https://www.cnblogs.com/luoxiao23/p/11210592.html可以学习以下这个。这是后来发现的

---

https://wenku.baidu.com/view/25d8fb49b6360b4c2e3f5727a5e9856a561226d2.html

https://wenku.baidu.com/view/f8b05f6ba36925c52cc58bd63186bceb19e8ed77.html

https://blog.csdn.net/weixin_28900531/article/details/79989088

http://www.dzsc.com/data/2015-10-14/108872.html 总线拓扑结构

地线处理

线材：485屏蔽双绞线

连接器：接线器1 紫铜GT连接管电线接头压接端子小铜管对接端子连接器套装对接 配合 压线器

---

can是现场总线，称为了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

现场总线（Field bus）是近年来迅速发展起来的一种工业数据总线，它主要解决工业现场的智能化仪器仪表、控制器、执行机构等现场设备间的数字通信以及这些现场控制设备和高级控制系统之间的信息传递问题。由于现场总线简单、可靠、经济实用等一系列突出的优点，因而受到了许多标准团体和计算机厂商的高度重视。

它是一种工业数据总线，是自动化领域中底层数据通信网络。

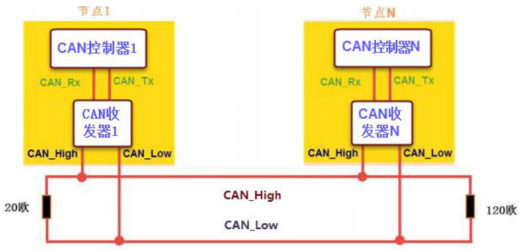
简单说，现场总线就是以数字通信替代了传统4-20mA模拟信号及普通开关量信号的传输，是连接智能现场设备和自动化系统的全数字、双向、多站的通信系统。

物理层

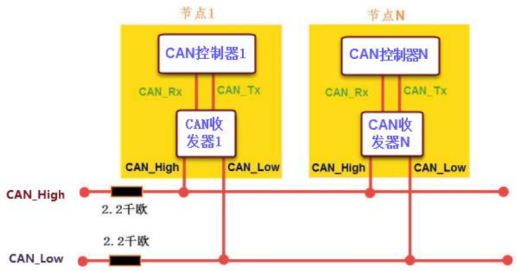
CAN是 一种异步通讯，只有 CAN_H igh 和 CAN_L ow 两条 信号线，共同构成一组 差 分信号线，以差分 信号的形式进行通讯。提高压差和使用差分方式抑制共模干扰。

总线形式

闭环总线两端各有一个120欧姆的匹配电阻，传输距离最远40m，最高速度1M。



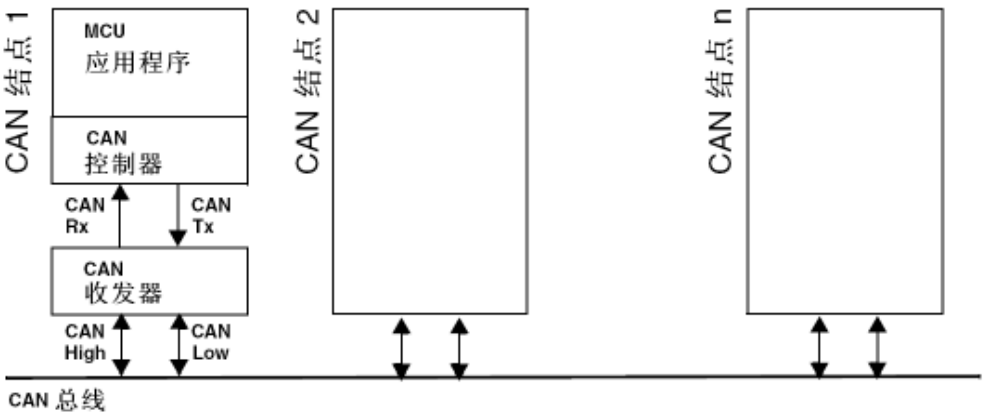
开环总线，每根总线上串联2.2K欧姆，最远距离1Km，通信速率125kbps。



通信节点：挂载在总线上的单元，通过总线实现各个节点间的通信。由于时延和电气特性，越低的传输速率可以挂载越多的节点

can节点由两部分组成

- can控制器：实现can总线协议层以及数据链路层
  - 生成can数据帧并以二进制码流发出，其中包括：位填充，添加CRC校验，应答检测。
  - 或将接收到的二进制码流进行解析并接收，进行收发比对，去位填充，执行CRC校验
  - 还需要执行总线冲突判断，错误处理等。
- can收发器：也叫驱动器
  - 将can控制器的二进制码流转换成差分信号输出
  - 将差分信号转换成二进制码流
- 其实stm32中的can是指其有can控制器，是没有收发器的，所以需要外加can驱动器。can控制器发出can_rx/can_tx,收发器将其转换成总线上的can_h/can_l信号。或者说是逻辑电平和差分信号的转换。



can信号是差模信号/差分信号。

逻辑1—— 隐性电平 —— CAN_H/L均为2.5V，差分电平0。

逻辑0—— 显性电平 —— CAN_H-3.5V CAN_L-1.5V 差分电平2V

由此可知，总线的逻辑状态由显性优先。故为显性。即有一个单元输出显性，则总线就是显性的。类似于线与的意思。会有高阻态。而显性电平是强驱动的，隐性电平弱驱动(开集或开漏输出)，故可以覆盖电平。这也是冲突仲裁的基础

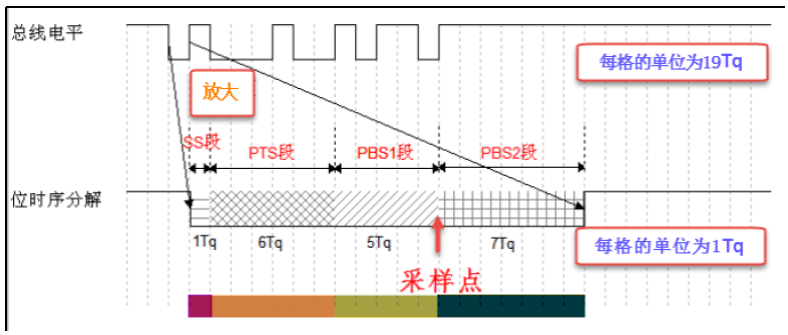
那么就知道了can是半双工的，同一时刻只能有一个单元发出信号，其他节点处于接收状态。

协议层

有位时序的概念，由于CAN 属于 异步通讯，没有时钟信号线， 那么 节点间就需要约定好波特率进行通讯 。同时 ， 为实现 正确 的总线电平采样，确保通讯正常， 由此 引出 位时序 的 概念 。

位时序

can协议将每一位bit的分成了4段时序，由8-25个Tq(Time quantum 时间量子，即最小时间单位)来组成1个位的时长。



- 同步段：SS段，要求有一个跳变沿在此时间段内，用于总线各时钟同步。该段长度位1Tq。（有点把数据线分时当作时钟线用的样子，某段时间内他是时钟线，比如在SS段，然后其他时间内当数据线？）
- 传播时间段：用于补偿网络的物理延时，长度位1-8Tq
- 相位缓冲1：大小位1-8Tq，用以补充边沿的误差，重同步中可以加长
- 相位缓冲2：2-8Tq，也是用于补充边沿误差，重同步过程中可以加长

重新同步跳跃宽度(SJW)定义了，在每位中可以延长或缩短多少个时间单元的上限。其值可以编程为1到4个时间单元。

通信过程中需要约定Tq大小和各个阶段Tq个数，这也才确定了波特率，然后就可以通信了。

采样点

$$波特率 = \frac{1}{正常的位时间}$$

$$正常的位时间 = 1 \times t_q + t_{BS1} + t_{BS2}$$

其中：

$t_{BS1} = t_q \times (TS1[3:0] + 1)$,

$t_{BS2} = t_q \times (TS2[2:0] + 1)$,

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

这里 t_q 表示1个时间单元

t_{PCLK} = APB 时钟的时间周期

BRP[9:0], TS1[3:0] 和 TS2[2:0] 在 CAN_BTR 寄存器中定义

同步机制

CAN规范 定义了自己独有的同步方式，同步与时序密切相关，由节点自身完成，节点检测总线上的边沿和自身的时序进行比较，并通过以下两种同步来调整时序。同步只有在节点检测到逻辑1->0的跳变才会产生，我们知道同步段需要跳变沿，当跳变沿不在同步段的时候，就会产生相位差

- 硬同步
  - 只在总线空闲的时候通过一个下降沿(或叫起始帧)来完成，此时不管有没有相位差，所有节点的时序重新同步。强迫引起硬同步的跳变沿位于重新开始的位时间的同步段内。
- 重同步
  - 在消息帧后，每当有逻辑1->0跳变，并且落在了同步段外。就会引起一次重同步。重同步会根据跳变沿的增长和缩短位时序，来调整采样点保证正确采样。

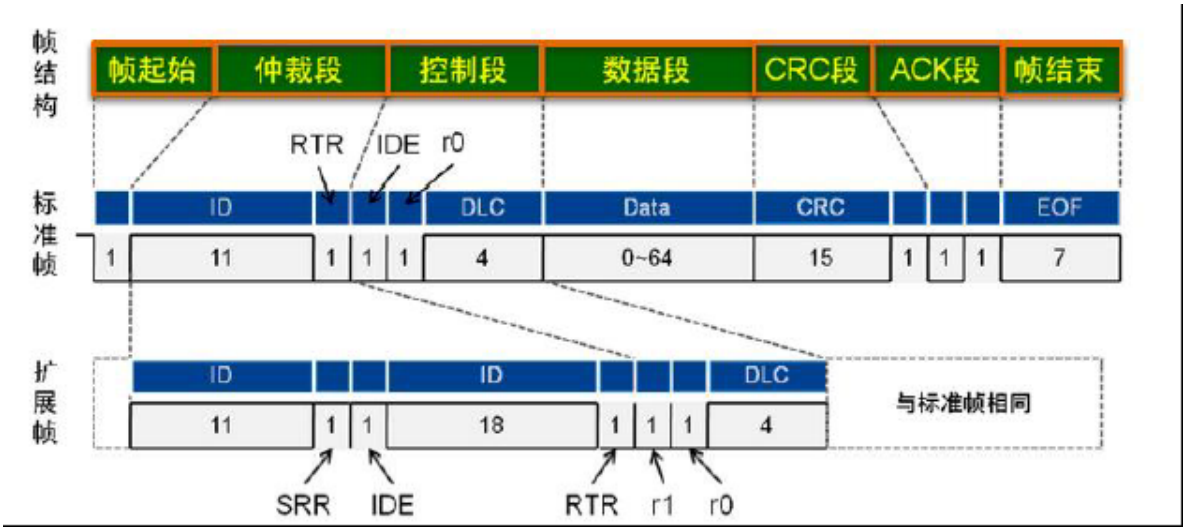
报文格式

can协议对数据、操作、同步信号打包，变成报文。

can一共有五种类型的帧：其中数据帧和遥控帧有标准格式和拓展格式两种，只是标识符ID不同

- 数据帧：发送单元通过此帧向接收单元发送信息
- 遥控帧：接收单元通过此帧向具有相同 ID 的 发送单元请求数据错误
- 错误帧：当检测到错误时，向其他单元通 知 检测到错误
- 过载帧：接收单元用来通知它还没有准备好接收帧
- 间隔帧：用来将数据帧或遥控帧同前面的帧隔开

以一个显性电平（逻辑0）开始，以7个连续的隐电平(逻辑1)结束。中间还有仲裁段，控制段，数据段，CRC段，ACK段。总共7段。



- 帧起始：SOF(Start Of Frame)产生一个显性电平(逻辑0)来表示。用于通知各个节点，将有数据传输。其他节点通过该帧起始信号电平跳变进行硬同步
- 仲裁段：用以确定优先级的段。标准格式和扩展格式不同点在于其ID长度不同，标准位11位，而扩展的位29位
  - ID：本数据帧的ID（标识符）信息。
  - RTR：remote transmission require，远程传输请求。用显性电平和隐性电平区分数据帧和遥控帧
  - IDE：identifier extended，ID扩展位。用显/隐 以区分标准帧/扩展帧
  - SRR：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，标准格式的优先级更高。
  - 如果有两个节点竞争总线的占有，则先出现隐性电平的退出竞争，进入接受模式(检测自己发送的和总线上读取的电平是否一致，如果不一致，表示自己发送的是隐性电平，而别人是显性的，信号被覆盖，对方优先级比自己高)。也正是由于这样优先级的分配 原则，使CAN扩展性大大加强，在总线上增加或减少节点并不会影响其它的设备。
- 控制段：r1r0为保留位，默认为显性，最主要的由四位组成的DLC段，MSB先行，表示的数字从0-8。DLC表示发送数据的长度，表示从0-8字节。
- 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：校验段，为了保证传输的正确，can报文包含了一段15位的CRC校验码。如果接收者计算CRC与此CRC不服。表示数据出错，然后发送错误帧，请求重发。最后还有一位定界符，用以分隔CRC和ACK。
- ACK段：发送端发送的这位为隐性，接收端在这位中发送显性表示应答。然后ACK槽后跟着一个ACK定界符。
- 帧结束：发送节点发送7个隐性位表示结束

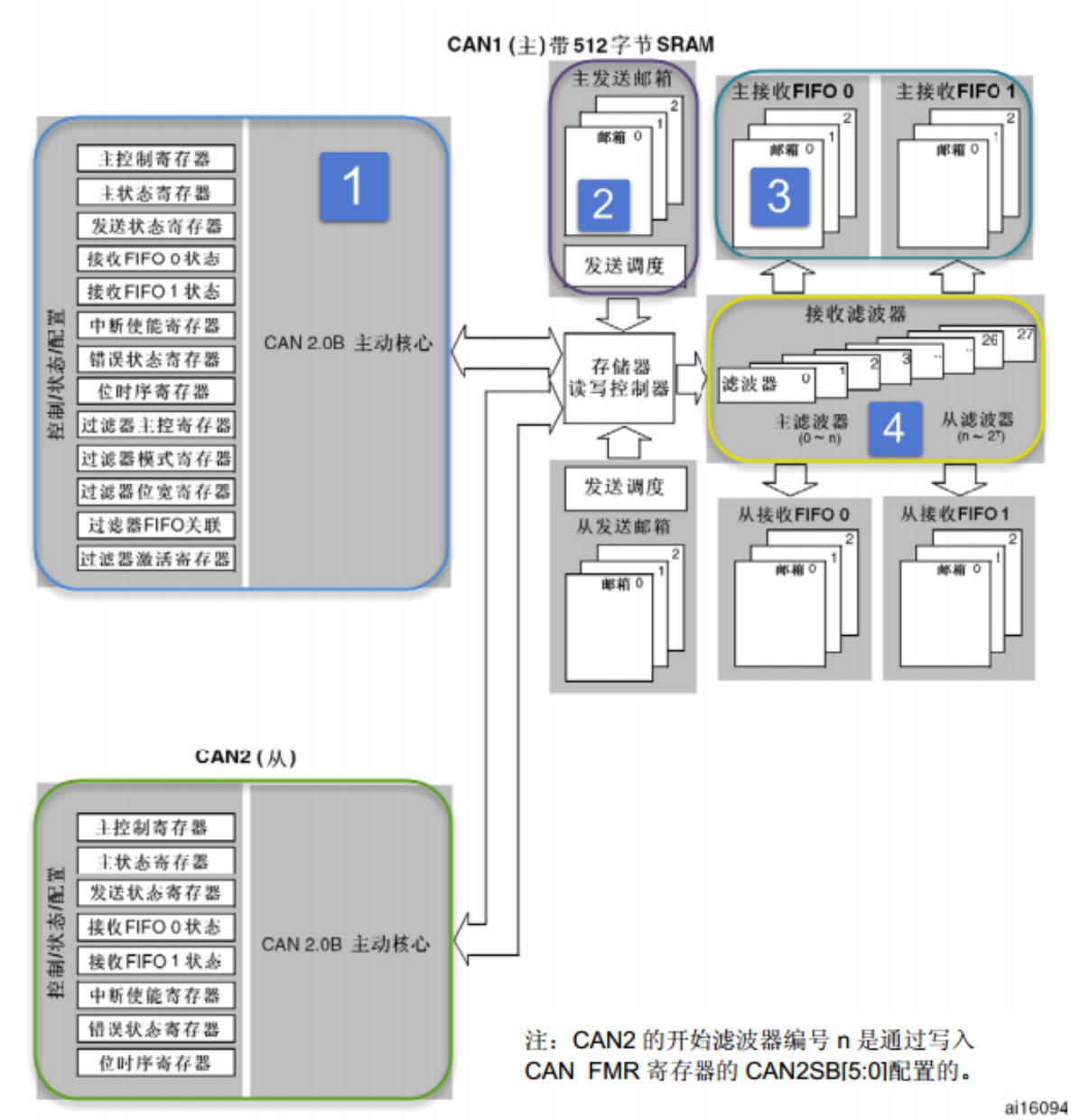
STm32的can

STm32具有bxcan(Basic Extended Can 基本扩展can)，支持以下两种协议(CAN只定义了物理层和数据链路层)

- CAN 2.0A(CAN协议的PART A部分，此部分定义了11bit的标识区)
- CAN 2.0B（也叫PART B，定义了29bit的标识区，其它部分与CAN2.0A一样）

该CAN控制器支持最高的通讯速率为1M b/s ； 它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.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。

具有双CAN ： CAN1是主bxCAN CAN2为 bxCAN 。这两个CAN外设各自都有自己的发送邮箱，接收FIFO0和FIFO1，但是，CAN1除了这个之外，还有接收过滤器，而CAN2没有。但是，在实际工作中，这个接收过滤器是只需要一个，并不是两路CAN各自都需要，因此CAN2完全可以共享CAN1的接收过滤器（这个就是CAN1与CAN2共享的512个字节的SRAM了）只不过是在芯片内部通过CAN1的存储器读写控制器间接的访问，因为CAN1为主设备，负责管理bxCAN和512kb的sram之间的通信。CAN2不能直接访问sram。且在使用CAN2的时候要使能CAN1时钟。从这种CAN1和CAN2的结构上来说，将CAN1看成是主CAN,CAN2看成是从CAN就不足为奇了，除了称呼，在使用和功能上没有任何区别，这些都只是芯片内部bxCAN的设计，对外bxCAN完全是多主模式。



注：在中容量和大容量产品中，USB和CAN共用一个专用的512字节的SRAM存储器用于数据的发送和接收，因此不同同时使用USB和CAN(共享的SRAM被USB和CAN模块互斥地访问)。USB和CAN可以同时用于一个应用中但不能在同一个时间使用

解析

1. 为控制内核
    - 主控制寄存器 CAN_MCR
        - 调试冻结DBF
        - 时间触发通信模式 TTCM
        - 自动离线管理ABOM
        - 自动唤醒 AWUM
        - 自动重传 NART
        - 接收FIFO锁定模式 RFLM
        - 发送FIFO优先级 TXFP
    - CAN位 时序寄存器 CAN_BTR
        - SILM 和 LBKM位可以设定正常、静肃、回环、禁止回环模式
        - 波特率分频器BRP,时间段1,2寄存器 TS1,TS2
            - BRP设定Tq=(BRP+1)*APB1_PCLK周期
            - stm32的位时间长与之前有所不同，只有同步段，时间段1和2，传输段被时间段1包括了。
            - 然后$t_{bs1}和t_{bs2}$=Tq*(TS1或2 +1)
2. 三个发送邮箱(可以缓存3个待发送报文，根据优先级决定哪个报文先发)。

    作用：每个邮箱有自己的 ： 我们需要发送报文的时候就把报文各个段分解了，写入到这些寄存器中，然后将TIxR中的发送请TXRQ置1，即可发送。
    - 标识符寄存器TIxR：32位，(STID/EXID_H)、(EXID_L)、(IDE)、(RTR)、(TXRQ，软件置1，发送完成时硬件清零)
    - 数据长度和时间戳寄存器TDTxR：32位，(TIME16位时钟计数)、(TGT是否发送时间戳，如果发送时间戳，则数据最后两位为时间戳)、(DLC发送数据长度8-0字节)
    - 两个数据寄存器TDLxR,TDHxR，各32位同8字节。
3. 两个三级FIFO做接收邮箱，共可以存6个接收到的报文，只能读取最近接收的报文。接收到报文后FIFO的计数器+1，读取FIFO数据后，计数器-1。可以读取计数值。他们也有同样4个寄存器。
4. 接收过滤器：硬件通过配置，根据报文的标识符决定是否接收到FIFO接收邮箱中否。有两种模式：ID列表模式和掩码模式，有28组过滤器？每组过滤器有2个32位寄存器。
    - FMR寄存器中的FINT位可以设置初始化模式或正常模式
    - FM1R过滤器模式寄存器，可以设置FBMx设置标识符屏蔽位模式或标识符列表模式
    - FS1R过滤器位宽寄存器，FSCx可以设置过滤器位宽2个16或1个32
5. 过滤器FIFO关联寄存器：将对应的过滤组过滤出来的放置对应的fifo中

工作状态

- 初始化模式：无法发送接收
- 正常模式：设置MCR中INRQ为0然后检测到连续11个隐性电平后进入。可以发送接收
- 睡眠模式：
- 静默模式：只能接收数据和遥控帧，但是只能发出隐性位，如果需要发送显性，则会被接回来被内部检测到。不会对总线产生印象还是位长隐性。
- 环回模式：将发送的报文当作接收的，如果他能通过滤波器就会进入邮箱

发送处理

软件中选择一个空邮箱，然后

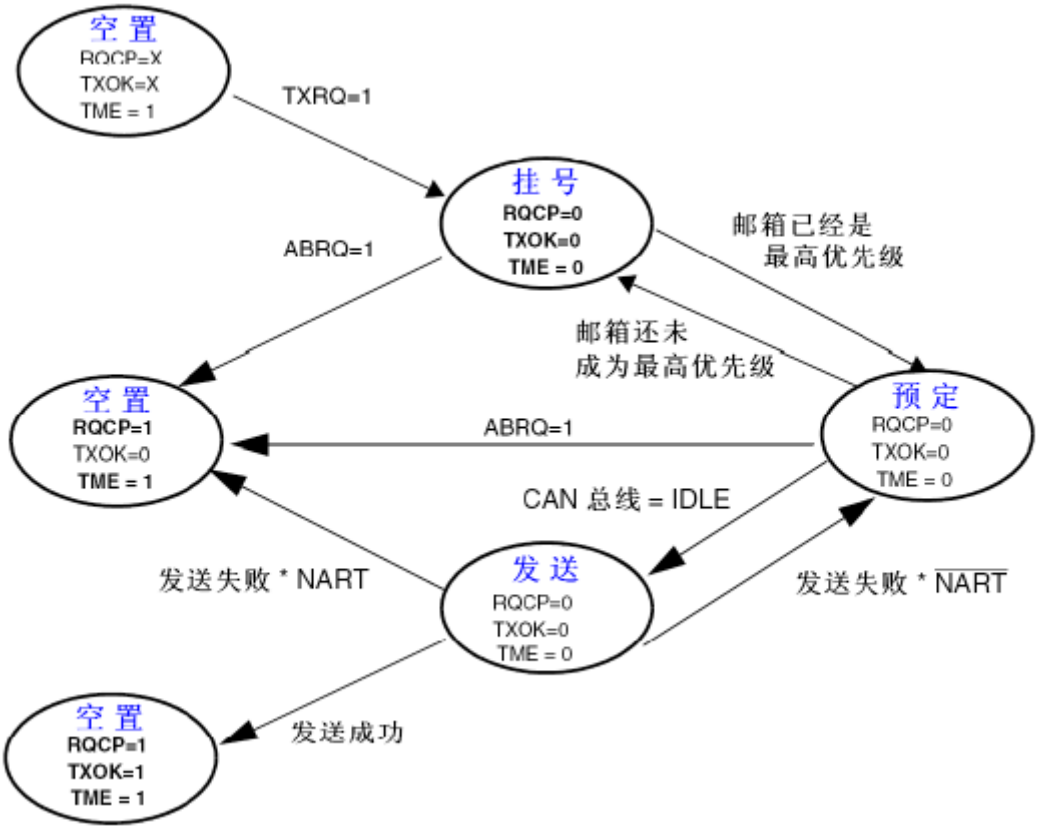
空邮箱：设置其标识符，数据长度，和待发送数据$\stackrel{置TXRQ1}{\longrightarrow}$挂号：等待称为最高优先级邮箱，且此时不能被写

$\stackrel{成为最高优先级邮箱}{\longrightarrow}$预定发送状态$\stackrel{总线空闲}{\longrightarrow}$发送状态$\stackrel{发送成功}{\longrightarrow}$空邮箱，且TSR(发送状态寄存器)的RQCP和TXOK置1。（如果失败了则由仲裁引起的话是TSR的ALST置1，发送错误引起的是TERR置1）

- 超过一个邮箱再挂号时，邮箱优先级由邮箱中报文的标识符决定，低的优先级高，如果标识符一致，则邮箱号小的先发。
- 如果设置了MCR(can主控寄存器)的TXFP为1，则把邮箱变成发送fifo，根据请求顺序来发送。

对TSR的ABRQ置1，如果处于挂号和预定，则进入空闲。如果发送中，则可能发送成功，变成空邮箱，TXOK=1。也可能发送失败，变为预定(自动重发的话)，然后被中止，进入空闲，TXOK=0。

用于can的时间触发通信选项。对MCR的NART置1来让硬件工作在这个状态，该模式下发送只会执行一次，不管出错没。且RQCP都置1表示请求完成，然后TXOK,ALST,TERR来显示是否发送正确。



时间触发通信模式

can内部时钟被激活。内部定时器在每个位时间累加(不是Tq)。内部定时器的值在帧起始的采样点被读取。产生时间戳存放在TDTxR或RDTxR中。

接收管理

接收到的报文，被存储在3级邮箱深度的FIFO中。

FIFO完全由硬件来管理，从而节省了CPU的处理负荷，简化了软件并保证了数据的一致性。

应用程序只能通过读取FIFO输出邮箱，来读取FIFO中最先收到的报文。

当报文被正确接收，也就是说直到EOF的最后一位都没有错误，且通过了标识符过滤器，那么报文就是有效的。

由两个寄存器，RFR接收fifo寄存器的FMP挂号状态和FOVR溢出标识。

当fifo为空的时候，FMP=0x00。

此时收到一个有效报文。FIFO变成挂号状态1，然后FMP=0X01。

软件可以在这时候读取FIFO的输出邮箱来读取报文，并且RFP的RFOM置1来释放邮箱！！这也FIFO又是空的了。否则再来一个邮箱的时候即使你读取了，但是还是会变成挂号2状态，FMP=0X10。再不释放就是0X11。然后再来报文就溢出了，丢失报文且FOVR=1。

溢出会丢失报文，至于丢失了哪个。看fifo的设置。如果设置了MCR的RFLM则变成锁定状态，原来的3个报文不会被覆盖。丢失的是新的报文。

而不设置就是禁用锁定。旧报文会被新报文覆盖。

fifo进入一个报文，则硬件更新RFR的FMP，然后IER的FMPIE置1请求中断。

fifo满了(有3个报文)，则RFR的FULL位就置1，然后IER的FFIE置1请求满中断。

fifo溢出，RFR的FOVR置1，然后IER的FOVIE置1请求溢出中断。

标识符过滤器

也就是上面说的过滤器，这里讲的细致一点。因为在can中是没有硬件地址的，而是在报文中加入标识符，可以认为是i2c中的那个地址位。发送者会把这个报文发给说有人，接收者就需要过滤报文看是否是该节点所需要的了。如果需要就拷贝到sram中，如果不需要就丢弃，无需软件干预。（过了滤波器才进邮箱）

如果没过滤器，是需要软件实现过滤器的，会占用CPU时间。如果有过滤器就由硬件完成过滤。节省CPU时间。

在stm中有28个或14个位宽可变的、可配置的过滤器组。每个过滤组包含两个寄存器FxR0和FxR1

每个过滤器组的位宽都可以单独配置，以满足应用程序

- 1个32位过滤器，包括：STID[10:0]，EXTID[17:0]，IDE，RTR位
- 2个16位过滤器，包括：STID[10:0]，IDE，RTR和EXTID[17:15]位

此外过滤器可配置为，屏蔽位模式和标识符列表模式

标识符寄存器和屏蔽寄存器一起，指定报文标识符的任何一位，应该按照 必须匹配 和 不用关系 处理。

屏蔽寄存器也被当作标识符寄存器使用。不是采用一个标识符加一个屏蔽位的方式，而是使用两个标识符寄存器。接收报文的时候标识符必须和过滤器标识符相同。

配置前通过清除FAR的FACT，把他禁用了。

通过FS1R的FSCx设置位宽。

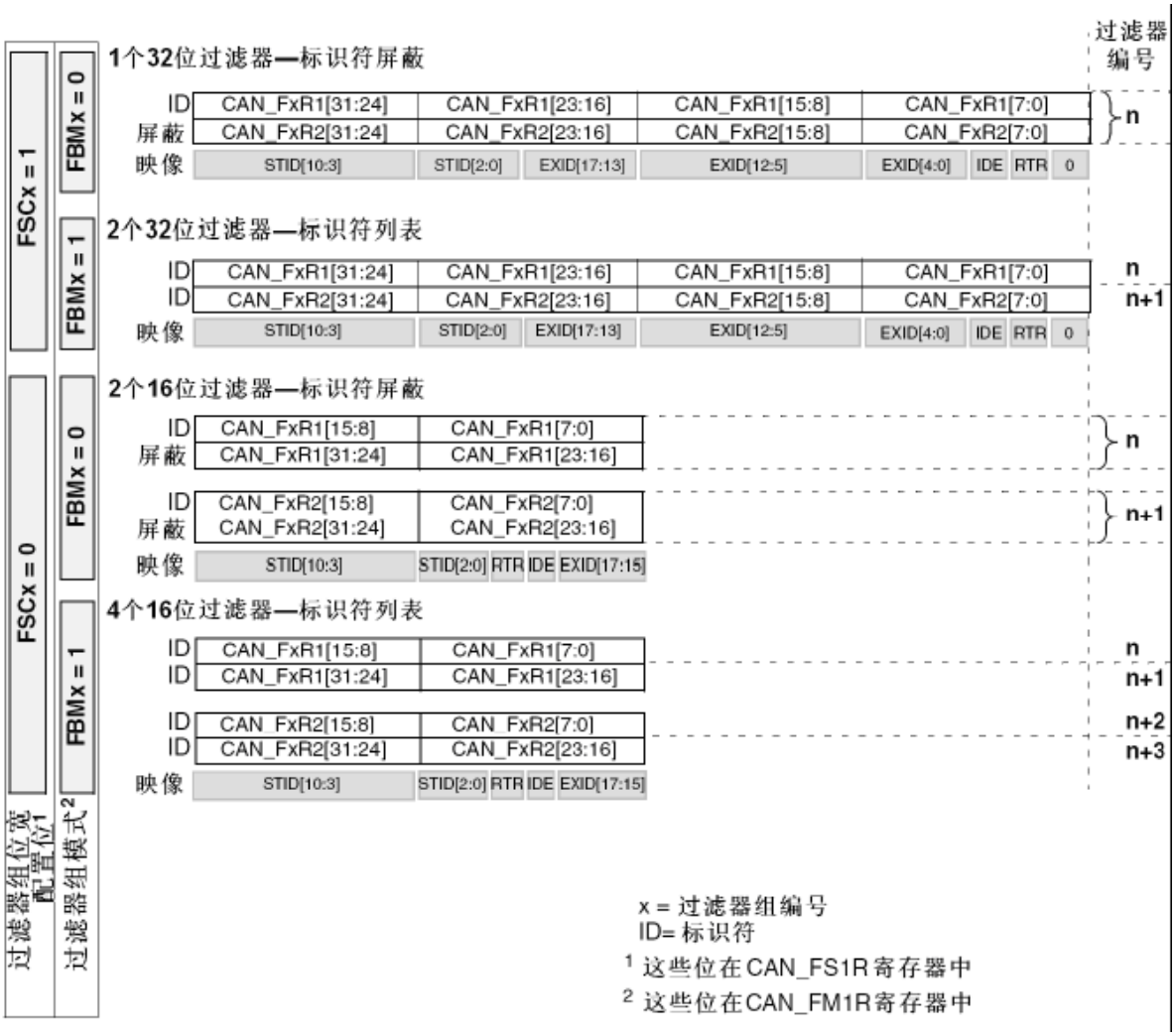
通过FMR的FBMx位，配置屏蔽/标识符寄存器的标识符列表或屏蔽位。

过滤器组中的每个过滤器，都被编号为(叫做过滤器号)从0开始，到某个最大数值 - 取决于过滤器组的模式和位宽的设置。

应用程序不用的过滤器组，应该保持在禁用状态。否则过滤出错(有默认值0000什么的)

为了过滤出一组标识符，应该设置过滤器组工作在屏蔽位模式。 为了过滤出一个标识符，应该设置过滤器组工作在标识符列表模式。

FB[31:0]：过滤器位 (Filter bits)
标识符模式
寄存器的每位对应于所期望的标识符的相应位的电平。
0: 期望相应位为显性位；
1: 期望相应位为隐性位。
屏蔽位模式
寄存器的每位指示是否对应的标识符寄存器位一定要与期望的标识符的相应位一致。
0: 不关心，该位不用于比较；
1: 必须匹配，到来的标识符位必须与滤波器对应的标识符寄存器位相一致。



- 如图所表示的，标识符屏蔽模式需要2个寄存器一起工作，2个32位寄存器(ID和屏蔽寄存器)是1个过滤器，此时过滤组只有1个过滤器
- 标识符列表模式，屏蔽寄存器也当标识符寄存器用。故有2个32位过滤器
- 如果是16位的，再分成一半就好了。

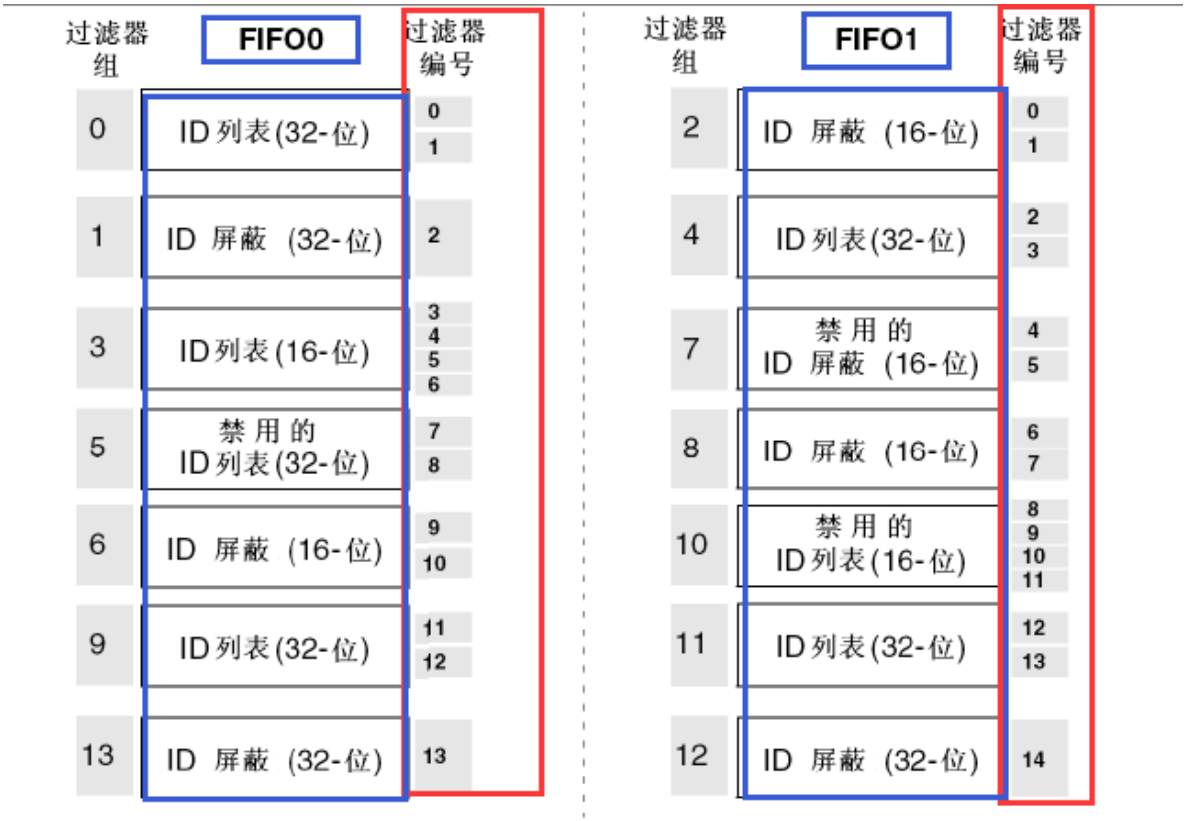
一旦收到的报文被存入FIFO，就可被应用程序访问。

通常情况下，报文中的数据被拷贝到SRAM中；为了把数据拷贝到合适的位置(比如一个数组的哪个位置？？表示哪个过滤器的数据？？)，应用程序需要根据报文的标识符来辨别不同的数据。bxCAN提供了过滤器匹配序号，以简化这一辨别过程。 根据过滤器优先级规则，过滤器匹配序号和报文一起，被存入FIFO邮箱中。因此每个收到的报文，都有与它相关联的过滤器匹配序号(知道存在哪里？？)。

过滤器匹配序号可以通过下面两种方式来使用： ● 把过滤器匹配序号跟一系列所期望的值进行比较 ● 把过滤器匹配序号当作一个索引来访问目标地址

对于标识符列表模式下的过滤器(非屏蔽方式的过滤器)，软件不需要直接跟标识符进行比较。 对于屏蔽位模式下的过滤器，软件只须对需要的那些屏蔽位(必须匹配的位)进行比较即可。

在给过滤器编号时，并不考虑过滤器组是否为激活状态。



由图可知，每个过滤器组根据配置可以变成0-4个过滤器。每个过滤器组可以关联到FIFO中，即过滤出来的报文放在哪个fifo中。且每个fifo中单独对这些过滤器进行标号（不管是否被禁用）。

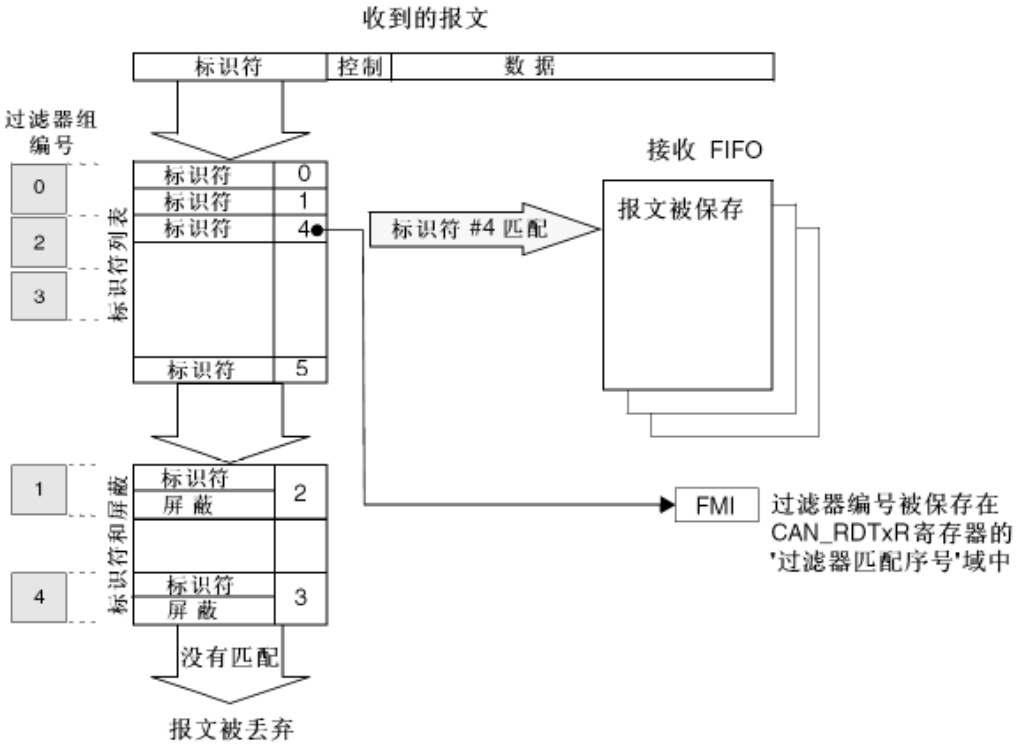
报文根据其标识符有优先级

邮箱的优先级跟他所放的报文优先级一致。

过滤器的优先级是因为有一个报文他可能可以通过多个过滤器，但是过滤器关联了FIFO和过滤器编号。只能存放在sram中的某个位置，不可能同时存入多个位置。所以需要优先级。

在这种情况下，存放在接收邮箱中的过滤器匹配序号，根据下列优先级规则来确定： ● 位宽为32位的过滤器，优先级高于位宽为16位的过滤器 ● 对于位宽相同的过滤器，标识符列表模式的优先级高于屏蔽位模式 ● 位宽和模式都相同的过滤器，优先级由过滤器号决定，过滤器号小的优先级高

例子：3个过滤器组处于标识符列表模式
其它的过滤器组处于标识符屏蔽模式



上面的例子说明了bxCAN的过滤器规则：

在接收一个报文时，其标识符首先与配置在标识符列表模式下的过滤器相比较；如果匹配上，报文就被存放到相关联的FIFO中，并且所匹配的过滤器的序号被存入过滤器匹配序号中。如同例子中所显示，报文标识符跟#4标识符匹配，因此报文内容和FMI4被存入FIFO。 如果没有匹配，报文标识符接着与配置在屏蔽位模式下的过滤器进行比较。 如果报文标识符没有跟过滤器中的任何标识符相匹配，那么硬件就丢弃该报文，且不会对软件有任何打扰。

报文存储

邮箱是硬件和软件之间传递报文的接口。

软件需要在一个空的发送邮箱中，把待发送报文的各种信息设置好(然后再发出发送的请求)。发送的状态可通过查询CAN_TSR寄存器获知。

表163 发送邮箱寄存器列表

相对发送邮箱基地址的偏移量	寄存器名
0	CAN_TIxR
4	CAN_TDTxR
8	CAN_TDLxR
12	CAN_TDHxR

在接收到一个报文后，软件就可以访问接收FIFO的输出邮箱来读取它。

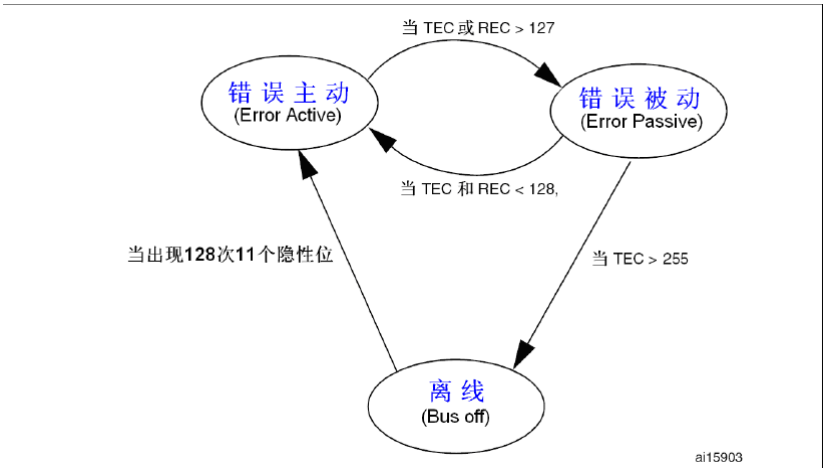
一旦软件处理了报文(如把它读出来)，软件就应该对CAN_RFxR寄存器的RFOM位进行置'1'，来释放该报文

过滤器匹配序号存放在CAN_RDTxR寄存器的FMI域中。

16位的时间戳存放在CAN_RDTxR寄存器的TIME[15:0]域中。

相对接收邮箱基地址的偏移量	寄存器名
0	CAN_RIxR
4	CAN_RDTxR
8	CAN_RDLxR
12	CAN_RDHxR

出错管理



CAN协议描述的出错管理，完全由硬件通过发送错误计数器(CAN_ESR寄存器里的TEC域)，和接收错误计数器(CAN_ESR寄存器里的REC域)来实现，其值根据错误的情况而增加或减少。关于TEC和REC管理的详细信息，请参考CAN标准。软件可以读出它们的值来判断CAN网络的稳定性。

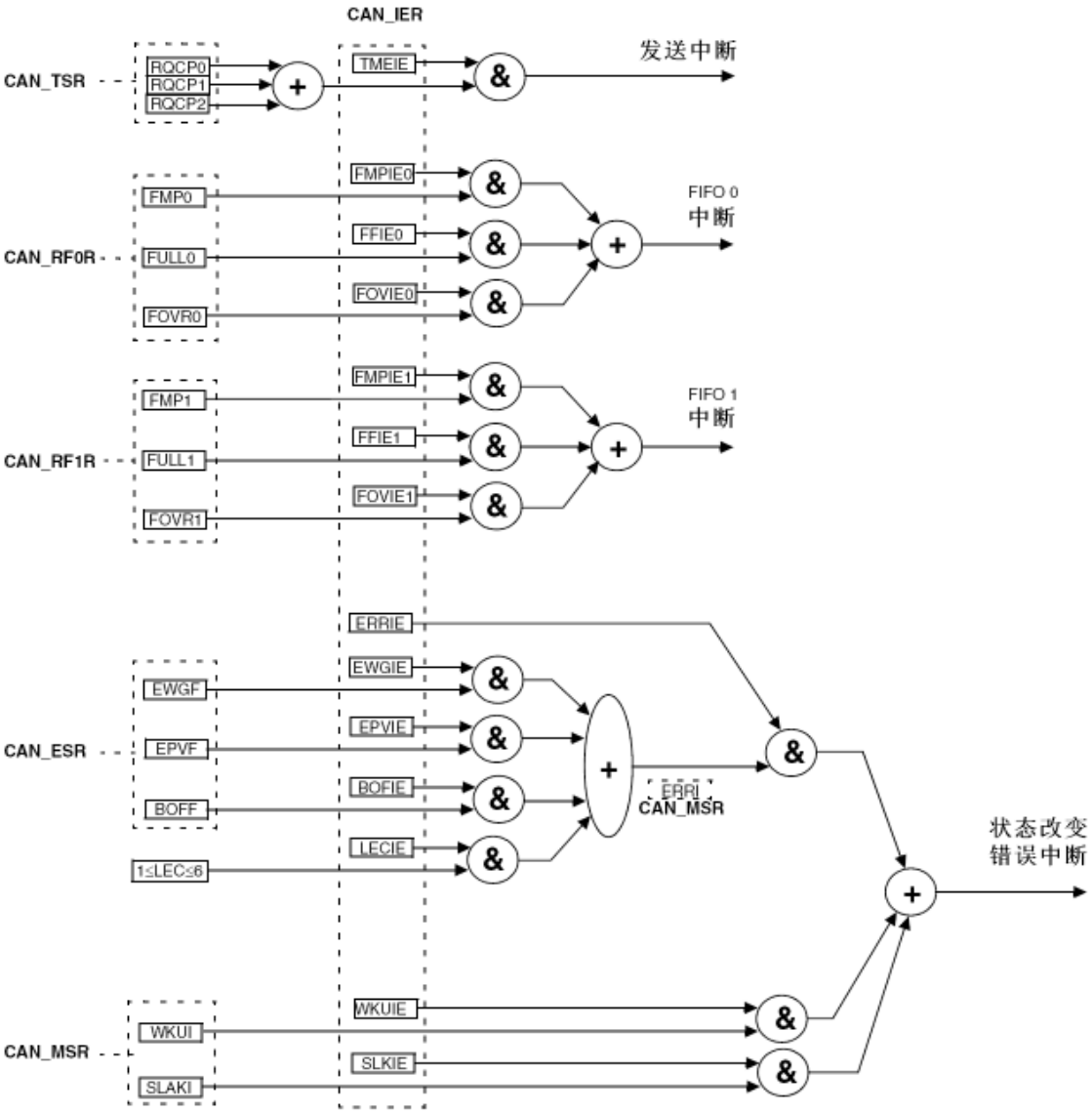
此外，CAN_ESR寄存器提供了当前错误状态的详细信息。通过设置CAN_IER寄存器(比如ERRIE位)，当检测到出错时软件可以灵活地控制中断的产生。

当TEC大于255时，bxCAN就进入离线状态，同时CAN_ESR寄存器的BOFF位被置'1'。

在离线状态下，bxCAN无法接收和发送报文。根据CAN_MCR寄存器中ABOM位的设置，bxCAN可以自动或在软件的请求下，从离线状态恢复(变为错误主动状态)。在这两种情况下，bxCAN都必须等待一个CAN标准所描述的恢复过程(CAN RX引脚上检测到128次11个连续的隐性位)。

如果ABOM位为'1'，bxCAN进入离线状态后，就自动开启恢复过程。 如果ABOM位为'0'，软件必须先请求bxCAN进入然后再退出初始化模式，随后恢复过程才被开启。

中断管理



bxCAN占用4个专用的中断向量。通过设置CAN中断允许寄存器(CAN_IER)，每个中断源都可以单独允许和禁用。

- 发送中断:CAN_TSR
  - 三个邮箱某个为空，发送状态寄存器的RQCPx被置1，触发中断。
- FIFO0中断:CAN_RF0R
  - FIFO有新报文：FMP0不再是00b时触发
  - FIFO满：FULL0被置1
  - FIFO溢出：FOVR0置1
- FIFO1中断同上
- 错误和状态变化中断
  - 出错情况，查看CAN_ESR
  - 唤醒情况，can接收引脚监测到帧起始
  - CAN进入睡眠

CUBE

Master Mode 勾选使能

Bit Timings Parameters	
Prescaler (for Time Quantum)	16
Time Quantum	1000.0 ns
Time Quanta in Bit Segment 1	1 Time
Time Quanta in Bit Segment 2	1 Time
ReSynchronization Jump Width	1 Time
Basic Parameters	
Time Triggered Communication ...	Disable
Automatic Bus-Off Management	Disable
Automatic Wake-Up Mode	Disable
Automatic Retransmission	Disable
Receive Fifo Locked Mode	Disable
Transmit Fifo Priority	Disable
Advanced Parameters	
Operating Mode	Normal

这图包括：

- Prescaler：预分频APB1到CAN_CLK的分频。
  -
```
WRITE_REG(hcan->Instance->BTR, (uint32_t)(hcan->Init.Mode           |
                                          hcan->Init.SyncJumpWidth  |
                                          hcan->Init.TimeSeg1       |
                                          hcan->Init.TimeSeg2       |
                                          (hcan->Init.Prescaler - 1U)));
```
  这里的分频不需要+1，不是填入寄存器的，是数学上的那个分频
  - 下面S1S2的设置也不用-1了。直接多长设置多大即可。
  - 波特率计算记得S1+S2+SS(1)才是一位的长度。
- Time Quantum：Tq的值由以下几个自动算出
- Time Quantum in bit s1/2：位时序中S1S2段占的clk数
- ReSynchronization jump width:重同步跳跃宽度：再重同步时，位的宽最多扩展几个CLK

- Time Triggered Communication :时间触发通信模式
- Auto Bus-off management ： 自动离线管理
- Auto Wake-up Mode：自动唤醒
- Auto Retransmission ： 自动重传
- Receive fifo Locked Mode：接收fifo锁定模式与否
- Transimit fifo priority ： 是否将输出邮箱变成FIFO

- Operating Mode：can的工作状态，有正常，环回，静默，静默环回。

Parameter Settings | User Constants | NVIC Settings | GPIO Settings

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
CAN1 TX interrupts	☐	0	0
CAN1 RX0 interrupts	☐	0	0
CAN1 RX1 interrupt	☐	0	0
CAN1 SCE interrupt	☐	0	0

中断可以配置发送中断等。这四个对应上面说的4个中断向量。接收、FIFO0、FIFO1、错误和状态变化。

KEIL

```
CAN_HandleTypeDef:含有
  CAN_TypeDef                 *Instance;      /*!< Register base address */
  CAN_InitTypeDef             Init;           /*!< CAN required parameters */
  __IO HAL_CAN_StateTypeDef   State;          /*!< CAN communication state */
  __IO uint32_t               ErrorCode;      /*!< CAN Error code.
```

其中CAN_TypeDef是一个结构体，其成员对应手册中的每个寄存器。是软件和硬件的接口。

```
typedef struct
{
  __IO uint32_t              MCR;              /*!< CAN master control register,         Address offset: 0x00          *
  __IO uint32_t              MSR;              /*!< CAN master status register,          Address offset: 0x04          *
  __IO uint32_t              TSR;              /*!< CAN transmit status register,        Address offset: 0x08          *
  __IO uint32_t              RF0R;             /*!< CAN receive FIFO 0 register,         Address offset: 0x0C          *
  __IO uint32_t              RF1R;             /*!< CAN receive FIFO 1 register,         Address offset: 0x10          *
  __IO uint32_t              IER;              /*!< CAN interrupt enable register,       Address offset: 0x14          *
  __IO uint32_t              ESR;              /*!< CAN error status register,           Address offset: 0x18          *
  __IO uint32_t              BTR;              /*!< CAN bit timing register,             Address offset: 0x1C          *
  uint32_t                   RESERVED0[88];    /*!< Reserved, 0x020 - 0x17F                                            *
  CAN_TxMailBox_TypeDef      sTxMailBox[3];    /*!< CAN Tx MailBox,                      Address offset: 0x180 - 0x1AC *
  CAN_FIFOMailBox_TypeDef    sFIFOMailBox[2];  /*!< CAN FIFO MailBox,                    Address offset: 0x1B0 - 0x1CC *
  uint32_t                   RESERVED1[12];    /*!< Reserved, 0x1D0 - 0x1FF                                            *
  __IO uint32_t              FMR;              /*!< CAN filter master register,          Address offset: 0x200         *
  __IO uint32_t              FM1R;             /*!< CAN filter mode register,            Address offset: 0x204         *
  uint32_t                   RESERVED2;        /*!< Reserved, 0x208                                                    *
  __IO uint32_t              FS1R;             /*!< CAN filter scale register,           Address offset: 0x20C         *
  uint32_t                   RESERVED3;        /*!< Reserved, 0x210                                                    *
  __IO uint32_t              FFA1R;            /*!< CAN filter FIFO assignment register, Address offset: 0x214         *
  uint32_t                   RESERVED4;        /*!< Reserved, 0x218                                                    *
  __IO uint32_t              FA1R;             /*!< CAN filter activation register,      Address offset: 0x21C         *
  uint32_t                   RESERVED5[8];     /*!< Reserved, 0x220-0x23F                                              *
  CAN_FilterRegister_TypeDef sFilterRegister[28]; /*!< CAN Filter Register,            Address offset: 0x240-0x31C   *
} CAN_TypeDef;
```

CAN_InitTypeDef是CUBE中填写的内容对应保存在得变量

```
typedef struct
{
  uint32_t Prescaler;                 /*!< Specifies the length of a time quantum.
                                           This parameter must be a number between Min_Data = 1 and Max_Data = 1024.

  uint32_t Mode;                      /*!< Specifies the CAN operating mode.
                                           This parameter can be a value of @ref CAN_operating_mode */

  uint32_t SyncJumpWidth;             /*!< Specifies the maximum number of time quanta the CAN hardware
                                           is allowed to lengthen or shorten a bit to perform resynchronization.
                                           This parameter can be a value of @ref CAN_synchronisation_jump_width */

  uint32_t TimeSeg1;                  /*!< Specifies the number of time quanta in Bit Segment 1.
                                           This parameter can be a value of @ref CAN_time_quantum_in_bit_segment_1 '

  uint32_t TimeSeg2;                  /*!< Specifies the number of time quanta in Bit Segment 2.
                                           This parameter can be a value of @ref CAN_time_quantum_in_bit_segment_2 '

  FunctionalState TimeTriggeredMode;  /*!< Enable or disable the time triggered communication mode.
                                           This parameter can be set to ENABLE or DISABLE. */

  FunctionalState AutoBusOff;         /*!< Enable or disable the automatic bus-off management.
                                           This parameter can be set to ENABLE or DISABLE. */

  FunctionalState AutoWakeUp;         /*!< Enable or disable the automatic wake-up mode.
                                           This parameter can be set to ENABLE or DISABLE. */

  FunctionalState AutoRetransmission; /*!< Enable or disable the non-automatic retransmission mode.
                                           This parameter can be set to ENABLE or DISABLE. */

  FunctionalState ReceiveFifoLocked;  /*!< Enable or disable the Receive FIFO Locked mode.
                                           This parameter can be set to ENABLE or DISABLE. */

  FunctionalState TransmitFifoPriority;/*!< Enable or disable the transmit FIFO priority.
                                           This parameter can be set to ENABLE or DISABLE. */

} CAN_InitTypeDef;
```

剩下得是hal库对CAN这个东西操作时候的状态

errorcode是错误代码

包括了stdid、extid、ide(标准还是扩展帧)、rtr(数据还是遥控帧)、DLC数据长、时间戳。使用时间戳需要开启时间触发通信和数据长度8，且数据的末2位表示时间戳

```
typedef struct
{
  uint32_t StdId;    /*!< Specifies the standard identifier.
                          This parameter must be a number between Min_Data = 0
and Max_Data = 0x7FF. */

  uint32_t ExtId;    /*!< Specifies the extended identifier.
                          This parameter must be a number between Min_Data = 0
and Max_Data = 0x1FFFFFFF. */

  uint32_t IDE;      /*!< Specifies the type of identifier for the message
that will be transmitted.
                          This parameter can be a value of @ref
CAN_identifier_type */

  uint32_t RTR;      /*!< Specifies the type of frame for the message that
will be transmitted.
                          This parameter can be a value of @ref
CAN_remote_transmission_request */

  uint32_t DLC;      /*!< Specifies the length of the frame that will be
transmitted.
                          This parameter must be a number between Min_Data = 0
and Max_Data = 8. */

  FunctionalState TransmitGlobalTime; /*!< Specifies whether the timestamp
counter value captured on start
                          of frame transmission, is sent in DATA6 and DATA7
replacing pData[6] and pData[7].
                          @note: Time Triggered Communication Mode must be
enabled.
                          @note: DLC must be programmed as 8 bytes, in order
these 2 bytes are sent.
                          This parameter can be set to ENABLE or DISABLE. */

} CAN_TxHeaderTypeDef;
```

Timestamp时间戳和上面的不一样，好像是说接受信息时候本地的时间戳而不是报文消息里包含的时间戳。

FilterMatchIndex滤波器序号匹配，表示上面讲到的哪个fifo的滤波器编号。

```
typedef struct
{
  uint32_t StdId;    /*!< Specifies the standard identifier.
                          This parameter must be a number between Min_Data = 0
```

```
  and Max_Data = 0x7FF. */

  uint32_t ExtId;    /*!< Specifies the extended identifier.
                          This parameter must be a number between Min_Data = 0
  and Max_Data = 0x1FFFFFFF. */

  uint32_t IDE;      /*!< Specifies the type of identifier for the message
  that will be transmitted.
                          This parameter can be a value of @ref
  CAN_identifier_type */

  uint32_t RTR;      /*!< Specifies the type of frame for the message that
  will be transmitted.
                          This parameter can be a value of @ref
  CAN_remote_transmission_request */

  uint32_t DLC;      /*!< Specifies the length of the frame that will be
  transmitted.
                          This parameter must be a number between Min_Data = 0
  and Max_Data = 8. */

  uint32_t Timestamp; /*!< Specifies the timestamp counter value captured on
  start of frame reception.
                          @note: Time Triggered Communication Mode must be
  enabled.
                          This parameter must be a number between Min_Data = 0
  and Max_Data = 0xFFFF. */

  uint32_t FilterMatchIndex; /*!< Specifies the index of matching acceptance
  filter element.
                          This parameter must be a number between Min_Data = 0
  and Max_Data = 0xFF. */

  } CAN_RxHeaderTypeDef;
```

滤波器设置

前4个表示的是一个滤波器组的2个32位滤波器寄存器。根据位宽和模式不同，可以分成4个部分。

模式	CAN_FilterId High	CAN_FilterId Low	CAN_FilterMaskId High	CAN_FilterIdLow
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 掩码的完整数值

FilterFIFOAssignment:是和该滤波器关联的FIFO

FilterBank：滤波器组，即使用哪个滤波器组。

FilterMode：滤波器模式，即使用标识符队列模式还是掩码模式

FilterScale：滤波器位宽，16位还是32位

FilterActivation：是否激活滤波器

SlaveStartFilterBank：主从can滤波器组的分界线。对于单独只用主can的不需要管这个。

```
typedef struct
{
  uint32_t FilterIdHigh;          /*!< Specifies the filter identification
number (MSBs for a 32-bit
                                       configuration, first one for a 16-bit
configuration).
                                       This parameter must be a number between
Min_Data = 0x0000 and Max_Data = 0xFFFF. */

  uint32_t FilterIdLow;           /*!< Specifies the filter identification
number (LSBs for a 32-bit
                                       configuration, second one for a 16-bit
configuration).
                                       This parameter must be a number between
Min_Data = 0x0000 and Max_Data = 0xFFFF. */

  uint32_t FilterMaskIdHigh;      /*!< Specifies the filter mask number or
identification number,
                                       according to the mode (MSBs for a 32-
bit configuration,
                                       first one for a 16-bit configuration).
                                       This parameter must be a number between
Min_Data = 0x0000 and Max_Data = 0xFFFF. */

  uint32_t FilterMaskIdLow;       /*!< Specifies the filter mask number or
```

```
identification number,
                                    according to the mode (LSBs for a 32-
bit configuration,
                                    second one for a 16-bit configuration).
                                    This parameter must be a number between
Min_Data = 0x0000 and Max_Data = 0xFFFF. */

  uint32_t FilterFIFOAssignment;  /*!< Specifies the FIFO (0 or 1U) which will
be assigned to the filter.
                                    This parameter can be a value of @ref
CAN_filter_FIFO */

  uint32_t FilterBank;            /*!< Specifies the filter bank which will be
initialized.
                                    For single CAN instance(14 dedicated
filter banks),
                                    this parameter must be a number between
Min_Data = 0 and Max_Data = 13.
                                    For dual CAN instances(28 filter banks
shared),
                                    this parameter must be a number between
Min_Data = 0 and Max_Data = 27. */

  uint32_t FilterMode;            /*!< Specifies the filter mode to be
initialized.
                                    This parameter can be a value of @ref
CAN_filter_mode */

  uint32_t FilterScale;           /*!< Specifies the filter scale.
                                    This parameter can be a value of @ref
CAN_filter_scale */

  uint32_t FilterActivation;      /*!< Enable or disable the filter.
                                    This parameter can be a value of @ref
CAN_filter_activation */

  uint32_t SlaveStartFilterBank;  /*!< Select the start filter bank for the
slave CAN instance.
                                    For single CAN instances, this
parameter is meaningless.
                                    For dual CAN instances, all filter
banks with lower index are assigned to master
                                    CAN instance, whereas all filter banks
with greater index are assigned to slave
                                    CAN instance.
                                    This parameter must be a number between
Min_Data = 0 and Max_Data = 27. */
```

```
} CAN_FilterTypeDef;
```

要使用stm32的can只需要

- 配置好GPIO和中断
- 波特率分频器、工作模式和两个时间段的长度设置好
- 滤波器设置，选哪组、位宽多少、要过滤的ID、关联的fifo
- 设置报文内容即设置
- 邮箱内容，后指定哪个邮箱发送
- 主机端的话，将发送邮箱内容发送一遍，然后开启接收中断，在中断中处理接收数据。

```
void HAL_CAN_MspInit(CAN_HandleTypeDef* canHandle)
{

  GPIO_InitTypeDef GPIO_InitStruct = {0};
  if(canHandle->Instance==CAN1)
  {
  /* USER CODE BEGIN CAN1_MspInit 0 */

  /* USER CODE END CAN1_MspInit 0 */
    /* CAN1 clock enable */
    __HAL_RCC_CAN1_CLK_ENABLE();

    __HAL_RCC_GPIOA_CLK_ENABLE();
    /**CAN1 GPIO Configuration
    PA11     ------> CAN1_RX
    PA12     ------> CAN1_TX
    */
    GPIO_InitStruct.Pin = GPIO_PIN_11|GPIO_PIN_12;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    GPIO_InitStruct.Alternate = GPIO_AF9_CAN1;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

    /* CAN1 interrupt Init */
    HAL_NVIC_SetPriority(CAN1_RX0_IRQn, 0, 0);
    HAL_NVIC_EnableIRQ(CAN1_RX0_IRQn);
  /* USER CODE BEGIN CAN1_MspInit 1 */

  /* USER CODE END CAN1_MspInit 1 */
  }
}
```

接收发送GPIO管脚设置和中断分组配置

```
void MX_CAN1_Init(void)
{

  hcan1.Instance = CAN1;
  hcan1.Init.Prescaler = 3;
  hcan1.Init.Mode = CAN_MODE_NORMAL;
  hcan1.Init.SyncJumpWidth = CAN_SJW_1TQ;
  hcan1.Init.TimeSeg1 = CAN_BS1_9TQ;
  hcan1.Init.TimeSeg2 = CAN_BS2_4TQ;
  hcan1.Init.TimeTriggeredMode = DISABLE;
  hcan1.Init.AutoBusOff = DISABLE;
  hcan1.Init.AutoWakeUp = DISABLE;
  hcan1.Init.AutoRetransmission = DISABLE;
  hcan1.Init.ReceiveFifoLocked = DISABLE;
  hcan1.Init.TransmitFifoPriority = DISABLE;
  if (HAL_CAN_Init(&hcan1) != HAL_OK)
  {
    Error_Handler();
  }

}
```

进行can模式设置

切记要在can初始化HAL_CAN_INIT后才进行滤波器配置和开启can。

```
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(hcan == &hcan1)
  {
    sFilterConfig.FilterBank = 0;
  }


  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;
```

过滤器设置

这里使用0号过滤器组，模式位标识符屏蔽模式，32位宽，滤波器ID寄存器中为0x00000000，屏蔽寄存器中也为0x00000000表示不需要匹配ID寄存器中的任何位，滤波器关联到FIFO0，然后激活滤波器。

如果需要匹配某个标识符才接收，则可以这也设置，相当于单个ID列表模式

```
33      sFilterConfig.FilterIdHigh = (((uint32_t)0x1314<<3)&0xFFFF0000)>>16; //要过滤的 ID 高位
34      sFilterConfig.FilterIdLow = (((uint32_t)0x1314<<3)|CAN_ID_EXT|CAN_RTR_DATA)&0xFFFF; //要过滤的
ID 低位
35      sFilterConfig.FilterMaskIdHigh     = 0xFFFF;        //过滤器高 16 位每位必须匹配
36      sFilterConfig.FilterMaskIdLow      = 0xFFFF;        //过滤器低 16 位每位必须匹配
```

这里的左移3是因为最后一位是保留，第二是RTR，第三是IDE

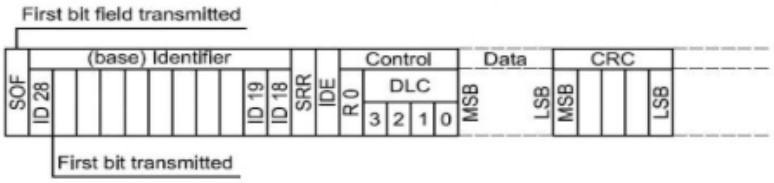
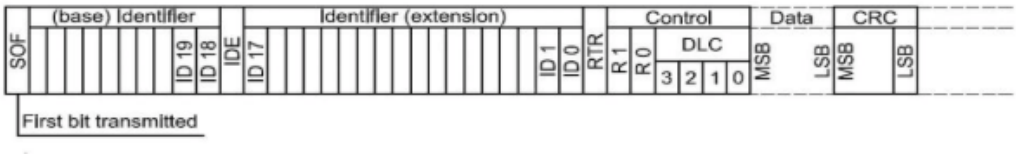


Figure 10 — Order of bit transmission (base format)



也就是说一个扩展id包含了基础id和扩展id两部分。高位表示基础id，低位表示扩展id。

对应以下来填写。如如果是32位屏蔽滤波器模式。则填写方法是把拓展格式id左移3位，这也基础id放在标准id中，扩展部分放在扩展id上，然后补上ide和rtr和一个0保留位。

调用HAL_CAN_ConfigFilter(&hCAN, &sFilterConfig);完成配置

发送的时候

```
void CanTransmit_1234(CAN_HandleTypeDef *hcanx, int16_t cm1_iq, int16_t cm2_iq, int16_t cm3_iq, int16_t cm4_iq)
{
  CAN_TxHeaderTypeDef TxMessage;

  TxMessage.DLC=0x08;
  TxMessage.StdId=0x200;
  TxMessage.IDE=CAN_ID_STD;
  TxMessage.RTR=CAN_RTR_DATA;

  uint8_t TxData[8];
  TxData[0] = (uint8_t)(cm1_iq >> 8);
  TxData[1] = (uint8_t)cm1_iq;
  TxData[2] = (uint8_t)(cm2_iq >> 8);
  TxData[3] = (uint8_t)cm2_iq;
  TxData[4] = (uint8_t)(cm3_iq >> 8);
  TxData[5] = (uint8_t)cm3_iq;
  TxData[6] = (uint8_t)(cm4_iq >> 8);
  TxData[7] = (uint8_t)cm4_iq;

  if(HAL_CAN_AddTxMessage(hcanx,&TxMessage,TxData,(uint32_t*)CAN_TX_MAILBOX0)!=HAL_OK)
  {
     Error_Handler();       //如果CAN信息发送失败则进入死循环
  }
}
```

配置发送邮箱的报文，如配置DLC数据长度，标准id，ide(是什么id格式)，rtr(遥控还是数据帧)，然后填写数据。

调用HAL_CAN_AddTxMessage(hcanx,&TxMessage,TxData,(uint32_t*)CAN_TX_MAILBOX0）来发送

需要填写can的句柄，发送邮箱报文，指定发送内容，指定用哪个邮箱发。

接收使用接收中断

中断来了调用中断服务函数，hal库读取寄存器判断是什么类型的接收中断

读取CAN_RF0R_FMP0不为0且还在中断则进行中断回调。

在回调中使用HAL_CAN_GetRxMessage(句柄，哪个fifo，接收报文信息类型，接收数据buf)

```
HAL_StatusTypeDef HAL_CAN_GetRxMessage(CAN_HandleTypeDef *hcan, uint32_t RxFifo, CAN_RxHeaderTypeDef *pHeader, uint8_t aData[])
```

该函数帮你从fifo邮箱寄存器中读取当前信息，比如读取id，rtr，dlc，rdl/hr的数据等。并且设置RFOMx释放fifo邮箱。

```
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
  CAN_RxHeaderTypeDef   RxHeader;
  if(HAL_CAN_GetRxMessage(hcan, CAN_RX_FIFO0, &RxHeader, CanReceiveData) != HAL_OK)
  {
    Error_Handler();            //错误状濁回调函敿
  }
}
```

就完成了can的通信了。

VESC上can的解析过程

首先看他是用stm32的，且用的是can1

```
void comm_can_set_baud(CAN_BAUD baud) {
  switch (baud) {
  case CAN_BAUD_125K: set_timing(15, 14, 4); break;
  case CAN_BAUD_250K: set_timing(7, 14, 4); break;
  case CAN_BAUD_500K: set_timing(5, 9, 2); break;
  case CAN_BAUD_1M: set_timing(2, 9, 2); break;
  default: break;
  }
}
```

那么找到can1的时钟APB1的外设时钟是42M。42M/3=14M,故CAN的时钟是14M的。14M/(1+(9+1)+(2+1))=1M=1/Tq故波特率为1M是没错的。

查找comm_can_set_baud，得到 `comm_can_set_baud(conf->can_baud_rate);` 继续寻找

```
void conf_general_get_default_app_configuration(app_configuration *conf) {
  memset(conf, 0, sizeof(app_configuration));
  conf->controller_id = APPCONF_CONTROLLER_ID;
  conf->timeout_msec = APPCONF_TIMEOUT_MSEC;
  conf->timeout_brake_current = APPCONF_TIMEOUT_BRAKE_CURRENT;
  conf->send_can_status = APPCONF_SEND_CAN_STATUS;
  conf->send_can_status_rate_hz = APPCONF_SEND_CAN_STATUS_RATE_HZ;
  conf->can_baud_rate = APPCONF_CAN_BAUD_RATE;
```

```
#ifndef APPCONF_CAN_BAUD_RATE
#define APPCONF_CAN_BAUD_RATE       CAN_BAUD_500K
```

一看默认的是用500K的波特率。

CAN 信号线

将 CAN 信号线连接到控制板接收 CAN 控制指令，CAN 总线比特率为 1Mbps。

比较遗憾的是，2006的波特率是用1M的，看看之后能怎么改把。把vesc的也改成1M的

```
void comm_can_init(void);
void comm_can_set_baud(CAN_BAUD baud);
void comm_can_transmit_eid(uint32_t id, uint8_t *data, uint8_t len);
void comm_can_transmit_sid(uint32_t id, uint8_t *data, uint8_t len);
void comm_can_set_sid_rx_callback(void (*p_func)(uint32_t id, uint8_t *data, uint8_t len));
void comm_can_send_buffer(uint8_t controller_id, uint8_t *data, unsigned int len, bool send);
void comm_can_set_duty(uint8_t controller_id, float duty);
void comm_can_set_current(uint8_t controller_id, float current);
void comm_can_set_current_brake(uint8_t controller_id, float current);
void comm_can_set_rpm(uint8_t controller_id, float rpm);
void comm_can_set_pos(uint8_t controller_id, float pos);
void comm_can_set_current_rel(uint8_t controller_id, float current_rel);
void comm_can_set_current_brake_rel(uint8_t controller_id, float current_rel);
can_status_msg *comm_can_get_status_msg_index(int index);
can_status_msg *comm_can_get_status_msg_id(int id);
```

想知道他的标识符定义，看发送这里，这里用的是eid发送也就是说按扩展标识符发送

```
void comm_can_transmit_sid(uint32_t id, uint8_t *data, uint8_t len);
```

标准id的并没调用

```
void comm_can_transmit_eid(uint32_t id, uint8_t *data, uint8_t len) {
  if (len > 8) {
    len = 8;
  }

#if CAN_ENABLE
  CANTxFrame txmsg;
  txmsg.IDE = CAN_IDE_EXT;
  txmsg.EID = id;
  txmsg.RTR = CAN_RTR_DATA;
  txmsg.DLC = len;
  memcpy(txmsg.data8, data, len);

  chMtxLock(&can_mtx);
  canTransmit(&CAND1, CAN_ANY_MAILBOX, &txmsg, MS2ST(20));
  chMtxUnlock(&can_mtx);

#else
  (void)id;
  (void)data;
  (void)len;
#endif
}
```

```
typedef struct {
  struct {
    uint8_t               DLC:4;          /**< @brief Data length.        */
    uint8_t               RTR:1;          /**< @brief Frame type.         */
    uint8_t               IDE:1;          /**< @brief Identifier type.    */
  };
  union {
    struct {
      uint32_t            SID:11;         /**< @brief Standard identifier.*/
    };
    struct {
      uint32_t            EID:29;         /**< @brief Extended identifier.*/
    };
  };
  union {
    uint8_t               data8[8];       /**< @brief Frame data.         */
    uint16_t              data16[4];      /**< @brief Frame data.         */
    uint32_t              data32[2];      /**< @brief Frame data.         */
  };
} CANTxFrame;
```

改类型定义了CAN的标识符和DLC,RTR,IDE,还有一个共用体来存放8个字节。所以前面的赋值和最后的memcpy好理解

```
#define CAN_ANY_MAILBOX                 0
```

用的零号邮箱

```
msg_t canTransmit(CANDriver *canp,
                  canmbx_t mailbox,
                  const CANTxFrame *ctfp,
                  systime_t timeout) {

  osalDbgCheck((canp != NULL) && (ctfp != NULL) &&
               (mailbox <= (canmbx_t)CAN_TX_MAILBOXES));

  osalSysLock();
  osalDbgAssert((canp->state == CAN_READY) || (canp->state == CAN_SLEEP),
                "invalid state");
  /*lint -save -e9007 [13.5] Right side is supposed to be pure.*/
  while ((canp->state == CAN_SLEEP) || !can_lld_is_tx_empty(canp, mailbox))
  /*lint -restore*/
   msg_t msg = osalThreadEnqueueTimeoutS(&canp->txqueue, timeout);
    if (msg != MSG_OK) {
      osalSysUnlock();
      return msg;
    }
  }
  can_lld_transmit(canp, mailbox, ctfp);
  osalSysUnlock();
  return MSG_OK;
}
```

```
  switch (mailbox) {
  case CAN_ANY_MAILBOX:
    tmbp = &canp->can->sTxMailBox[(canp->can->TSR & CAN_TS
    break;
  case 1:
    tmbp = &canp->can->sTxMailBox[0];
    break;
  case 2:
    tmbp = &canp->can->sTxMailBox[1];
    break;
  case 3:
    tmbp = &canp->can->sTxMailBox[2];
    break;
  default:
    return;
  }

  /* Preparing the message.*/
  if (ctfp->IDE)
    tir = ((uint32_t)ctfp->EID << 3) | ((uint32_t)ctfp->RT
          CAN_TI0R_IDE;
  else
    tir = ((uint32_t)ctfp->SID << 21) | ((uint32_t)ctfp->R
  tmbp->TDTR = ctfp->DLC;
  tmbp->TDLR = ctfp->data32[0];
  tmbp->TDHR = ctfp->data32[1];
  tmbp->TIR  = tir | CAN_TI0R_TXRQ;
}
```

把can报文塞到邮箱中最后txrq请求发送。

那么看来他id是外部设置的，所以查一下调用该函数的地方。找到了

app_get_configuration()->controller_id |((uint32_t)CAN_PACKET_STATUS << 8)

```
case COMM_GET_VALUES:
  ind = 0;
  send_buffer[ind++] = COMM_GET_VALUES;
  buffer_append_float16(send_buffer, mc_interface_temp_fet_filtered(), 1e1f, &ind);
  buffer_append_float16(send_buffer, mc_interface_temp_motor_filtered(), 1e1f, &ind);
  buffer_append_float32(send_buffer, mc_interface_read_reset_avg_motor_current(), 1e2f, &ind);
  buffer_append_float32(send_buffer, mc_interface_read_reset_avg_input_current(), 1e2f, &ind);
  buffer_append_float32(send_buffer, mc_interface_read_reset_avg_id(), 1e2f, &ind);
  buffer_append_float32(send_buffer, mc_interface_read_reset_avg_iq(), 1e2f, &ind);
  buffer_append_float16(send_buffer, mc_interface_get_duty_cycle_now(), 1e3f, &ind);
  buffer_append_float32(send_buffer, mc_interface_get_rpm(), 1e0f, &ind);
  buffer_append_float16(send_buffer, GET_INPUT_VOLTAGE(), 1e1f, &ind);
  buffer_append_float32(send_buffer, mc_interface_get_amp_hours(false), 1e4f, &ind);
  buffer_append_float32(send_buffer, mc_interface_get_amp_hours_charged(false), 1e4f, &ind);
  buffer_append_float32(send_buffer, mc_interface_get_watt_hours(false), 1e4f, &ind);
  buffer_append_float32(send_buffer, mc_interface_get_watt_hours_charged(false), 1e4f, &ind);
  buffer_append_int32(send_buffer, mc_interface_get_tachometer_value(false), &ind);
  buffer_append_int32(send_buffer, mc_interface_get_tachometer_abs_value(false), &ind);
  send_buffer[ind++] = mc_interface_get_fault();
  buffer_append_float32(send_buffer, mc_interface_get_pid_pos_now(), 1e6f, &ind);
  send_buffer[ind++] = app_get_configuration()->controller_id;
  commands_send_packet(send_buffer, ind);
  break;
```

发现发送的数据内就有controller_id返回。

```
conf->controller_id = APPCONF_CONTROLLER_ID;
```

```
#define APPCONF_CONTROLLER_ID        0        默认值是0
```

```
case COMM_SET_APPCONF:
  appconf = *app_get_configuration();

  ind = 0;
  appconf.controller_id = data[ind++];
```

该id可以被设置。不过以上都是发送的内容。补充：后来知道了原来他是为了用ppm信号发给1个vesc，然后几个vesc通过总线连接，然后把就可以用一个遥控器控制4个电机了的。

```
// Optionally send the duty cycles to the other ESCs seen on the CAN-bus
if (send_duty && config.multi_esc) {
  float duty = mc_interface_get_duty_cycle_now();

  for (int i = 0;i < CAN_STATUS_MSGS_TO_STORE;i++) {
    can_status_msg *msg = comm_can_get_status_msg_index(i);

    if (msg->id >= 0 && UTILS_AGE_S(msg->rx_time) < MAX_CAN_AGE) {
      comm_can_set_duty(msg->id, duty);
    }
  }
}

if (current_mode) {
  if (current_mode_brake) {
    mc_interface_set_brake_current_rel(current_rel);
```

我们是要发信息给他，所以看看他是怎么处理的

```
void comm_can_init(void) {
  for (int i = 0;i < CAN_STATUS_MSGS_TO_STORE;i++) {
    stat_msgs[i].id = -1;
  }

  rx_frame_read = 0;
  rx_frame_write = 0;

  chMtxObjectInit(&can_mtx);

  palSetPadMode(GPIOB, 8,
      PAL_MODE_ALTERNATE(GPIO_AF_CAN1) |
      PAL_STM32_OTYPE_PUSHPULL |
      PAL_STM32_OSPEED_MID1);
  palSetPadMode(GPIOB, 9,
      PAL_MODE_ALTERNATE(GPIO_AF_CAN1) |
      PAL_STM32_OTYPE_PUSHPULL |
      PAL_STM32_OSPEED_MID1);

  canStart(&CANDx, &cancfg);

  chThdCreateStatic(cancom_read_thread_wa, sizeof(cancom_read_thread_wa), NORMALPRIO + 1,
      cancom_read_thread, NULL);
  chThdCreateStatic(cancom_status_thread_wa, sizeof(cancom_status_thread_wa), NORMALPRIO,
      cancom_status_thread, NULL);
  chThdCreateStatic(cancom_process_thread_wa, sizeof(cancom_process_thread_wa), NORMALPRIO,
      cancom_process_thread, NULL);
}
```

```
static THD_FUNCTION(cancom_process_thread, arg) {
```

```
  unsigned int rxbuf_len;
  unsigned int rxbuf_ind;
  uint8_t crc_low;
  uint8_t crc_high;
  bool commands_send;

  for(;;) {
    chEvtWaitAny((eventmask_t) 1);

    while (rx_frame_read != rx_frame_write) {
      CANRxFrame rxmsg = rx_frames[rx_frame_read++];

      if (rxmsg.IDE == CAN_IDE_EXT) {
        uint8_t id = rxmsg.EID & 0xFF;
        CAN_PACKET_ID cmd = rxmsg.EID >> 8;
        can_status_msg *stat_tmp;

        if (id == 255 || id == app_get_configuration()->controller_id) {
          switch (cmd) {
        }

        switch (cmd) {
      } else {
        if (sid_callback) {
          sid_callback(rxmsg.SID, rxmsg.data8, rxmsg.DLC);
        }
      }

      if (rx_frame_read == RX_FRAMES_SIZE) {
        rx_frame_read = 0;
      }
    }
```

为了看明白上面这个rx_frame_read和write是啥(因为vesc并不需要写can啊。。而且也没必要相等才能用)

于是查找两个词。得到

```
static THD_FUNCTION(cancom_read_thread, arg) {
  (void)arg;
  chRegSetThreadName("CAN");

  event_listener_t el;
  CANRxFrame rxmsg;

  chEvtRegister(&CANDx.rxfull_event, &el, 0);

  while(!chThdShouldTerminateX()) {
    if (chEvtWaitAnyTimeout(ALL_EVENTS, MS2ST(10)) == 0) {
      continue;
    }

    msg_t result = canReceive(&CANDx, CAN_ANY_MAILBOX, &rxmsg, TIME_IMMEDIATE);

    while (result == MSG_OK) {
      rx_frames[rx_frame_write++] = rxmsg;
      if (rx_frame_write == RX_FRAMES_SIZE) {
        rx_frame_write = 0;
      }

      chEvtSignal(process_tp, (eventmask_t) 1);

      result = canReceive(&CANDx, CAN_ANY_MAILBOX, &rxmsg, TIME_IMMEDIATE);
    }
  }

  chEvtUnregister(&CANDx.rxfull_event, &el);
}
```

这是接收线程。每次循环执行1个10ms的延迟。也就是说10ms循环一次？？

然后调用CAN_receive，这个接收比较平凡...内部函数就是参数诊断，然后锁上，对fifo的报文收取，然后解锁。

```
if (mailbox == CAN_ANY_MAILBOX) {
  if ((canp->can->RF0R & CAN_RF0R_FMP0) != 0)
    mailbox = 1;
  else if ((canp->can->RF1R & CAN_RF1R_FMP1) != 0)
    mailbox = 2;
  else {
    /* Should not happen, do nothing.*/
    return;
  }
}
```

比如，因为使用的是任意FIFO，所以他一次找哪个哪个fifo不是空的就读哪个的。否则都空就返回。

```
  rir  = canp->can->sFIFOMailBox[0].RIR;
  rdtr = canp->can->sFIFOMailBox[0].RDTR;
  crfp->data32[0] = canp->can->sFIFOMailBox[0].RDLR;
  crfp->data32[1] = canp->can->sFIFOMailBox[0].RDHR;

  /* Releases the mailbox.*/
  canp->can->RF0R = CAN_RF0R_RFOM0;

  /* If the queue is empty re-enables the interrupt in order to generat
     events again.*/
  if ((canp->can->RF0R & CAN_RF0R_FMP0) == 0)
    canp->can->IER |= CAN_IER_FMPIE0;
  break;
```

然后对报文的RIR,RDTR读取(包含时间戳，DLC和过滤器编码序号)，数据的高低(各4字节，共8各字节)读取。然后释放fifo对fifo的报文计数-1。防止其溢出。然后启动中断。如果接收成功就进入死循环一直接收，直到失败后停止10ms再继续尝试接收。

```
  /* Decodes the various fields in the RX frame.*/
  crfp->RTR = (rir & CAN_RI0R_RTR) >> 1;
  crfp->IDE = (rir & CAN_RI0R_IDE) >> 2;
  if (crfp->IDE)
    crfp->EID = rir >> 3;
  else
    crfp->SID = rir >> 21;
  crfp->DLC = rdtr & CAN_RDT0R_DLC;
  crfp->FMI = (uint8_t)(rdtr >> 8);
  crfp->TIME = (uint16_t)(rdtr >> 16);
}
```

然后对RIR寄存器的内容进行解码。查手册有

位31:21	**STID[10:0]/EXID[28:18]:** 标准标识符或扩展标识符 (Standard identifier or extended identifier) 依据IDE位的内容，这些位或是标准标识符，或是扩展身份标识的高字节。
位20:3	**EXID[17:0]:** 扩展标识符 (Extended identifier) 扩展标识符的低字节。
位2	**IDE:** 标识符选择 (Identifier extension) 该位决定接收邮箱中报文使用的标识符类型 0：使用标准标识符； 1：使用扩展标识符。

故RTR要右移1位，IDE右移2位，又知道了右移3位是EID，右移21位是RTR高11位为STID。

位1	**RTR:** 远程发送请求 (Remote transmission request) 0：数据帧； 1：远程帧。
位0	保留位。

于是接收到的信息就放在了 `rx_frames[rx_frame_write++] = rxmsg;` 中，并且write+1了，所以原来write是指用户写进去的不是指vesc写的。所以read就是指vesc当前处理到哪里了。消息队列一共100大小。

回到处理这里。

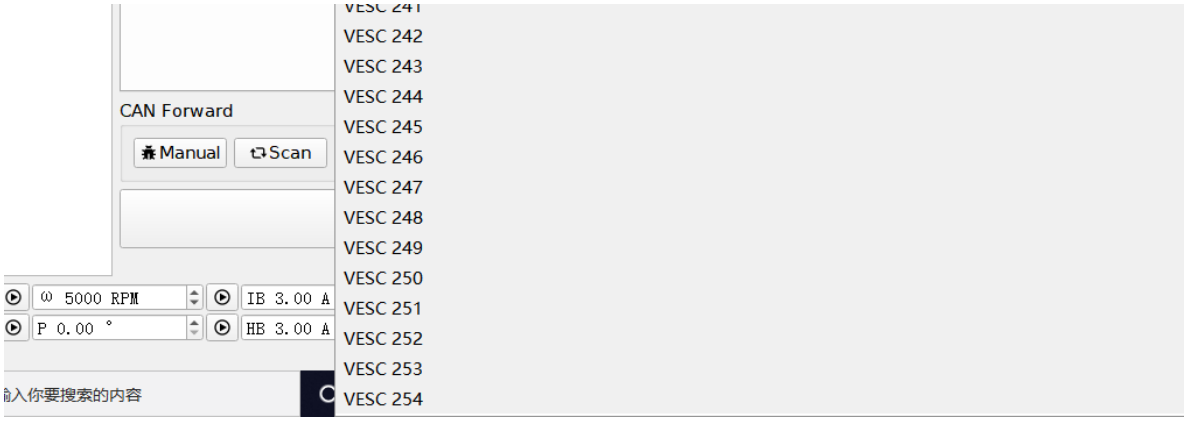
```
for(;;) {
  chEvtWaitAny((eventmask_t) 1);

  while (rx_frame_read != rx_frame_write) {
    CANRxFrame rxmsg = rx_frames[rx_frame_read++];

    if (rxmsg.IDE == CAN_IDE_EXT) {
      uint8_t id = rxmsg.EID & 0xFF;
      CAN_PACKET_ID cmd = rxmsg.EID >> 8;
      can_status_msg *stat_tmp;

      if (id == 255 || id == app_get_configuration()->controller_id) {
        switch (cmd) {
        }

        switch (cmd) {
      } else {
        if (sid_callback) {
          sid_callback(rxmsg.SID, rxmsg.data8, rxmsg.DLC);
        }
      }

      if (rx_frame_read == RX_FRAMES_SIZE) {
        rx_frame_read = 0;
      }
    }
  }
}
```

当有新消息来的时候，就读取下一个帧，

如果是扩展帧就进行消息处理，非扩展帧就调用回调，这个回调还没被赋值过，很奇怪，应该是还没用到的。

扩展id为29位。vesc使用扩展id的第8位作为真正的id。

这从vesctool中can的手动连接也可以看出来。



最多连接0-254id。大概是2个字节的宽度。

故写了id=eid&0xff。

然后cmd指令标志为eid的高位(从第8位起)。故cmd=eid>>8;丢掉低8位

接下来一个if判断id是否等于255或者是controller_id。即刚才说的2字节宽的id位应该有256个，这里只有0-254，所以这个255应该是用来指定所有vesc设备的。即你发指令的时候发了0-254这255个之1表示指定某个vesc进行控制，而255是指所有的。

然后就是报文和id匹配或者是255的时候就进行指令的执行。

指令和uart不同了，在这里。

```
typedef enum {
  CAN_PACKET_SET_DUTY = 0,
  CAN_PACKET_SET_CURRENT,
  CAN_PACKET_SET_CURRENT_BRAKE,
  CAN_PACKET_SET_RPM,
  CAN_PACKET_SET_POS,
  CAN_PACKET_FILL_RX_BUFFER,
  CAN_PACKET_FILL_RX_BUFFER_LONG,
  CAN_PACKET_PROCESS_RX_BUFFER,
  CAN_PACKET_PROCESS_SHORT_BUFFER,
  CAN_PACKET_STATUS,
  CAN_PACKET_SET_CURRENT_REL,
  CAN_PACKET_SET_CURRENT_BRAKE_REL,
  CAN_PACKET_SET_CURRENT_HANDBRAKE,
  CAN_PACKET_SET_CURRENT_HANDBRAKE_REL
} CAN_PACKET_ID;
```

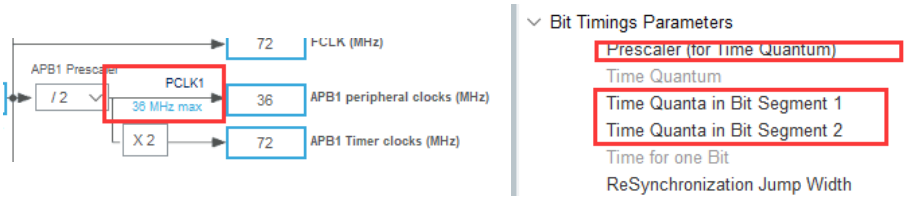
实践代码

常用函数

- HAL_CAN_ConfigFilter
- HAL_CAN_AddTxMessage
- HAL_CAN_GetRxMessage
- HAL_CAN_Start
- HAL_CAN_ActivateNotification(hcan, CAN_IT_RX_FIFO0_MSG_PENDING)使能中断(之前只是配置中断)

代码模板

设置比特率



比特率=${\frac{P_{CLK1}/Prescaler}{1+S1+S2}}$ ，比如PCKL1=36MHz，则Pre=3， S1=8,S2=3，则波特率为1M.

选择模式或功能

测试的时候可以用环回模式。

选择中断

代码

在can初始化后： 设置滤波器

```
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();
  }
```

发送函数

```
void Can_Transmit(CAN_HandleTypeDef *hcanx,uint8_t adata[],uint8_t len)
{
    CAN_TxHeaderTypeDef TxMessage;

    TxMessage.DLC=(uint8_t)len;
    TxMessage.RTR=CAN_RTR_DATA;
    TxMessage.IDE=CAN_ID_EXT;
    TxMessage.ExtId=0x00000001;

    if(HAL_CAN_AddTxMessage(hcanx,&TxMessage,adata,(uint32_t*)&addr)!=HAL_OK)
    {
         Error_Handler();        //如果CAN信息发誱失败则进入死循玿
    }
}
```

接收中断

```
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
    CAN_RxHeaderTypeDef   RxHeader;
    if(HAL_CAN_GetRxMessage(hcan, CAN_RX_FIFO0, &RxHeader, get) != HAL_OK)
  {
    Error_Handler();
  }
}
```

遇到的错误

1. 初始化失败，

```
void MX_CAN_Init(void)
{

  hcan.Instance = CAN1;
  hcan.Init.Prescaler = 3;
  hcan.Init.Mode = CAN_MODE_LOOPBACK;
  hcan.Init.SyncJumpWidth = CAN_SJW_1TQ;
  hcan.Init.TimeSeg1 = CAN_BS1_8TQ;
  hcan.Init.TimeSeg2 = CAN_BS2_3TQ;
  hcan.Init.TimeTriggeredMode = DISABLE;
  hcan.Init.AutoBusOff = DISABLE;
  hcan.Init.AutoWakeUp = DISABLE;
  hcan.Init.AutoRetransmission = DISABLE;
  hcan.Init.ReceiveFifoLocked = DISABLE;
  hcan.Init.TransmitFifoPriority = DISABLE;
  if (HAL_CAN_Init(&hcan) != HAL_OK)
  {
    Error_Handler();
  }
```

进入Error_Handler，

原因是：将把CAN_RX，和CAN_TX，接到收发器上，或者接上收发器后收发器没上电，或者电压不对(3.3V/5V)

解决：把接收器上电，或者上拉引脚。

2. 进入硬件错误，

```
void Can_Transmit(CAN_HandleTypeDef *hcanx,uint8_t adata[],uint8_t len)
{
  CAN_TxHeaderTypeDef TxMessage;

  TxMessage.DLC=(uint8_t)len;
  TxMessage.RTR=CAN_RTR_DATA;
  TxMessage.IDE=CAN_ID_EXT;
  TxMessage.ExtId=0x00000001;

  if(HAL_CAN_AddTxMessage(hcanx,&TxMessage,adata,(uint32_t*)CAN_TX_MAILBOX0)!=HAL_OK)
  {
    Error_Handler();       //如果CAN信息发酰失败则进入死循珏
  }
}
```

在F4中这样写是没错的。但是在103上就进入HardFault硬件错误死循环中。一般硬件错误是读写了非法内存，比如地址乱值，或数组越界。然后

```
if(HAL_CAN_AddTxMessage(hcanx,&TxMessage,adata,(uint32_t*)&addr)!=HAL_OK)
```

自己搞个变量接住就好了，返回发送的邮箱号

```
@param  pTxMailbox pointer to a variable where the function will return
        the TxMailbox used to store the Tx message.
        This parameter can be a value of @arg CAN Tx Mailboxes.
```

他的意思是这个值可以是以下几个。

```
#define CAN_TX_MAILBOX0            (0x00000001U)  /*!< Tx Mailbox 0  */
#define CAN_TX_MAILBOX1            (0x00000002U)  /*!< Tx Mailbox 1  */
#define CAN_TX_MAILBOX2            (0x00000004U)  /*!< Tx Mailbox 2  */
```

但不是说可以把这几个丢进去，因为按第一个写法。

把0x00000001u丢进去了。会被在这个地址上写用哪个邮箱号发的。但这个地址是不能写的地址，写了就报错。

所以不是这样用的，而是自己弄个变量去接。

3. 有一个坑就是，HAL库的uart，spi，i2c等的buff，对于数组都是数组低位为先行位且是高位。

也就是说data[]={0x00,0x01,0x02}是先发0x00，再发0x01，且0x00是高位即这串数据是0x000102.

但是HAL库的can不是，对于tx，

```
: data[8]={0x20,0x4e,0x00,0x00,0x00,0x00,0x00,0x00};
```

ESR	0x00800033	unsigned int
BTR	0x00270002	unsigned int
RESERVED0	0x40006420	unsigned int[88]
sTxMailBox	0x40006580	struct <untagged>[3]
[0]	0x40006580	struct <untagged>
TIR	0x00000004	unsigned int
TDTR	0x00000008	unsigned int
TDLR	0x00004E20	unsigned int
TDHR	0x00000000	unsigned int
[1]	0x40006590	struct <untagged>
[2]	0x400065A0	struct <untagged>
sFIFOMailBox	0x400065B0	struct <untagged>[2]

可以发现，他先发送了buff的高位，且高位为数据高位，即

对于上面的数据data[]={0x00,0x01,0x02} 用can发送表示的是0x020100；这造成我在用can调vesc的时候。我看其源码

```
case CAN_PACKET_SET_DUTY:
  ind = 0;
  mc_interface_set_duty(buffer_get_float32(rxmsg.data8, 1e5f, &ind));
  timeout_reset();
  break;
```

为了设置其占空比0-1.其比例为1e5就是说发送100000D表示占空比1，发送20000D表示0.2占空比(20%)，而20000D=4e20H，所以我以开始发送buff为00 00 00 00 00 00 4e 20。电机不转，身边没有示波器，看can句柄，其ESR寄存器无报错，STATUS无报错，ERROR_CODE无报错，确认CAN无误。转而确定是否为can报文出错。

再看源码

```
int32_t buffer_get_int32(const uint8_t *buffer, int32_t *index) {
  int32_t res = ((uint32_t) buffer[*index]) << 24 |
          ((uint32_t) buffer[*index + 1]) << 16 |
          ((uint32_t) buffer[*index + 2]) << 8 |
          ((uint32_t) buffer[*index + 3]);
  *index += 4;
  return res;
```

其只用了buff的低4位。猜测为上面的原因，即报文高低位问题。果然，发送buff装20 4e 00 00 00 00 00 00 电机就飞起来了。但是更坑的是，按这个发并不是0.2的占空比，而是1，为啥呢，因为溢出了。。。看上面的图，应该发成00 00 4e 20 00 00 00 00才行，因为他把低位解释成float的高位。

4.

```
if(HAL_CAN_AddTxMessage(hvesc->hcann,&TxMessage,txbuf,&txmailbox)!=HAL_OK){
    //Error_Handler();
}
```

这条语句当你两次发送之间不间隔的时候是会发送失败的，不把errorhandle注释了会报错。比如这样

```
 for(int i=0;i<4;++i){
   VESC_CAN_SET_DUTY(&vesc[i],(chassis.wheel[i].roll.ctrl[wtrcfg_PID_v].output));
```

一个for里面2次是每间隔的。所以为了避免应该加个小循环延时，或者说受用all发送给所有电机把。

5. 不要在初始化模式下进行过滤器初值的设置，但必须在它处在非激活状态下完成(相应的FACT位为0)。而过滤器的位宽和模式的设置，则必须在初始化模式中进入正常模式前完成。
6. 如果邮箱满了，先检擦是不是线接错了，查看ESR寄存器，估计就是离线了。如果是应该是接收发送计数溢出。可以确定就是代码没问题，是硬件接错了，包括CANHL，CANRXTX，终端电阻等。