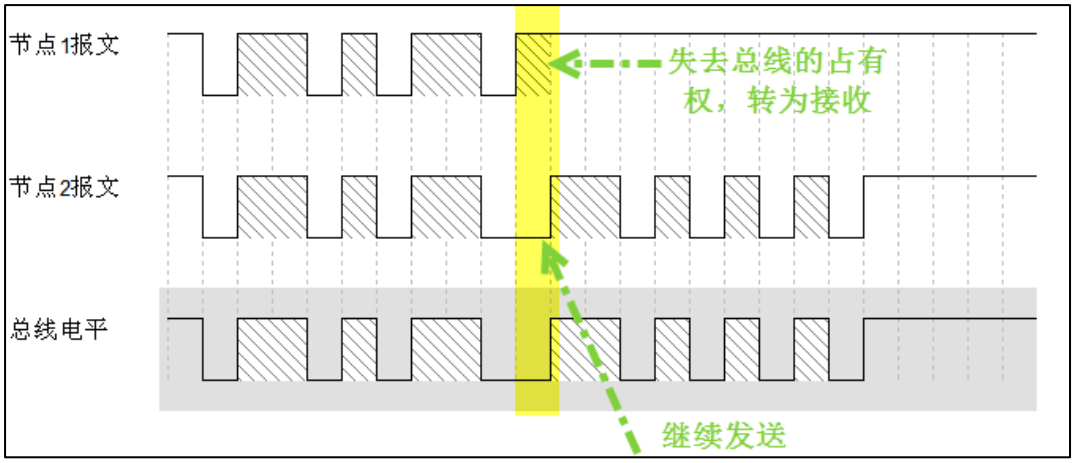

图 39-10 仲裁过程

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。

回看图 39-9 中的数据帧格式，可看到仲裁段除了报文 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。

❑ 控制段

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。

❑ CRC 段

为了保证报文的正确传输，CAN 的报文包含了一段 15 位的 CRC 校验码，一旦接收节点算出的 CRC 码跟接收到的 CRC 码不同，则它会向发送节点反馈出错信息，利用错误帧请求它重新发送。CRC 部分的计算一般由 CAN 控制器硬件完成，出错时的处理则由软件控制最大重发数。

在 CRC 校验码之后，有一个 CRC 界定符，它为隐性位，主要作用是把 CRC 校验码与后面的 ACK 段间隔起来。

- ACK 段

ACK 段包括一个 ACK 槽位，和 ACK 界定符位。类似 I2C 总线，在 ACK 槽位中，发送节点发送的是隐性位，而接收节点则在这一位中发送显性位以示应答。在 ACK 槽和帧结束之间由 ACK 界定符间隔开。

- 帧结束

EOF 段(End Of Frame)，译为帧结束，帧结束段由发送节点发送的 7 个隐性位表示结束。

其它报文的结构

关于其它的 CAN 报文结构，不再展开讲解，其主要内容见图 39-11。

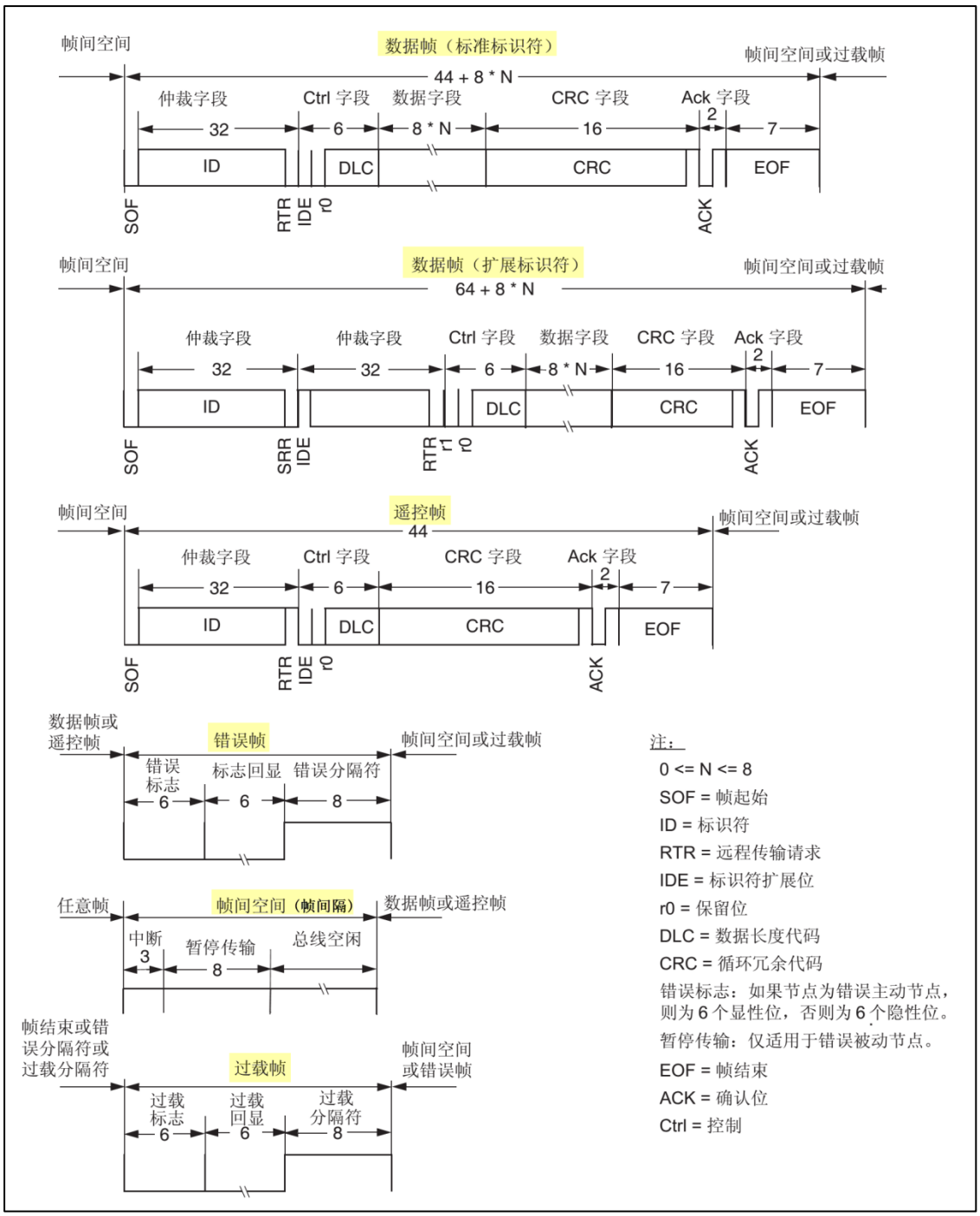

图 39-11 各种 CAN 报文的结构

39.2 STM32 的 CAN 外设简介

STM32 的芯片中具有 bxCAN 控制器 (Basic Extended CAN)，它支持 CAN 协议 2.0A 和 2.0B 标准。

该 CAN 控制器支持最高的通讯速率为 1Mb/s；可以自动地接收和发送 CAN 报文，支持使用标准 ID 和扩展 ID 的报文；外设中具有 3 个发送邮箱，发送报文的优先级可以使用

软件控制，还可以记录发送的时间；具有 2 个 3 级深度的接收 FIFO，可使用过滤功能只接收或不接收某些 ID 号的报文；可配置成自动重发；不支持使用 DMA 进行数据收发。

39.2.1 STM32 的 CAN 架构剖析

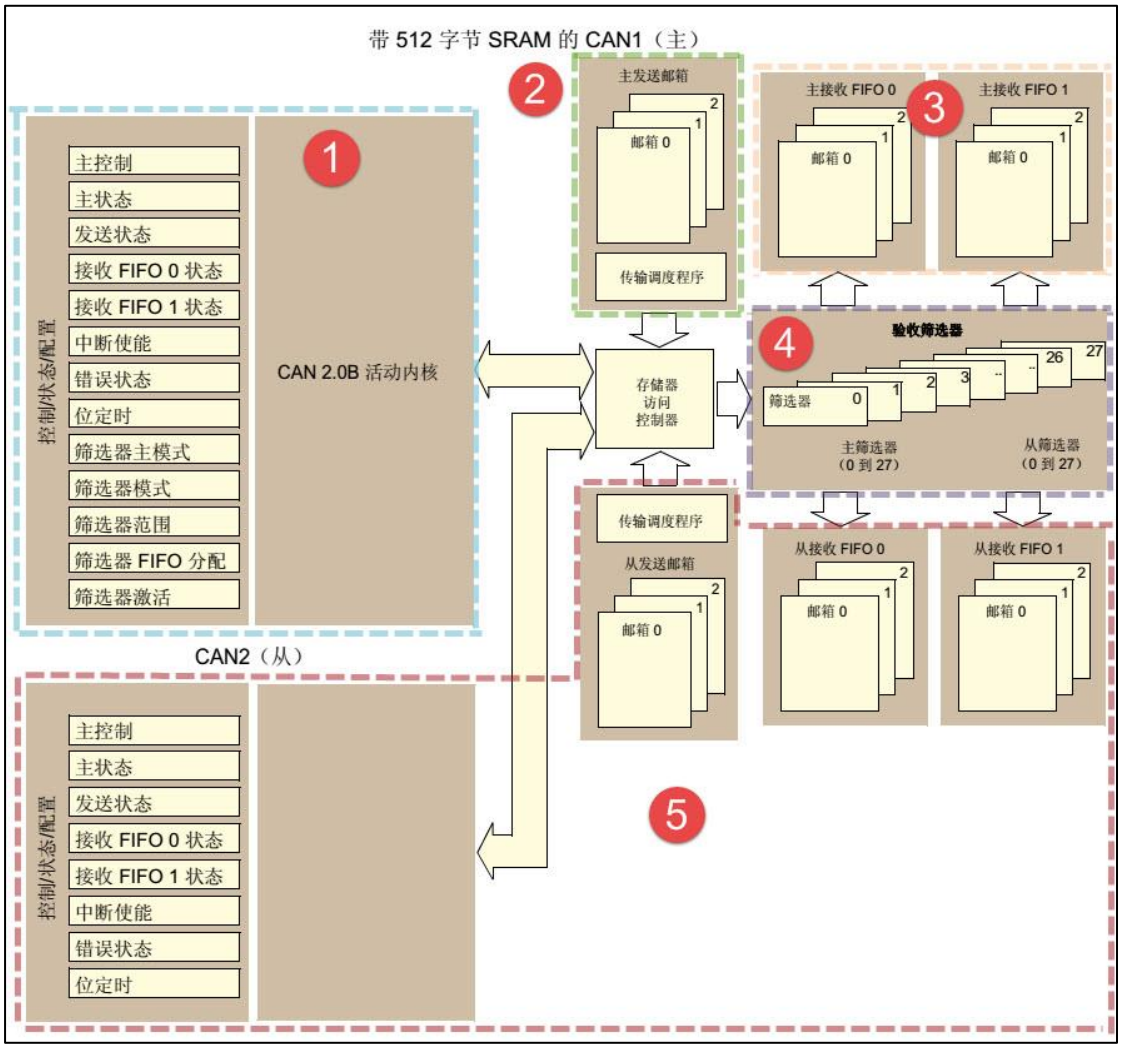

图 39-12 STM32 的 CAN 外设架构图（互联型产品）

图 39-12 是 STM32F105/107 系列互联型芯片的 CAN 外设架构图，图里具有 2 组 CAN 控制器，其中 CAN1 是主设备，框图中的“存储访问控制器”是由 CAN1 控制的，CAN2 无法直接访问存储区域，所以使用 CAN2 的时候必须使能 CAN1 外设的时钟。框图中主要包含 CAN 控制内核、发送邮箱、接收 FIFO 以及验收筛选器。我们实验板中使用的 STM32F103 系列芯片跟上述框图类似，但该系列只包含 1 组 CAN 控制器，即它们不包含图中标号⑤的部分。

下面对框图中的各个部分进行介绍。

1. CAN 控制内核

框图中标号①处的 CAN 控制内核包含了各种控制寄存器及状态寄存器，我们主要讲解其中的主控制寄存器 CAN_MCR 及位时序寄存器 CAN_BTR。

主控制寄存器 CAN_MCR

主控制寄存器 CAN_MCR 负责管理 CAN 的工作模式，它使用以下寄存器位实现控制。

(1) DBF 调试冻结功能

DBF(Debug freeze)调试冻结，使用它可设置 CAN 处于工作状态或禁止收发的状态，禁止收发时仍可访问接收 FIFO 中的数据。这两种状态是当 STM32 芯片处于程序调试模式时才使用的，平时使用并不影响。

(2) TTCM 时间触发模式

TTCM(Time triggered communication mode)时间触发模式，它用于配置 CAN 的时间触发通信模式，在此模式下，CAN 使用它内部定时器产生时间戳，并把它保存在 CAN_RDTxR、CAN_TDTxR 寄存器中。内部定时器在每个 CAN 位时间累加，在接收和发送的帧起始位被采样，并生成时间戳。利用它可以实现 ISO 11898-4 CAN 标准的分时同步通信功能。

(3) ABOM 自动离线管理

ABOM(Automatic bus-off management) 自动离线管理，它用于设置是否使用自动离线管理功能。当节点检测到它发送错误或接收错误超过一定值时，会自动进入离线状态，在离线状态中，CAN 不能接收或发送报文。处于离线状态的时候，可以软件控制恢复或者直接使用这个自动离线管理功能，它会在适当的时候自动恢复。

(4) AWUM 自动唤醒

AWUM(Automatic bus-off management)，自动唤醒功能，CAN 外设可以使用软件进入低功耗的睡眠模式，如果使能了这个自动唤醒功能，当 CAN 检测到总线活动的时候，会自动唤醒。

(5) NART 自动重传

NART(No automatic retransmission)报文自动重传功能，设置这个功能后，当报文发送失败时会自动重传至成功为止。若不使用这个功能，无论发送结果如何，消息只发送一次。

(6) RFLM 锁定模式

RFLM(Receive FIFO locked mode)FIFO 锁定模式，该功能用于锁定接收 FIFO。锁定后，当接收 FIFO 溢出时，会丢弃下一个接收的报文。若不锁定，则下一个接收到的报文会覆盖原报文。

(7) TXFP 报文发送优先级的判定方法

TXFP(Transmit FIFO priority)报文发送优先级的判定方法，当 CAN 外设的发送邮箱中有多个待发送报文时，本功能可以控制它是根据报文的 ID 优先级还是报文存进邮箱的顺序来发送。

位时序寄存器(CAN_BTR)及波特率

CAN 外设中的位时序寄存器 CAN_BTR 用于配置测试模式、波特率以及各种位内的段参数。

(1) 测试模式

为方便调试，STM32 的 CAN 提供了测试模式，配置位时序寄存器 CAN_BTR 的 SILM 及 LBKM 寄存器位可以控制使用正常模式、静默模式、回环模式及静默回环模式，见图 39-13。

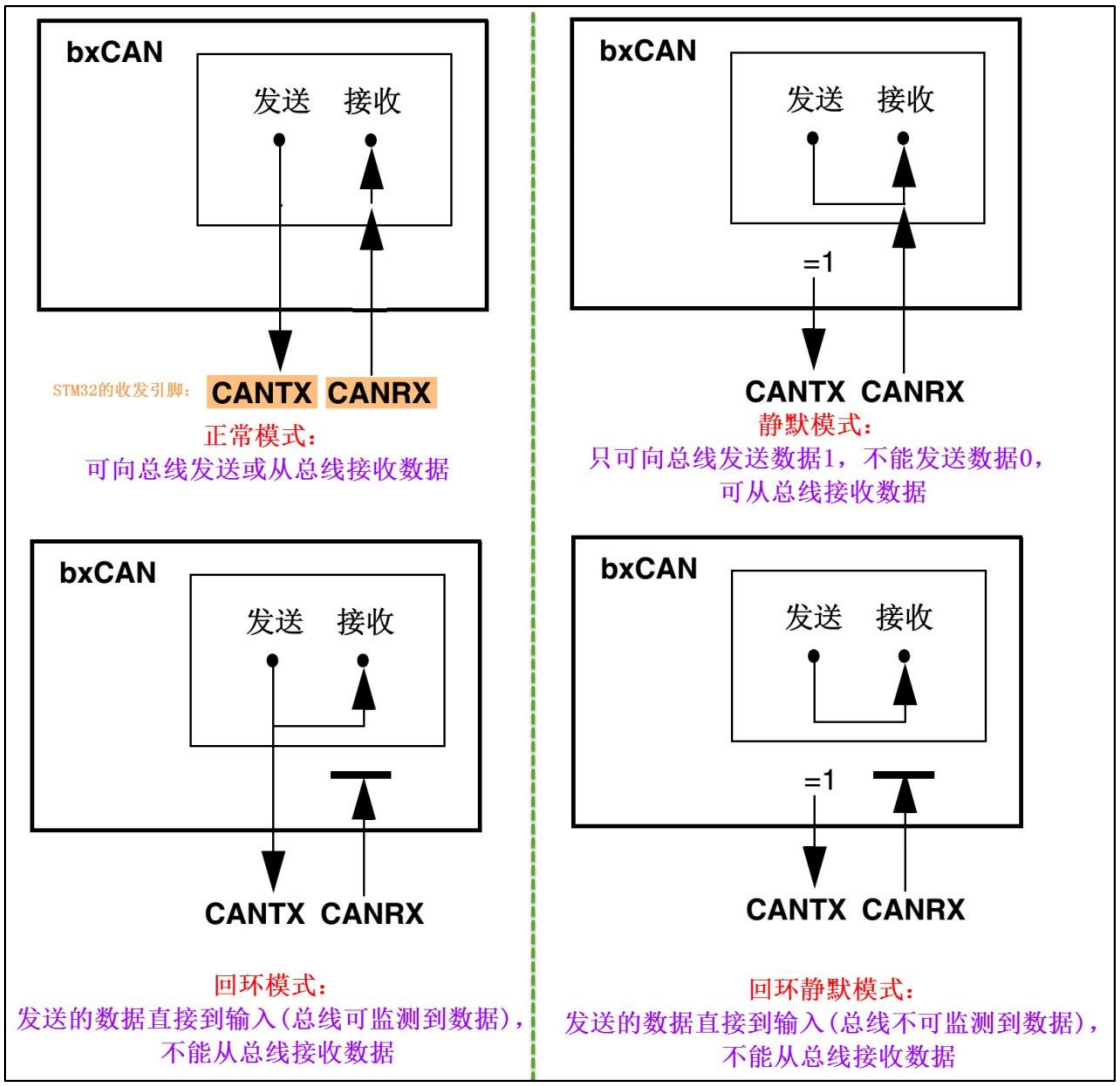

图 39-13 四种工作模式

各个工作模式介绍如下：

- 正常模式

  正常模式下就是一个正常的 CAN 节点，可以向总线发送数据和接收数据。

- 静默模式

  静默模式下，它自己的输出端的逻辑 0 数据会直接传输到它自己的输入端，逻辑 1 可以被发送到总线，所以它不能向总线发送显性位(逻辑 0)，只能发送隐性位(逻辑 1)。输入端可以从总线接收内容。由于它只可发送的隐性位不会强制影响总线的状态，所以把它称为静默模式。这种模式一般用于监测，它可以用于分析总线上的流量，但又不会因为发送显性位而影响总线。

- 回环模式

回环模式下，它自己的输出端的所有内容都直接传输到自己的输入端，输出端的内容同时也会被传输到总线上，即也可使用总线监测它的发送内容。输入端只接收自己发送端的内容，不接收来自总线上的内容。使用回环模式可以进行自检。

- 回环静默模式

回环静默模式是以上两种模式的结合，自己的输出端的所有内容都直接传输到自己的输入端，并且不会向总线发送显性位影响总线，不能通过总线监测它的发送内容。输入端只接收自己发送端的内容，不接收来自总线上的内容。这种方式可以在“热自检”时使用，即自我检查的时候，不会干扰总线。

以上说的各个模式，是不需要修改硬件接线的，例如，当输出直接连输入时，它是在 STM32 芯片内部连接的，传输路径不经过 STM32 的 CAN_Tx/Rx 引脚，更不经过外部连接的 CAN 收发器，只有输出数据到总线或从总线接收的情况下才会经过 CAN_Tx/Rx 引脚和收发器。

(2) 位时序及波特率

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，见图 39-14。

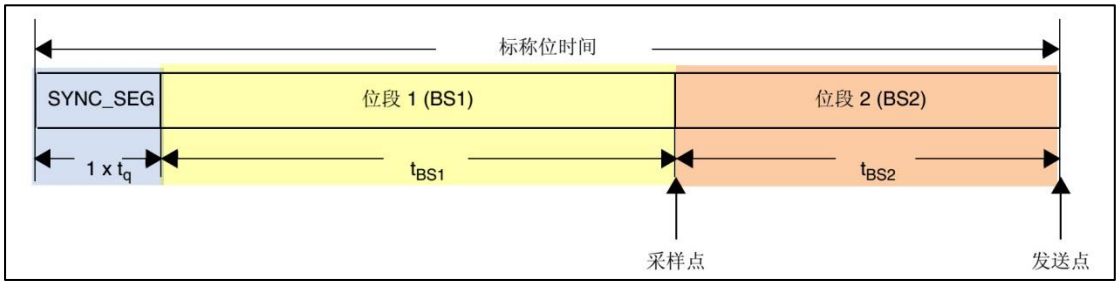

图 39-14 STM32 中 CAN 的位时序

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。

理解 STM32 的 CAN 外设的位时序时，可以把它的 BS1 段理解为是由前面介绍的 CAN 标准协议中 PTS 段与 PBS1 段合在一起的，而 BS2 段就相当于 PBS2 段。

了解位时序后，我们就可以配置波特率了。通过配置位时序寄存器 CAN_BTR 的 TS1[3:0]及 TS2[2:0]寄存器位设定 BS1 及 BS2 段的长度后，我们就可以确定每个 CAN 数据位的时间：

BS1 段时间：

$$T_{S1}=Tq \times (TS1[3:0] + 1),$$

BS2 段时间：

$$T_{S2}= Tq \times (TS2[2:0] + 1),$$

一个数据位的时间：

$$T_{1bit} =1Tq+T_{S1}+T_{S2} =1+ (TS1[3:0] + 1)+ (TS2[2:0] + 1)= N\ Tq$$

其中单个时间片的长度 Tq 与 CAN 外设的所挂载的时钟总线及分频器配置有关，CAN1 和 CAN2 外设都是挂载在 APB1 总线上的，而位时序寄存器 CAN_BTR 中的 BRP[9:0]寄存器位可以设置 CAN 外设时钟的分频值，所以：

$$Tq = (BRP[9:0]+1) \times T_{PCLK}$$

其中的 PCLK 指 APB1 时钟，默认值为 36MHz。

最终可以计算出 CAN 通讯的波特率：

$$BaudRate = 1/N\ Tq$$

例如表 39-3 说明了一种把波特率配置为 1Mbps 的方式。

表 39-3 一种配置波特率为 1Mbps 的方式

参数	说明
SYNC_SE 段	固定为 1Tq
BS1 段	设置为 5Tq (实际写入 TS1[3:0]的值为 4)
BS2 段	设置为 3Tq (实际写入 TS2[2:0]的值为 2)
T_{PCLK}	APB1 按默认配置为 F=36MHz，T_{PCLK}=1/36M
CAN 外设时钟分频	设置为 4 分频(实际写入 BRP[9:0]的值为 3)
1Tq 时间长度	$Tq = (BRP[9:0]+1) \times T_{PCLK}$ = 4 x 1/36M=1/9M
1 位的时间长度	$T_{1bit} = 1Tq+T_{S1}+T_{S2}$ = 1+5+3 = 9Tq
波特率	BaudRate = 1/N Tq = 1/(1/9M x 9)=1Mbps

2. CAN 发送邮箱

回到图 25-5 中的 CAN 外设框图，在标号②处的是 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 39-5。

表 39-4 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。

3. CAN 接收 FIFO

图 39-12 中的 CAN 外设框图，在标号③处的是 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 39-5。

表 39-5 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

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。

4. 验收筛选器

图 39-12 中的 CAN 外设框图，在标号④处的是 CAN 外设的验收筛选器，一共有 28 个筛选器组，每个筛选器组有 2 个寄存器，CAN1 和 CAN2 共用的筛选器的。其中 STM32F103 系列芯片仅有 14 个筛选器组：0-13 号。

在 CAN 协议中，消息的标识符与节点地址无关，但与消息内容有关。因此，发送节点将报文广播给所有接收器时，接收节点会根据报文标识符的值来确定软件是否需要该消息，为了简化软件的工作，STM32 的 CAN 外设接收报文前会先使用验收筛选器检查，只接收需要的报文到 FIFO 中。

筛选器工作的时候，可以调整筛选 ID 的长度及过滤模式。根据筛选 ID 长度来分类有有以下两种：

(1) 检查 STDID[10:0]、EXTID[17:0]、IDE 和 RTR 位，一共 31 位。

(2) 检查 STDID[10:0]、RTR、IDE 和 EXTID[17:15]，一共 16 位。

通过配置筛选尺度寄存器 CAN_FS1R 的 FSCx 位可以设置筛选器工作在哪个尺度。

而根据过滤的方法分为以下两种模式：

(1) 标识符列表模式，它把要接收报文的 ID 列成一个表，要求报文 ID 与列表中的某一个标识符完全相同才可以接收，可以理解为白名单管理。

(2) 掩码模式，它把可接收报文 ID 的某几位作为列表，这几位被称为掩码，可以把它理解成关键字搜索，只要掩码(关键字)相同，就符合要求，报文就会被保存到接收 FIFO。

通过配置筛选模式寄存器 CAN_FM1R 的 FBMx 位可以设置筛选器工作在哪个模式。不同的尺度和不同的过滤方法可使筛选器工作在图 39-15 的 4 种状态。

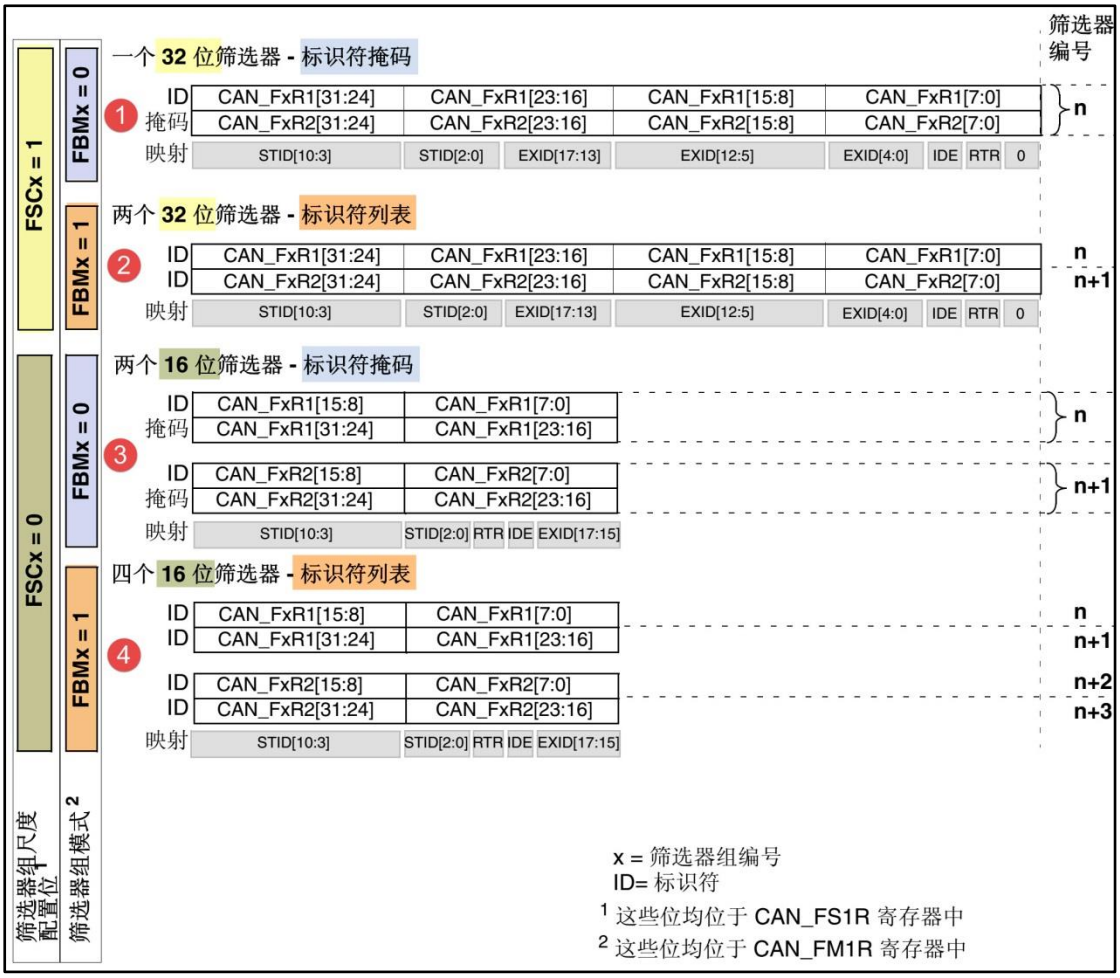

图 39-15 筛选器的 4 种工作状态

每组筛选器包含 2 个 32 位的寄存器，分别为 CAN_FxR1 和 CAN_FxR2，它们用来存储要筛选的 ID 或掩码，各个寄存器位代表的意义与图中两个寄存器下面“映射”的一栏一致，各个模式的说明见表 39-6。

表 39-6 筛选器的工作状态说明

模式	说明
32 位掩码模式	CAN_FxR1 存储 ID，CAN_FxR2 存储哪个位必须要与 CAN_FxR1 中的 ID 一致，2 个寄存器表示 1 组掩码。
32 位标识符模式	CAN_FxR1 和 CAN_FxR2 各存储 1 个 ID，2 个寄存器表示 2 个筛选的 ID
16 位掩码模式	CAN_FxR1 高 16 位存储 ID，低 16 位存储哪个位必须要与高 16 位的 ID 一致； CAN_FxR2 高 16 位存储 ID，低 16 位存储哪个位必须要与高 16 位的 ID 一致 2 个寄存器表示 2 组掩码。
16 位标识符模式	CAN_FxR1 和 CAN_FxR2 各存储 2 个 ID，2 个寄存器表示 4 个筛选的 ID

例如下面的表格所示，在掩码模式时，第一个寄存器存储要筛选的 ID，第二个寄存器存储掩码，掩码为 1 的部分表示该位必须与 ID 中的内容一致，筛选的结果为表中第三行的 ID 值，它是一组包含多个的 ID 值，其中 x 表示该位可以为 1 可以为 0。

ID	1	0	1	1	1	0	1	…
掩码	1	1	1	0	0	1	0	…
筛选的 ID	1	0	1	x	x	0	x	…

而工作在标识符模式时，2 个寄存器存储的都是要筛选的 ID，它只包含 2 个要筛选的 ID 值(32 位模式时)。

如果使能了筛选器，且报文的 ID 与所有筛选器的配置都不匹配，CAN 外设会丢弃该报文，不存入接收 FIFO。

5. 整体控制逻辑

回到图 39-12 结构框图，图中的标号⑤处表示的是 CAN2 外设的结构，它与 CAN1 外设是一样的，他们共用筛选器且由于存储访问控制器由 CAN1 控制，所以要使用 CAN2 的时候必须要使能 CAN1 的时钟。其中 STM32F103 系列芯片不具有 CAN2 控制器。

39.3 CAN 初始化结构体

从 STM32 的 CAN 外设我们了解到它的功能非常多，控制涉及的寄存器也非常丰富，而使用 STM32 标准库提供的各种结构体及库函数可以简化这些控制过程。跟其它外设一样，STM32 标准库提供了 CAN 初始化结构体及初始化函数来控制 CAN 的工作方式，提供了收发报文使用的结构体及收发函数，还有配置控制筛选器模式及 ID 的结构体。这些内容都定义在库文件“stm32f10x_can.h”及“stm32f10x_can.c”中，编程时我们可以结合这两个文件内的注释使用或参考库帮助文档。

首先我们来学习初始化结构体的内容，见代码清单 39-1。

代码清单 39-1 CAN 初始化结构体

```
/**
  * @brief  CAN 初始化结构体
  */
typedef struct {
    uint16_t CAN_Prescaler;   /*配置 CAN 外设的时钟分频，可设置为 1-1024*/
    uint8_t CAN_Mode;         /*配置 CAN 的工作模式，回环或正常模式*/
    uint8_t CAN_SJW;          /*配置 SJW 极限值 */
    uint8_t CAN_BS1;          /*配置 BS1 段长度*/
    uint8_t CAN_BS2;          /*配置 BS2 段长度 */
    FunctionalState CAN_TTCM; /*是否使能 TTCM 时间触发功能*/
    FunctionalState CAN_ABOM;  /*是否使能 ABOM 自动离线管理功能*/
    FunctionalState CAN_AWUM;  /*是否使能 AWUM 自动唤醒功能 */
    FunctionalState CAN_NART;  /*是否使能 NART 自动重传功能*/
    FunctionalState CAN_RFLM;  /*是否使能 RFLM 锁定 FIFO 功能*/
    FunctionalState CAN_TXFP;  /*配置 TXFP 报文优先级的判定方法*/
} CAN_InitTypeDef;
```

这些结构体成员说明如下，其中括号内的文字是对应参数在 STM32 标准库中定义的宏，这些结构体成员都是“39.2.1 1CAN 控制内核”小节介绍的内容，可对比阅读：

(1) CAN_Prescaler

本成员设置 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减 1 后再写入 BRP 寄存器位，即前面介绍的 Tq 计算公式：

$$Tq = (BRP[9:0]+1) \times T_{PCLK}$$

等效于：$Tq = CAN\_Prescaler \times T_{PCLK}$

(2) CAN_Mode

本成员设置 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)。

(3) CAN_SJW

本成员可以配置 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)。

(4) CAN_BS1

本成员用于设置 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)。

(5) CAN_BS2

本成员用于设置 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)。

SYNC_SEG、BS1 段及 BS2 段的长度加起来即一个数据位的长度，即前面介绍的原来计算公式：

$$T_{1bit} = 1Tq + T_{S1} + T_{S2} = 1 + (TS1[3:0] + 1) + (TS2[2:0] + 1)$$

等效于：$T_{1bit} = 1Tq + CAN\_BS1 + CAN\_BS2$

(6) CAN_TTCM

本成员用于设置是否使用时间触发功能(ENABLE/DISABLE)，时间触发功能在某些 CAN 标准中会使用到。

(7) CAN_ABOM

本成员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。

(8) CAN_ AWUM

本成员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。

(9) CAN_ABOM

本成员用于设置是否使用自动离线管理功能(ENABLE/DISABLE)，使用自动离线管理可以在出错时离线后适时自动恢复，不需要软件干预。

(10) CAN_NART

本成员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。

(11) CAN_RFLM

本成员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。

(12) CAN_RFLM

本成员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。

配置完这些结构体成员后，我们调用库函数 CAN_Init 即可把这些参数写入到 CAN 控制寄存器中，实现 CAN 的初始化。

39.4 CAN 发送及接收结构体

在发送或接收报文时，需要往发送邮箱中写入报文信息或从接收 FIFO 中读取报文信息，利用 STM32 标准库的发送及接收结构体可以方便地完成这样的工作，它们的定义见代码清单 39-2。

代码清单 39-2 CAN 发送及接收结构体

```
/**
  * @brief  CAN Tx message structure definition
  * 发送结构体
  */
typedef struct {
    uint32_t StdId;  /*存储报文的标准标识符 11 位，0-0x7FF. */
    uint32_t ExtId;  /*存储报文的扩展标识符 29 位，0-0x1FFFFFFF. */
    uint8_t IDE;     /*存储 IDE 扩展标志 */
    uint8_t RTR;     /*存储 RTR 远程帧标志*/
    uint8_t DLC;     /*存储报文数据段的长度，0-8 */
    uint8_t Data[8]; /*存储报文数据段的内容 */
} CanTxMsg;

/**
  * @brief  CAN Rx message structure definition
  * 接收结构体
  */
typedef struct {
    uint32_t StdId;  /*存储了报文的标准标识符 11 位，0-0x7FF. */
    uint32_t ExtId;  /*存储了报文的扩展标识符 29 位，0-0x1FFFFFFF. */
    uint8_t IDE;     /*存储了 IDE 扩展标志 */
    uint8_t RTR;     /*存储了 RTR 远程帧标志*/
    uint8_t DLC;     /*存储了报文数据段的长度，0-8 */
    uint8_t Data[8]; /*存储了报文数据段的内容 */
    uint8_t FMI;     /*存储了 本报文是由经过筛选器存储进 FIFO 的，0-0xFF */
} CanRxMsg;
```

这些结构体成员都是“39.2.1 2CAN 发送邮箱及 CAN 接收 FIFO”小节介绍的内容，可对比阅读，发送结构体与接收结构体是类似的，只是接收结构体多了一个 FMI 成员，说明如下：

(1) StdId

本成员存储的是报文的 11 位标准标识符，范围是 0-0x7FF。

(2) ExtId

本成员存储的是报文的 29 位扩展标识符，范围是 0-0x1FFFFFFF。ExtId 与 StdId 这两个成员根据下面的 IDE 位配置，只有一个是有效的。

(3) IDE

本成员存储的是扩展标志 IDE 位，当它的值为宏 CAN_ID_STD 时表示本报文是标准帧，使用 StdId 成员存储报文 ID；当它的值为宏 CAN_ID_EXT 时表示本报文是扩展帧，使用 ExtId 成员存储报文 ID。

(4) RTR

本成员存储的是报文类型标志 RTR 位，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。

(5) DLC

本成员存储的是数据帧数据段的长度，它的值的范围是 0-8，当报文是遥控帧时 DLC 值为 0。

(6) Data[8]

本成员存储的就是数据帧中数据段的数据。

(7) FMI

本成员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。

当需要使用 CAN 发送报文时，先定义一个上面发送类型的结构体，然后把报文的内容按成员赋值到该结构体中，最后调用库函数 CAN_Transmit 把这些内容写入到发送邮箱即可把报文发送出去。

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。

39.5 CAN 筛选器结构体

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，它的定义见代码清单 39-3。

代码清单 39-3 CAN 筛选器结构体

```
/**
  * @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;
```

这些结构体成员都是“39.2.1 4 验收筛选器”小节介绍的内容，可对比阅读，各个结构体成员的介绍如下：

(1) CAN_FilterIdHigh

CAN_FilterIdHigh 成员用于存储要筛选的 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。

(2) CAN_FilterIdLow

类似地，CAN_FilterIdLow 成员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。

(3) CAN_FilterMaskIdHigh

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。

(4) CAN_FilterMaskIdLow

类似地，CAN_FilterMaskIdLow 存储的内容也分两种情况，当筛选器工作在标识符列表模式时，它的功能与 CAN_FilterIdLow 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_FilterIdLow 成员对应的掩码，与 CAN_FilterIdLow 组成一组筛选器。

上面四个结构体的存储的内容很容易让人糊涂，请结合前面的图 39-14 和下面的表 39-7 理解，如果还搞不清楚，再结合库函数 CAN_FilterInit 的源码来分析。

表 39-7 不同模式下各结构体成员的内容

模式	CAN_FilterIdHigh	CAN_FilterIdLow	CAN_FilterMaskIdHigh	CAN_FilterMaskIdLow
32 位列表模式	ID1 的高 16 位	ID1 的低 16 位	ID2 的高 16 位	ID2 的低 16 位
16 位列表模式	ID1 的完整数值	ID2 的完整数值	ID3 的完整数值	ID4 的完整数值
32 位掩码模式	ID1 的高 16 位	ID1 的低 16 位	ID1 掩码的高 16 位	ID1 掩码的低 16 位
16 位掩码模式	ID1 的完整数值	ID2 的完整数值	ID1 掩码的完整数值	ID2 掩码完整数值

对这些结构体成员赋值的时候，还要注意寄存器位的映射，即注意哪部分代表 STID，哪部分代表 EXID 以及 IDE、RTR 位。

(5) CAN_FilterFIFOAssignment

本成员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 或 FIFO1(宏 CAN_Filter_FIFO0/1)。

(6) CAN_FilterNumber

本成员用于设置筛选器的编号，即本过滤器结构体配置的是哪一组筛选器，CAN 一共有 28 个筛选器，所以它的可输入参数范围为 0-27(STM32F103 系列芯片的可输入参数为 0-13)。

(7) CAN_FilterMode

本成员用于设置筛选器的工作模式，可以设置为列表模式(宏 CAN_FilterMode_IdList)及掩码模式(宏 CAN_FilterMode_IdMask)。

(8) CAN_FilterScale

本成员用于设置筛选器的尺度，可以设置为 32 位长(宏 CAN_FilterScale_32bit)及 16 位长(宏 CAN_FilterScale_16bit)。

(9) CAN_FilterActivation

本成员用于设置是否激活这个筛选器(宏 ENABLE/DISABLE)。

配置完这些结构体成员后，我们调用库函数 CAN_FilterInit 即可把这些参数写入到筛选控制寄存器中，从而使用筛选器。我们前面说如果不理解那几个 ID 结构体成员存储的内容时，可以直接阅读库函数 CAN_FilterInit 的源代码理解，就是因为它直接对寄存器写入内容，代码的逻辑是非常清晰的。

39.6 CAN—双机通讯实验

本小节演示如何使用 STM32 的 CAN 外设实现两个设备之间的通讯，该实验中使用了两个实验板，如果您只有一个实验板，也可以使用 CAN 的回环模式进行测试，不影响学习的。为此，我们提供了“CAN—双机通讯”及“CAN—回环测试”两个工程，可根据自己的实验环境选择相应的工程来学习。这两个工程的主体都是一样的，本教程主要以“CAN—双机通讯”工程进行讲解。

39.6.1 硬件设计

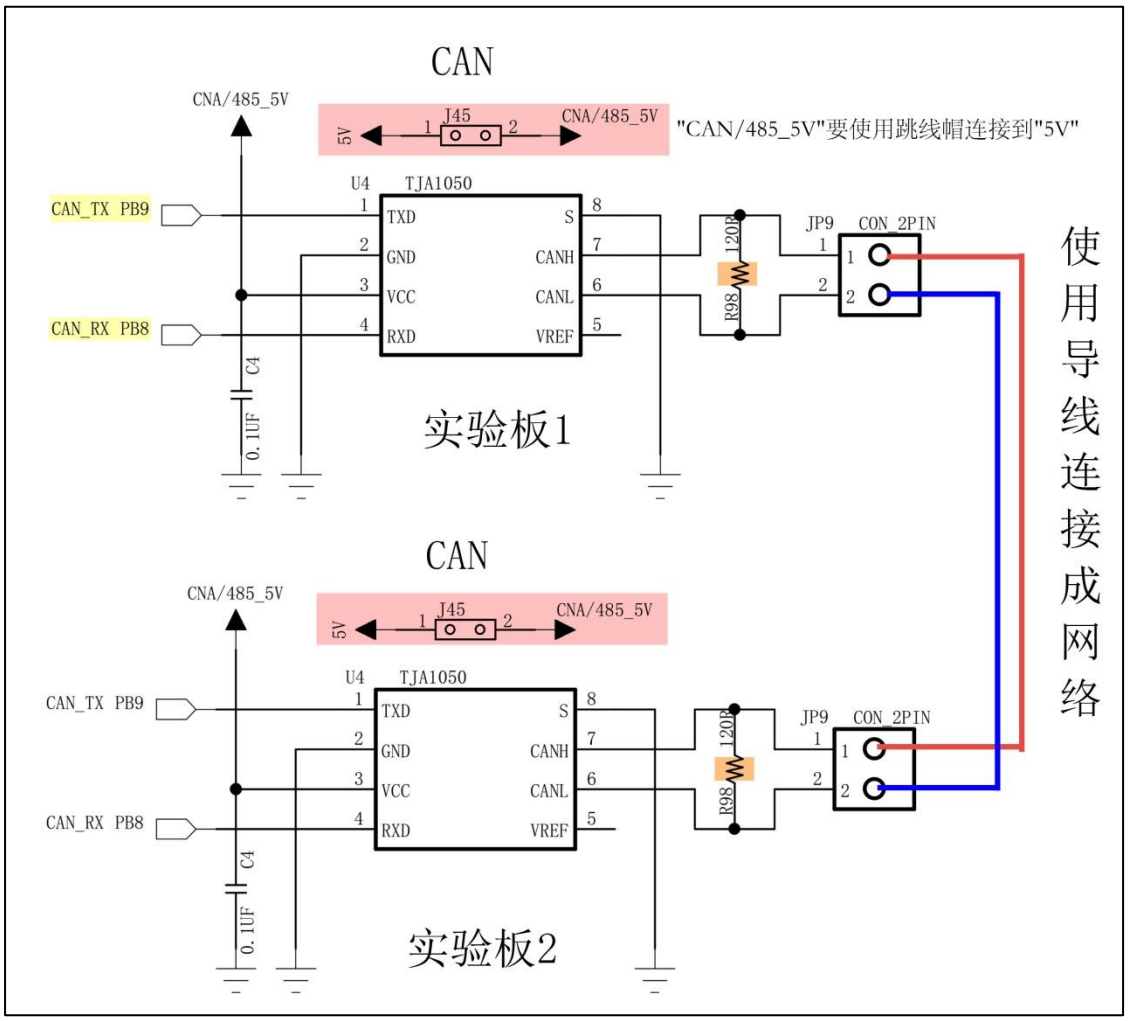

图 39-16 双 CAN 通讯实验硬件连接图

图 39-16 中的是两个实验板的硬件连接。在单个实验板中，作为 CAN 控制器的 STM32 引出 CAN_Tx 和 CAN_Rx 两个引脚与 CAN 收发器 TJA1050 相连，收发器使用 CANH 及 CANL 引脚连接到 CAN 总线网络中。为了方便使用，我们每个实验板引出的 CANH 及 CANL 都连接了 1 个 120 欧的电阻作为 CAN 总线的端电阻，所以要注意如果您要把实验板作为一个普通节点连接到现有的 CAN 总线时，是不应添加该电阻的！

要实现通讯，我们还要使用导线把实验板引出的 CANH 及 CANL 两条总线连接起来，才能构成完整的网络。实验板之间 CANH1 与 CANH2 连接，CANL1 与 CANL2 连接即可。

要注意的是，由于我们的实验板 CAN 使用的信号线与摄像头共用了，为防止干扰，平时我们默认是不给 CAN 收发器供电的，使用 CAN 的时候一定要把 CAN 接线端子旁边的“C/4-5V”排针使用跳线帽与“5V”排针连接起来进行供电，并且把摄像头从板子上拔下来。

如果您使用的是单机回环测试的工程实验，就不需要使用导线连接板子了，而且也不需要给收发器供电，因为回环模式的信号是不经过收发器的。

39.6.2 软件设计

为了使工程更加有条理，我们把 CAN 控制器相关的代码独立分开存储，方便以后移植。在“串口实验”之上新建“bsp_can.c”及“bsp_can.h”文件，这些文件也可根据您的喜好命名，它们不属于 STM32 标准库的内容，是由我们自己根据应用需要编写的。

1. 编程要点

(1) 初始化 CAN 通讯使用的目标引脚及端口时钟；
(2) 使能 CAN 外设的时钟；
(3) 配置 CAN 外设的工作模式、位时序以及波特率；
(4) 配置筛选器的工作方式；
(5) 编写测试程序，收发报文并校验。

2. 代码分析

CAN 硬件相关宏定义

我们把 CAN 硬件相关的配置都以宏的形式定义到 “bsp_can.h”文件中，见代码清单 39-4。

代码清单 39-4 CAN 硬件配置相关的宏(bsp_can.h 文件)

```
1  #define CANx                       CAN1
2  #define CAN_CLK                    RCC_APB1Periph_CAN1
3  #define CAN_RX_IRQ                 USB_LP_CAN1_RX0_IRQn
4  #define CAN_RX_IRQHandler          USB_LP_CAN1_RX0_IRQHandler
5
6  #define CAN_RX_PIN                 GPIO_Pin_8
7  #define CAN_TX_PIN                 GPIO_Pin_9
8  #define CAN_TX_GPIO_PORT           GPIOB
9  #define CAN_RX_GPIO_PORT           GPIOB
10 #define CAN_TX_GPIO_CLK          (RCC_APB2Periph_AFIO|RCC_APB2Periph_GPIOB)
11 #define CAN_RX_GPIO_CLK            RCC_APB2Periph_GPIOB
```

以上代码根据硬件连接，把与 CAN 通讯使用的 CAN 号 、引脚号以及时钟都以宏封装起来，并且定义了接收中断的中断向量和中断服务函数，我们通过中断来获知接收 FIFO 的信息。注意在 GPIO 时钟部分我们还加入了 AFIO 时钟，这是为下面 CAN 进行复用功能重映射而设置的，当使用复用功能重映射时，必须开启 AFIO 时钟。

初始化 CAN 的 GPIO

利用上面的宏，编写 CAN 的初始化函数，见代码清单 25-3。

代码清单 39-5 CAN 的 GPIO 初始化函数(bsp_can.c 文件)

```
1 /*
2  * 函数名：CAN_GPIO_Config
3  * 描述  ：CAN 的 GPIO 配置
4  * 输入  ：无
5  * 输出  ：无
6  * 调用  ：内部调用
7  */
8 static void CAN_GPIO_Config(void)
9 {
```

```
10   GPIO_InitTypeDef GPIO_InitStructure;
11
12   /* Enable GPIO clock */
13   RCC_APB2PeriphClockCmd(CAN_TX_GPIO_CLK|CAN_RX_GPIO_CLK, ENABLE);
14   //重映射引脚
15   GPIO_PinRemapConfig(GPIO_Remap1_CAN1, ENABLE);
16
17    /* Configure CAN TX pins */
18   GPIO_InitStructure.GPIO_Pin = CAN_TX_PIN;
19   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;              // 复用推挽输
出
20   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
21
22   GPIO_Init(CAN_TX_GPIO_PORT, &GPIO_InitStructure);
23
24   /* Configure CAN RX  pins */
25   GPIO_InitStructure.GPIO_Pin = CAN_RX_PIN ;
26   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;               // 上拉输入
27   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
28
29   GPIO_Init(CAN_RX_GPIO_PORT, &GPIO_InitStructure);

30 }
```

与所有使用到 GPIO 的外设一样，都要先把使用到的 GPIO 引脚模式初始化，配置好复用功能，CAN 的两个引脚都配置成通用推挽输出模式即可。根据 CAN 外设的要求，TX 引脚要被配置成推挽复用输出、RX 引脚要被设置成浮空输入或上拉输入。另外，由于本实验板中的 CAN 使用的 PB8、PB9 的默认复用功能不是 CAN 外设的引脚，需要使用外设引脚的复用功能重映射才能正常使用，代码中使用 GPIO_PinRemapConfig 函数把 CAN 映射至 PB8、PB9，见表 39-8。

表 39-8 CAN 引脚的复用功能重映射

复用功能	CAN_REMAP[1:0]="00"	CAN_REMAP[1:0]="10"(1)	CAN_REMAP[1:0]="11"(2)
CAN_RX	PA11	PB8	PD0
CAN_TX	PA12	PB9	PD1

配置 CAN 的工作模式

接下来我们配置 CAN 的工作模式，由于我们是自己用的两个板子之间进行通讯，波特率之类的配置只要两个板子一致即可。如果您要使实验板与某个 CAN 总线网络的通讯的节点通讯，那么实验板的 CAN 配置必须要与该总线一致。我们实验中使用的配置见代码清单 39-6。

代码清单 39-6 配置 CAN 的工作模式(bsp_can.c 文件)

```
1 /*
2  * 函数名: CAN_Mode_Config
3  * 描述  : CAN 的模式 配置
4  * 输入  : 无
5  * 输出  : 无
6  * 调用  : 内部调用
7  */
8 static void CAN_Mode_Config(void)
9 {
10     CAN_InitTypeDef        CAN_InitStructure;
11     /************************CAN 通信参数设置**********************/
12     /* Enable CAN clock */
13     RCC_APB1PeriphClockCmd(CAN_CLK, ENABLE);
14
```

```
    /*CAN 寄存器初始化*/
    CAN_DeInit(CAN1);
    CAN_StructInit(&CAN_InitStructure);

    /*CAN 单元初始化*/
    CAN_InitStructure.CAN_TTCM=DISABLE;   //MCR-TTCM  关闭时间触发通信模式使能
    CAN_InitStructure.CAN_ABOM=ENABLE;          //MCR-ABOM  使能自动离线管理
    CAN_InitStructure.CAN_AWUM=ENABLE;          //MCR-AWUM  使用自动唤醒模式
    CAN_InitStructure.CAN_NART=DISABLE;         //MCR-NART  禁止报文自动重传
    CAN_InitStructure.CAN_RFLM=DISABLE;         //MCR-RFLM  接收 FIFO 不锁定
                                               // 溢出时新报文会覆盖原有报文
    CAN_InitStructure.CAN_TXFP=DISABLE; //MCR-TXFP  发送 FIFO 优先级 取决于报文标示符
    CAN_InitStructure.CAN_Mode = CAN_Mode_Normal;  //正常工作模式
    CAN_InitStructure.CAN_SJW=CAN_SJW_2tq; //BTR-SJW 重新同步跳跃宽度 2 个时间单元

    /* ss=1 bs1=5 bs2=3 位时间宽度为(1+5+3) 波特率即为时钟周期 tq*(1+3+5)  */
    CAN_InitStructure.CAN_BS1=CAN_BS1_5tq; //BTR-TS1 时间段 1 占用了 5 个时间单元
    CAN_InitStructure.CAN_BS2=CAN_BS2_3tq; //BTR-TS1 时间段 2 占用了 3 个时间单元

 /* CAN Baudrate = 1 MBps (1MBps 已为 stm32 的 CAN 最高速率) (CAN 时钟频率为 APB 1 = 36 MHz) */
    ////BTR-BRP 波特率分频器  定义了时间单元的时间长度 36/(1+5+3)/4=1 Mbps
    CAN_InitStructure.CAN_Prescaler =4;
    CAN_Init(CANx, &CAN_InitStructure);
}
```

这段代码主要是把 CAN 的模式设置成了正常工作模式，如果您阅读的是“CAN—回环测试”的工程，这里是被配置成回环模式的，除此之外，两个工程就没有其它差别了。

代码中还把位时序中的 BS1 和 BS2 段分别设置成了 5Tq 和 3Tq，再加上 SYNC_SEG 段，一个 CAN 数据位就是 9Tq 了，加上 CAN 外设的分频配置为 4 分频，CAN 所使用的总线时钟 f_{APB1} = 36MHz，于是我们可计算出它的波特率：

$$1Tq = 1/(36M/4)=1/9\ us$$

$$T_{1bit}=(5+3+1) \times Tq = 1us$$

$$波特率=1/T_{1bit} = 1Mbps$$

配置筛选器

以上是配置 CAN 的工作模式，为了方便管理接收报文，我们还要把筛选器用起来，见代码清单 39-7。

代码清单 39-7 配置 CAN 的筛选器(bsp_can.c 文件)

```

/*IDE 位的标志*/
#define CAN_ID_STD             ((uint32_t)0x00000000)  /*标准 ID */
#define CAN_ID_EXT             ((uint32_t)0x00000004)  /*扩展 ID */

/*RTR 位的标志*/
#define CAN_RTR_Data            ((uint32_t)0x00000000)  /*数据帧 */
#define CAN_RTR_Remote          ((uint32_t)0x00000002)  /*远程帧*/

/**************************************************************/
/*
 * 函数名：CAN_Filter_Config
 * 描述  ：CAN 的筛选器 配置
 * 输入  ：无
 * 输出  ：无
 * 调用  ：内部调用
 */
static void CAN_Filter_Config(void)
```

```
{
    CAN_FilterInitTypeDef  CAN_FilterInitStructure;

    /*CAN 筛选器初始化*/
    CAN_FilterInitStructure.CAN_FilterNumber=0; //筛选器组 0
    //工作在掩码模式
    CAN_FilterInitStructure.CAN_FilterMode=CAN_FilterMode_IdMask;
    //筛选器位宽为单个 32 位。
    CAN_FilterInitStructure.CAN_FilterScale=CAN_FilterScale_32bit;

    /* 使能筛选器，按照标志符的内容进行比对筛选，
        扩展 ID 不是如下的就抛弃掉，是的话，会存入 FIFO0。 */
//要筛选的 ID 高位，第 0 位保留，第 1 位为 RTR 标志，第 2 位为 IDE 标志，从第 3 位开始是 EXID
CAN_FilterInitStructure.CAN_FilterIdHigh= ((((u32)0x1314<<3)|CAN_ID_EXT|CAN_RTR_DATA)&0xFFFF0000)>>16;
    //要筛选的 ID 低位
    CAN_FilterInitStructure.CAN_FilterIdLow= (((u32)0x1314<<3)|CAN_ID_EXT|CAN_RTR_DATA)&0xFFFF;
    //筛选器高 16 位每位必须匹配
    CAN_FilterInitStructure.CAN_FilterMaskIdHigh= 0xFFFF;
    //筛选器低 16 位每位必须匹配
    CAN_FilterInitStructure.CAN_FilterMaskIdLow= 0xFFFF;
    //筛选器被关联到 FIFO0
    CAN_FilterInitStructure.CAN_FilterFIFOAssignment=CAN_Filter_FIFO0 ;
    //使能筛选器
    CAN_FilterInitStructure.CAN_FilterActivation=ENABLE;

    CAN_FilterInit(&CAN_FilterInitStructure);
    /*CAN 通信中断使能*/
    CAN_ITConfig(CANx, CAN_IT_FMP0, ENABLE);
}
```

这段代码把筛选器第 0 组配置成了 32 位的掩码模式，并且把它的输出连接到接收 FIFO0，若通过了筛选器的匹配，报文会被存储到接收 FIFO0。

筛选器配置的重点是配置 ID 和掩码，根据我们的配置，这个筛选器工作在图 39-17 中的模式。

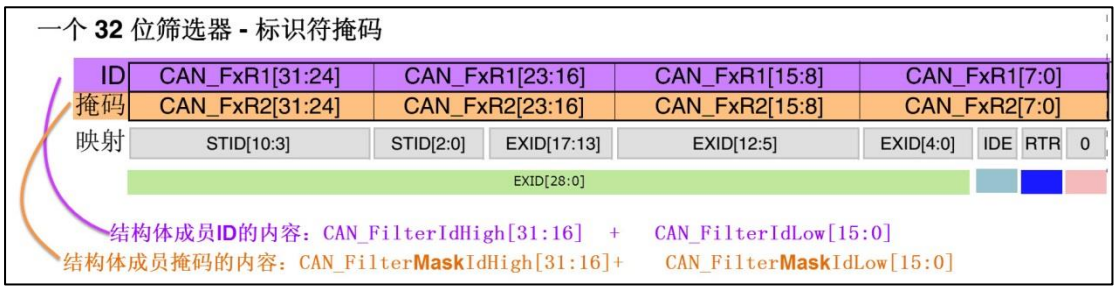

图 39-17 一个 32 位的掩码模式筛选器

在该配置中，结构体成员 CAN_FilterIdHigh 和 CAN_FilterIdLow 存储的是要筛选的 ID，而 CAN_FilterMaskIdHigh 和 CAN_FilterMaskIdLow 存储的是相应的掩码。在赋值时，要注意寄存器位的映射，在 32 位的 ID 中，第 0 位是保留位，第 1 位是 RTR 标志，第 2 位是 IDE 标志，从第 3 位起才是报文的 ID(扩展 ID)。

因此在上述代码中我们先把扩展 ID“0x1314”、IDE 位标志“宏 CAN_ID_EXT”以及 RTR 位标志“宏 CAN_RTR_DATA”根据寄存器位映射组成一个 32 位的数据，然后再把它的高 16 位和低 16 位分别赋值给结构体成员 CAN_FilterIdHigh 和 CAN_FilterIdLow。

而在掩码部分，为简单起见我们直接对所有位赋值为 1，表示上述所有标志都完全一样的报文才能经过筛选，所以我们这个配置相当于单个 ID 列表的模式，只筛选了一个 ID 号，而不是筛选一组 ID 号。这里只是为了演示方便，实际使用中一般会对不要求相等的数

据位赋值为 0，从而过滤一组 ID，如果有需要，还可以继续配置多个筛选器组，最多可以配置 28 个，代码中只是配置了筛选器组 0。

对结构体赋值完毕后调用库函数 CAN_FilterInit 把个筛选器组的参数写入到寄存器中。

配置接收中断

在配置筛选器代码的最后部分我们还调用库函数 CAN_ITConfig 使能了 CAN 的中断，该函数使用的输入参数宏 CAN_IT_FMP0 表示当 FIFO0 接收到数据时会引起中断，该接收中断的优先级配置如下，见代码清单 39-8。

代码清单 39-8 配置 CAN 接收中断的优先级(bsp_can.c 文件)

```
1  /*接收中断号*/
2  #define CAN_RX_IRQ                                      USB_LP_CAN1_RX0_IRQn
3  /*
4   * 函数名：CAN_NVIC_Config
5   * 描述   ：CAN 的 NVIC 配置,第 1 优先级组，0，0 优先级
6   * 输入   ：无
7   * 输出   ：无
8   * 调用   ：内部调用
9   */
10 static void CAN_NVIC_Config(void)
11 {
12     NVIC_InitTypeDef NVIC_InitStructure;
13     /* Configure one bit for preemption priority */
14     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
15     /*中断设置*/
16     NVIC_InitStructure.NVIC_IRQChannel = CAN_RX_IRQ;        //CAN RX 中断
17     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
18     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
19     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
20     NVIC_Init(&NVIC_InitStructure);
21 }
```

这部分与我们配置其它中断的优先级无异，都是配置 NVIC 结构体，优先级可根据自己的需要配置，最主要的是中断向量，上述代码中把中断向量配置成了 CAN 的接收中断。

设置发送报文

要使用 CAN 发送报文时，我们需要先定义一个发送报文结构体并向它赋值，见代码清单 39-9。

代码清单 39-9 设置要发送的报文(bsp_can.c 文件)

```
1  /*IDE 位的标志*/
2  #define CAN_ID_STD              ((uint32_t)0x00000000)  /*标准 ID */
3  #define CAN_ID_EXT              ((uint32_t)0x00000004)  /*扩展 ID */
4
5  /*RTR 位的标志*/
6  #define CAN_RTR_Data             ((uint32_t)0x00000000)  /*数据帧 */
7  #define CAN_RTR_Remote           ((uint32_t)0x00000002)  /*远程帧*/
8
9  /*
10  * 函数名：CAN_SetMsg
11  * 描述   ：CAN 通信报文内容设置,设置一个数据内容为 0-7 的数据包
12  * 输入   ：无
13  * 输出   ：无
14  * 调用   ：外部调用
15  */
16 void CAN_SetMsg(CanTxMsg *TxMessage)
```

```
{
    uint8_t ubCounter = 0;

    //TxMessage.StdId=0x00;
    TxMessage->ExtId=0x1314;                          //使用的扩展 ID
    TxMessage->IDE=CAN_ID_EXT;                        //扩展模式
    TxMessage->RTR=CAN_RTR_DATA;                      //发送的是数据
    TxMessage->DLC=8;                                 //数据长度为 8 字节

    /*设置要发送的数据 0-7*/
    for (ubCounter = 0; ubCounter < 8; ubCounter++)
    {
        TxMessage->Data[ubCounter] = ubCounter;
    }
}
```

这段代码是我们为了方便演示而自己定义的设置报文内容的函数，它把报文设置成了扩展模式的数据帧，扩展 ID 为 0x1314，数据段的长度为 8，且数据内容分别为 0-7，实际应用中您可根据自己的需求发设置报文内容。当我们设置好报文内容后，调用库函数 CAN_Transmit 即可把该报文存储到发送邮箱，然后 CAN 外设会把它发送出去：

CAN_Transmit(CANx, &TxMessage);

接收报文

由于我们设置了接收中断，所以接收报文的操作是在中断的服务函数中完成的，见代码清单 39-10。

代码清单 39-10 接收报文(stm32f4xx_it.c)

```

/*接收中断服务函数*/
#define CAN_RX_IRQHandler                    USB_LP_CAN1_RX0_IRQHandler

extern __IO uint32_t flag ;       //用于标志是否接收到数据，在中断函数中赋值
extern CanRxMsg RxMessage;                //接收缓冲区
/*****************************************************************/
void CAN_RX_IRQHandler(void)
{
    /*从邮箱中读出报文*/
    CAN_Receive(CANx, CAN_FIFO0, &RxMessage);

    /* 比较 ID 是否为 0x1314 */
   if ((RxMessage.ExtId==0x1314) && (RxMessage.IDE==CAN_ID_EXT) && (RxMessage.DLC==8) )
    {
        flag = 1;                          //接收成功
    }
    else
    {
        flag = 0;                         //接收失败
    }
}
```

根据我们前面的配置，若 CAN 接收的报文经过筛选器匹配后会被存储到 FIFO0 中，并引起中断进入到这个中断服务函数中，在这个函数里我们调用了库函数 CAN_Receive 把报文从 FIFO 复制到自定义的接收报文结构体 RxMessage 中，并且比较了接收到的报文 ID 是否与我们希望接收的一致，若一致就设置标志 flag=1，否则为 0，通过 flag 标志通知主程序流程获知是否接收到数据。

要注意如果设置了接收报文中断，必须要在中断内调用 CAN_Receive 函数读取接收 FIFO 的内容，因为只有这样才能清除该 FIFO 的接收中断标志，如果不在中断内调用它清除标志的话，一旦接收到报文，STM32 会不断进入中断服务函数，导致程序卡死。

3. main 函数

最后我们来阅读 main 函数，了解整个通讯流程，见代码清单 39-11。

代码清单 39-11 main 函数

```
1
2  _IO uint32_t flag = 0;        //用于标志是否接收到数据，在中断函数中赋值
3  CanTxMsg TxMessage;               //发送缓冲区
4  CanRxMsg RxMessage;               //接收缓冲区
5
6  /**
7    * @brief  主函数
8    * @param  无
9    * @retval 无
10   */
11 int main(void)
12 {
13     LED_GPIO_Config();
14
15     /*初始化 USART1*/
16     Debug_USART_Config();
17
18     /*初始化按键*/
19     Key_GPIO_Config();
20
21     /*初始化 can,在中断接收 CAN 数据包*/
22     CAN_Config();
23
24     printf("\r\n 欢迎使用秉火  STM32 开发板。\r\n");
25     printf("\r\n 秉火 CAN 通讯实验例程\r\n");
26
27     printf("\r\n 实验步骤：\r\n");
28
29     printf("\r\n 1.使用导线连接好两个 CAN 讯设备\r\n");
30     printf("\r\n 2.使用跳线帽连接好:5v --- C/4-5V \r\n");
31 printf("\r\n 3.按下开发板的 KEY1 键，会使用 CAN 向外发送 0-7 的数据包，包的扩展 ID 为 0x1314 \r\n");
32 printf("\r\n 4.若开发板的 CAN 接收到扩展 ID 为 0x1314 的数据包，会把数据以打印到串口。 \r\n");
33     printf("\r\n 5.本例中的 can 波特率为 1MBps，为 stm32 的 can 最高速率。 \r\n");
34
35     while (1)
36     {
37         /*按一次按键发送一次数据*/
38         if (    Key_Scan(KEY1_GPIO_PORT,KEY1_PIN) == KEY_ON)
39         {
40             LED_BLUE;
41             /*设置要发送的报文*/
42             CAN_SetMsg(&TxMessage);
43             /*把报文存储到发送邮箱，发送*/
44             CAN_Transmit(CANx, &TxMessage);
45
46             can_delay(10000);//等待发送完毕，可使用 CAN_TransmitStatus 查看状态
47
48             LED_GREEN;
49
50             printf("\r\n 已使用 CAN 发送数据包！\r\n");
51             printf("\r\n 发送的报文内容为：\r\n");
```

```
52              printf("\r\n 扩展 ID 号 ExtId: 0x%x \r\n",TxMessage.ExtId);
53              CAN_DEBUG_ARRAY(TxMessage.Data,8);
54          }
55          if (flag==1)
56          {
57              LED_GREEN;
58              printf("\r\nCAN 接收到数据: \r\n");
59
60              CAN_DEBUG_ARRAY(RxMessage.Data,8);
61
62              flag=0;
63          }
64      }
65  }
```

在 main 函数里，我们调用了 CAN_Config 函数初始化 CAN 外设，它包含我们前面解说的 GPIO 初始化函数 CAN_GPIO_Config、中断优先级设置函数 CAN_NVIC_Config、工作模式设置函数 CAN_Mode_Config 以及筛选器配置函数 CAN_Filter_Config。

初始化完成后，我们在 while 循环里检测按键，当按下实验板的按键 1 时，它就调用 CAN_SetMsg 函数设置要发送的报文，然后调用 CAN_Transmit 函数把该报文存储到发送邮箱，等待 CAN 外设把它发送出去。代码中并没有检测发送状态，如果需要，您可以调用库函数 CAN_TransmitStatus 检查发送状态。

while 循环中在其它时间一直检查 flag 标志，当接收到报文时，我们的中断服务函数会把它置 1，所以我们可以通过它获知接收状态，当接收到报文时，我们把它使用宏 CAN_DEBUG_ARRAY 输出到串口。

39.6.3 下载验证

下载验证这个 CAN 实验时，我们建议您先使用“CAN—回环测试”的工程进行测试，它的环境配置比较简单，只需要一个实验板，用 USB 线使实验板“USB TO UART”接口跟电脑连接起来，在电脑端打开串口调试助手，并且把编译好的该工程下载到实验板，然后复位。这时在串口调试助手可看到 CAN 测试的调试信息，按一下实验板上的 KEY1 按键，实验板会使用回环模式向自己发送报文，在串口调试助手可以看到相应的发送和接收的信息。

使用回环测试成功后，如果您有两个实验板，需要按照“硬件设计”小节中的图例连接两个板子的 CAN 总线，并且一定要接上跳线帽给 CAN 收发器供电、把摄像头拔掉防止干扰。用 USB 线使实验板“USB TO UART”接口跟电脑连接起来，在电脑端打开串口调试助手，然后使用“CAN—双机通讯”工程编译，并给两个板子都下载该程序，然后复位。这时在串口调试助手可看到 CAN 测试的调试信息，按一下其中一个实验板上的 KEY1 按键，另一个实验板会接收到报文，在串口调试助手可以看到相应的发送和接收的信息。

39.7 课后练习

1. 在工程中尝试修改发送报文的 ID，试验它能不能经过筛选器被接收到。
2. 修改筛选器的掩码配置，使得它能接收 ID 为“0x 13xx”的报文。