

第三十二章 CAN 通讯实验

本章我们将向大家介绍如何使用 STM32F4 自带的 CAN 控制器来实现两个开发板之间的 CAN 通讯，并将结果显示在 TFTLCD 模块上。本章分为如下几个部分：

- 32.1 CAN 简介
- 32.2 硬件设计
- 32.3 软件设计
- 32.4 下载验证

32.1 CAN 简介

CAN 是 Controller Area Network 的缩写（以下称为 CAN），是 ISO 国际标准化的串行通信协议。在当前的汽车产业中，出于对安全性、舒适性、方便性、低公害、低成本的要求，各种各样的电子控制系统被开发了出来。由于这些系统之间通信所用的数据类型及对可靠性的要求不尽相同，由多条总线构成的情况很多，线束的数量也随之增加。为适应“减少线束的数量”、“通过多个 LAN，进行大量数据的高速通信”的需要，1986 年德国电气商博世公司开发出面向汽车的 CAN 通信协议。此后，CAN 通过 ISO11898 及 ISO11519 进行了标准化，现在在欧洲已是汽车网络的标准协议。

现在，CAN 的高性能和可靠性已被认同，并被广泛地应用于工业自动化、船舶、医疗设备、工业设备等方面。现场总线是当今自动化领域技术发展的热点之一，被誉为自动化领域的计算机局域网。它的出现为分布式控制系统实现各节点之间实时、可靠的数据通信提供了强有力的技术支持。

CAN 控制器根据两根线上的电位差来判断总线电平。总线电平分为显性电平和隐性电平，二者必居其一。发送方通过使总线电平发生变化，将消息发送给接收方。

CAN 协议具有一下特点：

- 1) **多主控制**。在总线空闲时，所有单元都可以发送消息（多主控制），而两个以上的单元同时开始发送消息时，根据标识符（Identifier 以下称为 ID）决定优先级。ID 并不是表示发送的目的地址，而是表示访问总线的消息的优先级。两个以上的单元同时开始发送消息时，对各消息 ID 的每个位进行逐个仲裁比较。仲裁获胜（被判定为优先级最高）的单元可继续发送消息，仲裁失利的单元则立刻停止发送而进行接收工作。
- 2) **系统的柔软性**。与总线相连的单元没有类似于“地址”的信息。因此在总线上增加单元时，连接在总线上的其它单元的软硬件及应用层都不需要改变。
- 3) **通信速度较快，通信距离远**。最高 1Mbps（距离小于 40M），最远可达 10KM（速率低于 5Kbps）。
- 4) **具有错误检测、错误通知和错误恢复功能**。所有单元都可以检测错误（错误检测功能），检测出错误的单元会立即同时通知其他所有单元（错误通知功能），正在发送消息的单元一旦检测出错误，会强制结束当前的发送。强制结束发送的单元会不断反复地重新发送此消息直到成功发送为止（错误恢复功能）。
- 5) **故障封闭功能**。CAN 可以判断出错误的类型是总线上暂时的数据错误（如外部噪声等）还是持续的数据错误（如单元内部故障、驱动器故障、断线等）。由此功能，当总线上发生持续数据错误时，可将引起此故障的单元从总线上隔离出去。
- 6) **连接节点多**。CAN 总线是可同时连接多个单元的总线。可连接的单元总数理论上是没有限制的。但实际上可连接的单元数受总线上的时间延迟及电气负载的限制。降低通

信速度，可连接的单元数增加；提高通信速度，则可连接的单元数减少。

正是因为 CAN 协议的这些特点，使得 CAN 特别适合工业过程监控设备的互连，因此，越来越受到工业界的重视，并已公认为最有前途的现场总线之一。

CAN 协议经过 ISO 标准化后有两个标准：ISO11898 标准和 ISO11519-2 标准。其中 ISO11898 是针对通信速率为 125Kbps~1Mbps 的高速通信标准，而 ISO11519-2 是针对通信速率为 125Kbps 以下的低速通信标准。

本章，我们使用的是 500Kbps 的通信速率，使用的是 ISO11898 标准，该标准的物理层特征如图 32.1.1 所示：

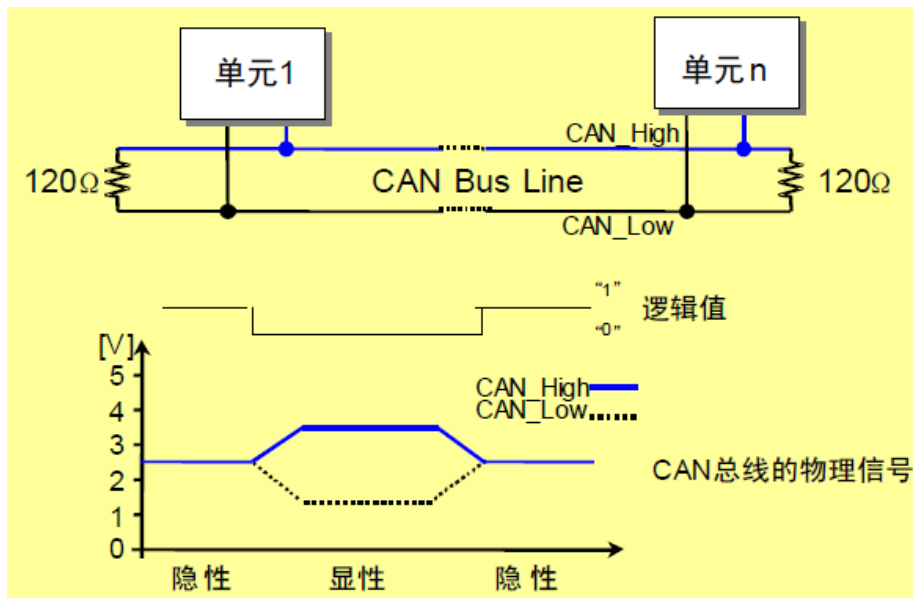


图 32.1.1 ISO11898 物理层特性

从该特性可以看出，显性电平对应逻辑 0，CAN_H 和 CAN_L 之差为 2.5V 左右。而隐性电平对应逻辑 1，CAN_H 和 CAN_L 之差为 0V。在总线上显性电平具有优先权，只要有一个单元输出显性电平，总线上即为显性电平。而隐性电平则具有包容的意味，只有所有的单元都输出隐性电平，总线上才为隐性电平（显性电平比隐性电平更强）。另外，在 CAN 总线的起止端都有一个 120Ω 的终端电阻，来做阻抗匹配，以减少回波反射。

CAN 协议是通过以下 5 种类型的帧进行的：

- 数据帧
- 遥控帧
- 错误帧
- 过载帧
- 间隔帧

另外，数据帧和遥控帧有标准格式和扩展格式两种格式。标准格式有 11 个位的标识符 (ID)，扩展格式有 29 个位的 ID。各种帧的用途如表 32.1.1 所示：

帧类型	帧用途
数据帧	用于发送单元向接收单元传送数据的帧
遥控帧	用于接收单元向具有相同 ID 的发送单元请求数据的帧
错误帧	用于当检测出错误时向其它单元通知错误的帧
过载帧	用于接收单元通知其尚未做好接收准备的帧
间隔帧	用于将数据帧及遥控帧与前面的帧分离开来的帧

表 32.1.1 CAN 协议各种帧及其用途

由于篇幅所限，我们这里仅对数据帧进行详细介绍，数据帧一般由 7 个段构成，即：

- (1) 帧起始。表示数据帧开始的段。
- (2) 仲裁段。表示该帧优先级的段。
- (3) 控制段。表示数据的字节数及保留位的段。
- (4) 数据段。数据的内容，一帧可发送 0~8 个字节的数据。
- (5) CRC 段。检查帧的传输错误的段。
- (6) ACK 段。表示确认正常接收的段。
- (7) 帧结束。表示数据帧结束的段。

数据帧的构成如图 32.1.2 所示：

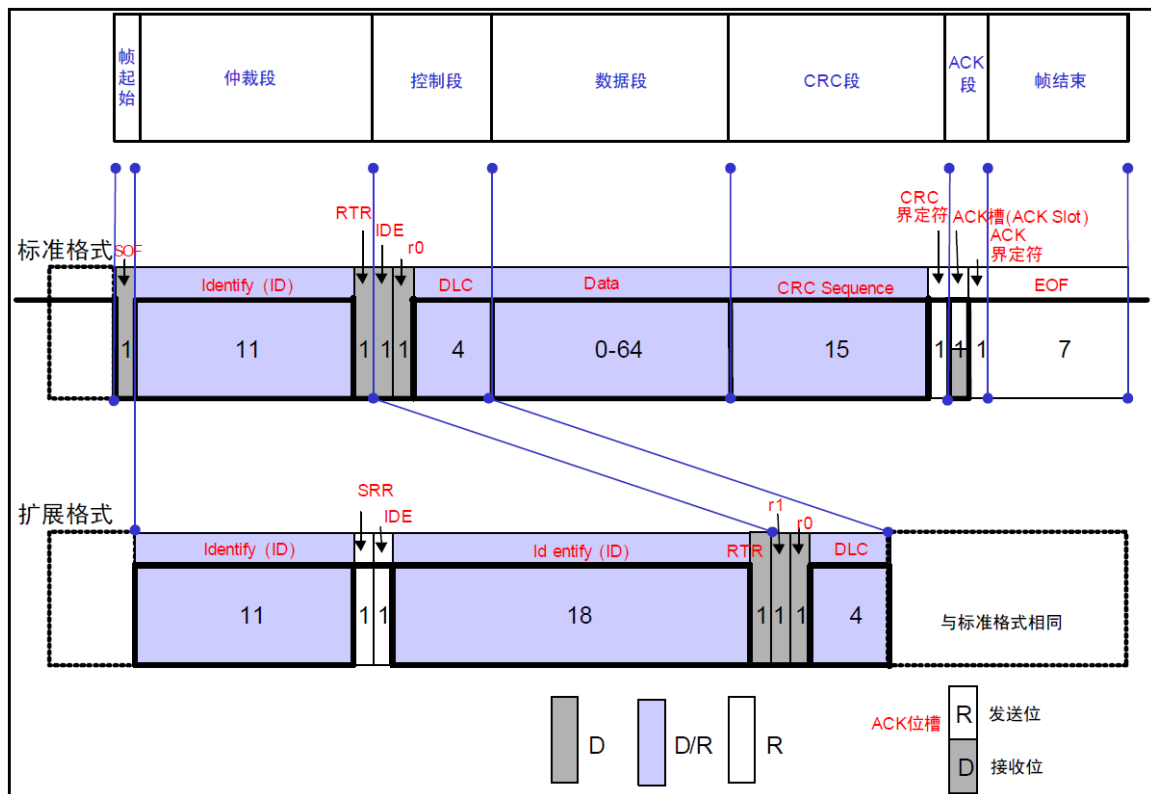


图 32.1.2 数据帧的构成

图中 D 表示显性电平，R 表示隐性电平（下同）。

帧起始，这个比较简单，标准帧和扩展帧都是由 1 个位的显性电平表示帧起始。

仲裁段，表示数据优先级的段，标准帧和扩展帧格式在本段有所区别，如图 32.1.3 所示：

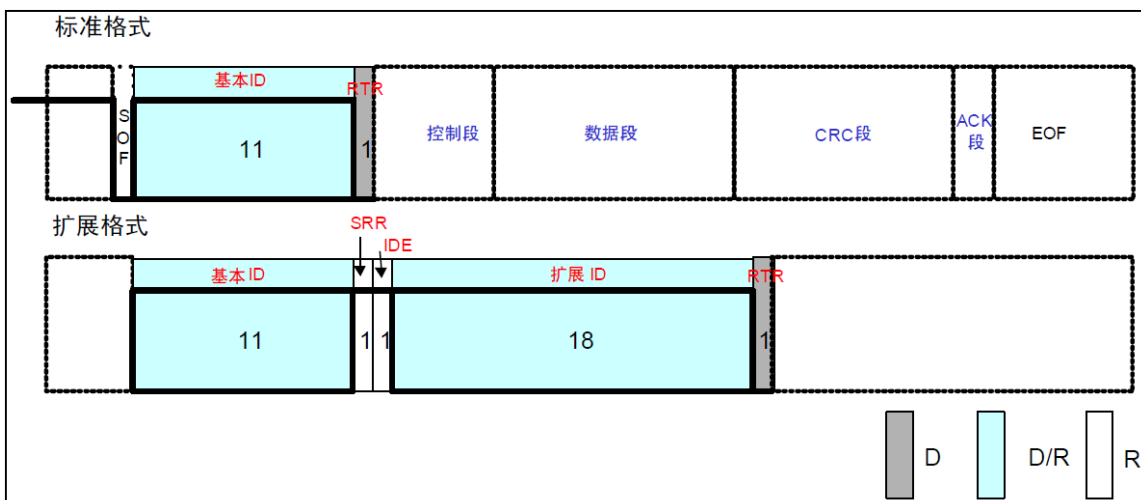


图 32.1.3 数据帧仲裁段构成

标准格式的 ID 有 11 个位。从 ID28 到 ID18 被依次发送。禁止高 7 位都为隐性（禁止设定：ID=1111111XXXX）。扩展格式的 ID 有 29 个位。基本 ID 从 ID28 到 ID18，扩展 ID 由 ID17 到 ID0 表示。基本 ID 和标准格式的 ID 相同。禁止高 7 位都为隐性（禁止设定：基本 ID=1111111XXXX）。

其中 RTR 位用于标识是否是远程帧（0，数据帧；1，远程帧），IDE 位为标识符选择位（0，使用标准标识符；1，使用扩展标识符），SRR 位为代替远程请求位，为隐性位，它代替了标准帧中的 RTR 位。

控制段，由 6 个位构成，表示数据段的字节数。标准帧和扩展帧的控制段稍有不同，如图 32.1.4 所示：

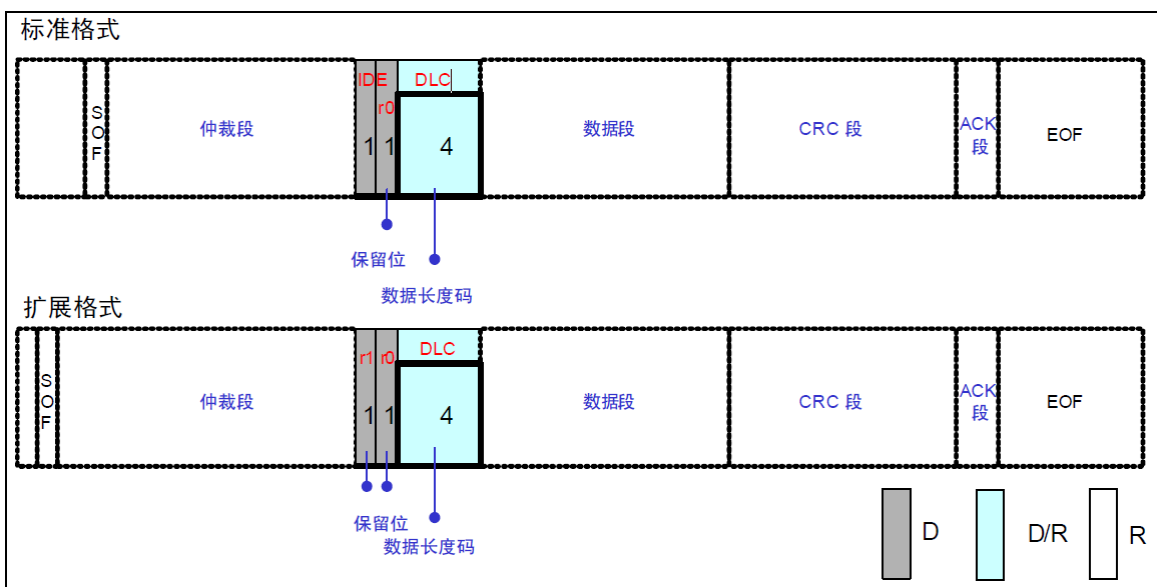


图 32.1.4 数据帧控制段构成

上图中，r0 和 r1 为保留位，必须全部以显性电平发送，但是接收端可以接收显性、隐性及任意组合的电平。DLC 段为数据长度表示段，高位在前，DLC 段有效值为 0~8，但是接收方接收到 9~15 的时候并不认为是错误。

数据段，该段可包含 0~8 个字节的数据。从最高位（MSB）开始输出，标准帧和扩展帧在

这个段的定义都是一样的。如图 32.1.5 所示：

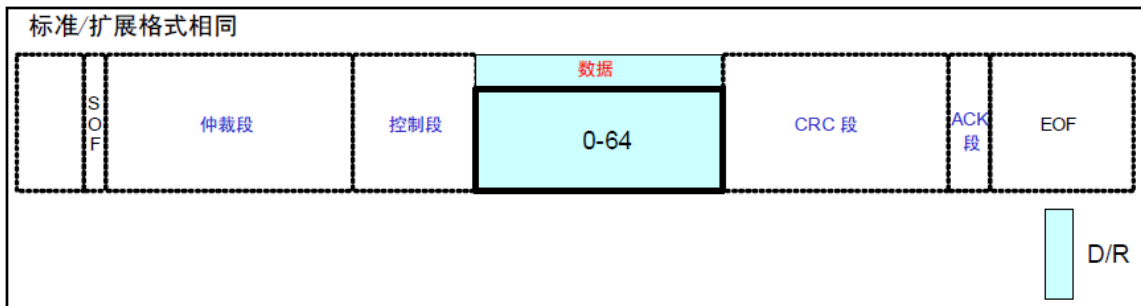


图 32.1.5 数据帧数据段构成

CRC 段，该段用于检查帧传输错误。由 15 个位的 CRC 顺序和 1 个位的 CRC 界定符（用于分隔的位）组成，标准帧和扩展帧在这个段的格式也是相同的。如图 32.1.6 所示：

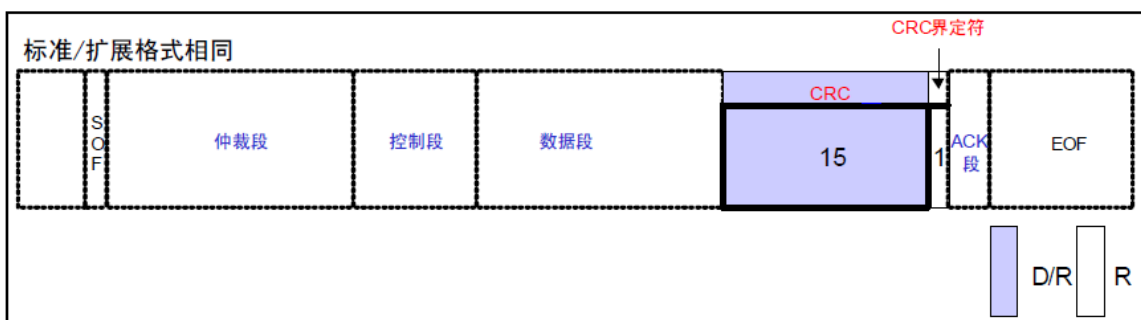


图 32.1.6 数据帧 CRC 段构成

此段 CRC 的值计算范围包括：帧起始、仲裁段、控制段、数据段。接收方以同样的算法计算 CRC 值并进行比较，不一致时会通报错误。

ACK 段，此段用来确认是否正常接收。由 ACK 槽(ACK Slot)和 ACK 界定符 2 个位组成。标准帧和扩展帧在这个段的格式也是相同的。如图 32.1.7 所示：

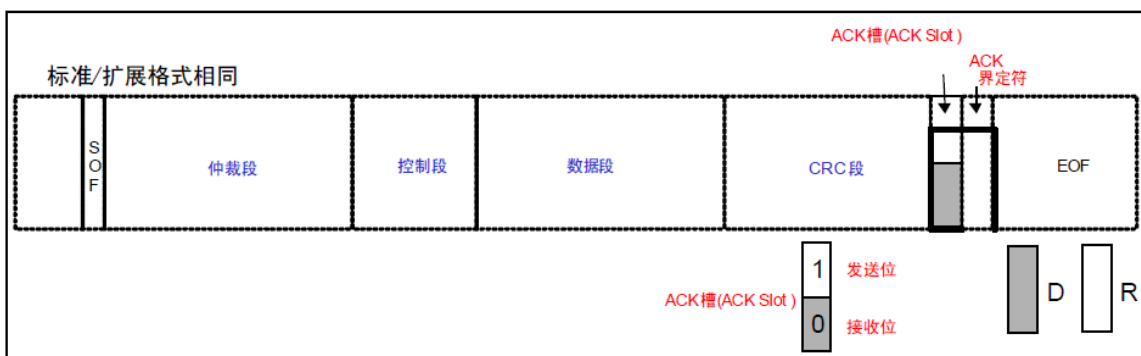


图 32.1.7 数据帧 ACK 段构成

发送单元的 ACK，发送 2 个位的隐性位，而接收到正确消息的单元在 ACK 槽（ACK Slot）发送显性位，通知发送单元正常接收结束，这个过程叫发送 ACK/返回 ACK。发送 ACK 的是在既不处于总线关闭态也不处于休眠态的所有接收单元中，接收到正常消息的单元（发送单元不发送 ACK）。所谓正常消息是指不含填充错误、格式错误、CRC 错误的消息。

帧结束，这个段也比较简单，标准帧和扩展帧在这个段格式一样，由 7 个位的隐性位组成。

至此，数据帧的 7 个段就介绍完了，其他帧的介绍，请大家参考光盘的 CAN 入门书.pdf 相关章节。接下来，我们再来看看 CAN 的位时序。

由发送单元在非同步的情况下发送的每秒钟的位数称为位速率。一个位可分为 4 段。

- 同步段 (SS)
- 传播时间段 (PTS)
- 相位缓冲段 1 (PBS1)
- 相位缓冲段 2 (PBS2)

这些段又由可称为 Time Quantum (以下称为 T_q) 的最小时间单位构成。

1 位分为 4 个段, 每个段又由若干个 T_q 构成, 这称为位时序。

1 位由多少个 T_q 构成、每个段又由多少个 T_q 构成等, 可以任意设定位时序。通过设定位时序, 多个单元可同时采样, 也可任意设定采样点。各段的作用和 T_q 数如表 32.1.2 所示:

段名称	段的作用	T_q 数	
同步段 (SS: Synchronization Segment)	多个连接在总线上的单元通过此段实现时序调整, 同步进行接收和发送的工作。由隐性电平到显性电平的边沿或由显性电平到隐性电平边沿最好出现在此段中。	1 T_q	8~25 T_q
传播时间段 (PTS: Propagation Time Segment)	用于吸收网络上的物理延迟的段。 所谓的网络的物理延迟指发送单元的输出延迟、总线上信号的传播延迟、接收单元的输入延迟。 这个段的时间为以上各延迟时间的和的两倍。	1~8 T_q	
相位缓冲段 1 (PBS1: Phase Buffer Segment 1)	当信号边沿不能被包含于 SS 段中时, 可在此段进行补偿。	1~8 T_q	
相位缓冲段 2 (PBS2: Phase Buffer Segment 2)	由于各单元以各自独立的时钟工作, 细微的时钟误差会累积起来, PBS 段可用于吸收此误差。 通过对相位缓冲段加减 SJW 吸收误差。 SJW 加大后允许误差加大, 但通信速度下降。	2~8 T_q	
再同步补偿宽度 (SJW: reSynchronization Jump Width)	因时钟频率偏差、传送延迟等, 各单元有同步误差。SJW 为补偿此误差的最大值。	1~4 T_q	

表 32.1.2 一个位各段及其作用

1 个位的构成如图 32.1.8 所示:

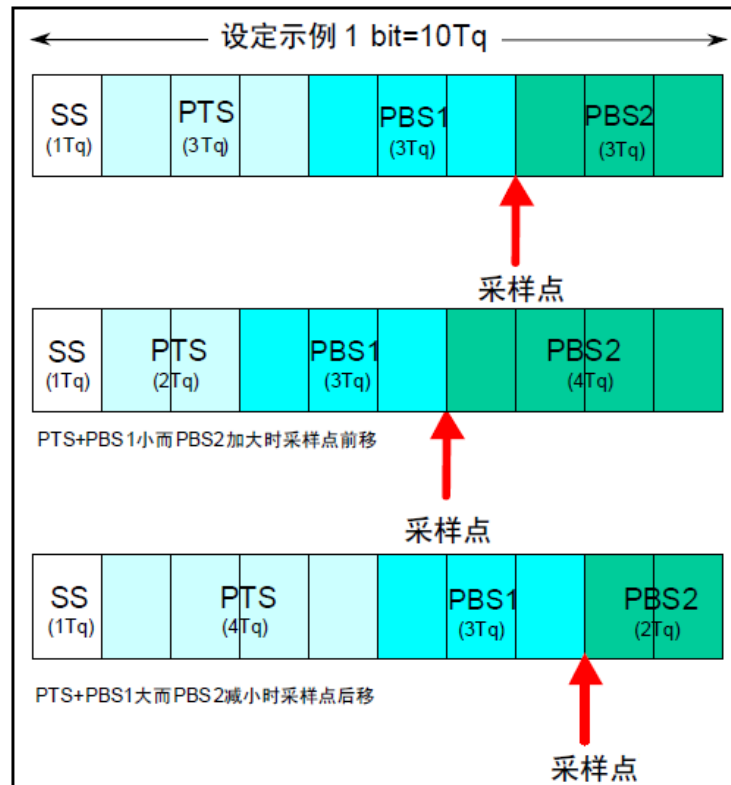


图 32.1.8 一个位的构成

上图的采样点,是指读取总线电平,并将读到的电平作为位值的点。位置在 PBS1 结束处。根据这个位时序,我们就可以计算 CAN 通信的波特率了。具体计算方法,我们等下再介绍,前面提到的 CAN 协议具有仲裁功能,下面我们来看看是如何实现的。

在总线空闲态,最先开始发送消息的单元获得发送权。

当多个单元同时开始发送时,各发送单元从仲裁段的第一位开始进行仲裁。连续输出显性电平最多的单元可继续发送。实现过程,如图 32.1.9 所示:

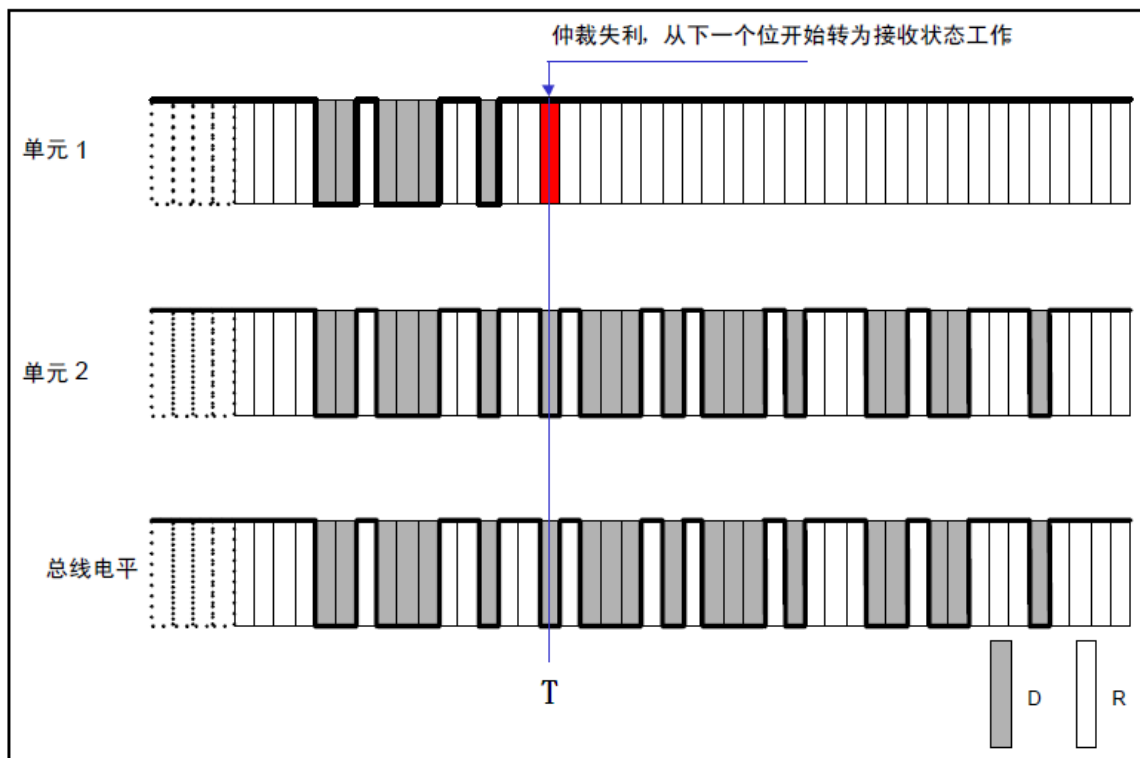


图 32.1.9 CAN 总线仲裁过程

上图中，单元 1 和单元 2 同时开始向总线发送数据，开始部分他们的数据格式是一样的，故无法区分优先级，直到 T 时刻，单元 1 输出隐性电平，而单元 2 输出显性电平，此时单元 1 仲裁失利，立刻转入接收状态工作，不再与单元 2 竞争，而单元 2 则顺利获得总线使用权，继续发送自己的数据。这就实现了仲裁，让连续发送显性电平多的单元获得总线使用权。

通过以上介绍，我们对 CAN 总线有了个大概了解（详细介绍参考光盘的：《CAN 入门书.pdf》），接下来我们介绍下 STM32F4 的 CAN 控制器。

STM32F4 自带的是 bxCAN，即基本扩展 CAN。它支持 CAN 协议 2.0A 和 2.0B。它的设计目标是，以最小的 CPU 负荷来高效处理大量收到的报文。它也支持报文发送的优先级要求（优先级特性可软件配置）。对于安全紧要的应用，bxCAN 提供所有支持时间触发通信模式所需的硬件功能。

STM32F4 的 bxCAN 的主要特点有：

- 支持 CAN 协议 2.0A 和 2.0B 主动模式
- 波特率最高达 1Mbps
- 支持时间触发通信
- 具有 3 个发送邮箱
- 具有 3 级深度的 2 个接收 FIFO
- 可变的过滤器组（28 个，CAN1 和 CAN2 共享）

在 STM32F407ZGT6 中，带有 2 个 CAN 控制器，而我们本章只用了 1 个 CAN，即 CAN1。双 CAN 的框图如图 32.1.10 所示：

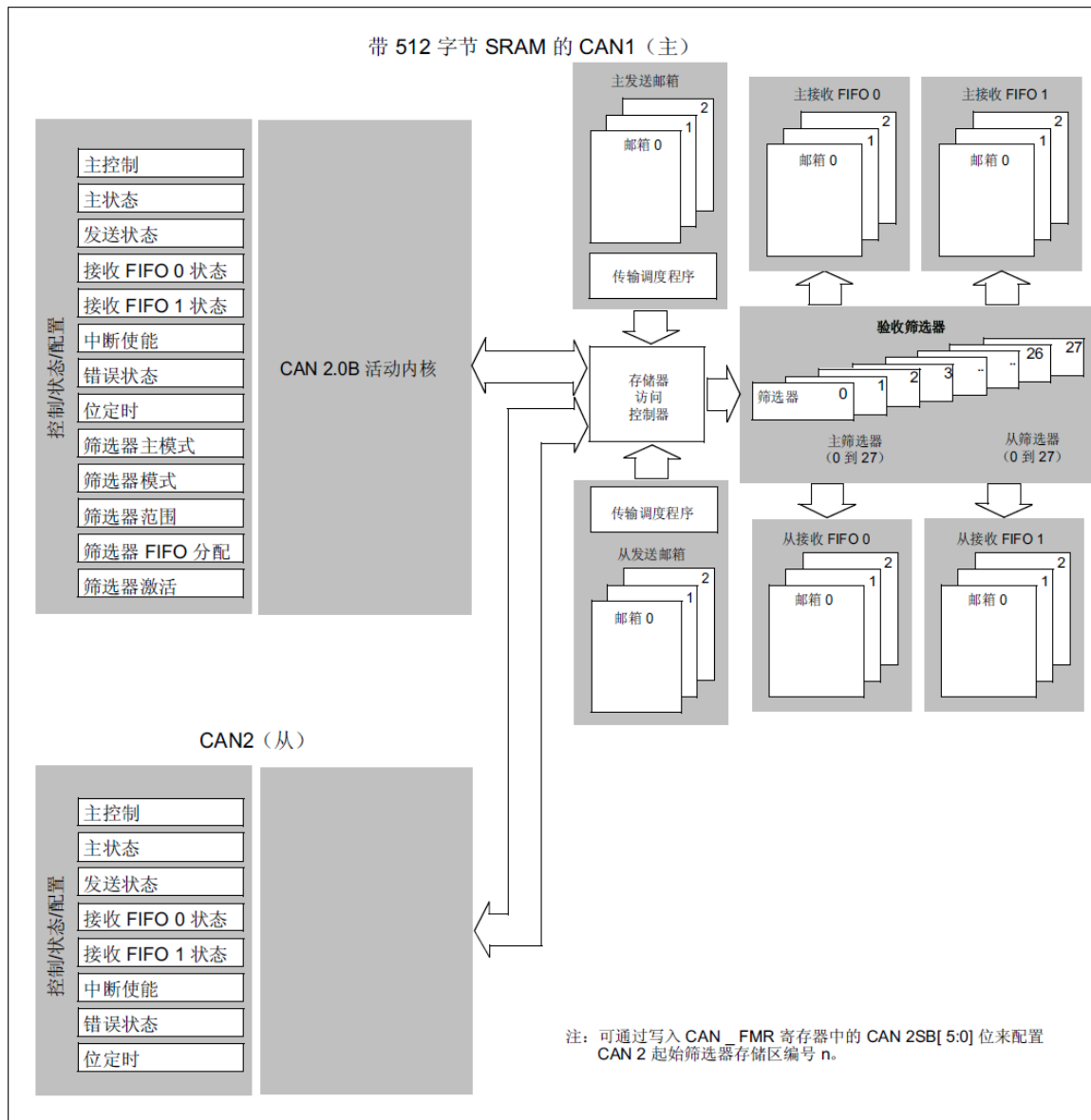


图 32.1.10 双 CAN 框图

从图中可以看出两个 CAN 都分别拥有自己的发送邮箱和接收 FIFO，但是他们共用 28 个滤波器。通过 CAN_FMR 寄存器的设置，可以设置滤波器的分配方式。

STM32F4 的标识符过滤是一个比较复杂的东东，它的存在减少了 CPU 处理 CAN 通信的开销。STM32F4 的过滤器（也称筛选器）组最多有 28 个，每个滤波器组 x 由 2 个 32 为寄存器，CAN_FxR1 和 CAN_FxR2 组成。

STM32F4 每个过滤器组的位宽都可以独立配置，以满足应用程序的不同需求。根据位宽的不同，每个过滤器组可提供：

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

在屏蔽位模式下，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”或“不用关心”处理。

而在标识符列表模式下，屏蔽寄存器也被当作标识符寄存器用。因此，不是采用一个标识

符加一个屏蔽位的方式，而是使用 2 个标识符寄存器。接收报文标识符的每一位都必须跟过滤器标识符相同。

通过 CAN_FMR 寄存器，可以配置过滤器组的位宽和工作模式，如图 32.1.11 所示：

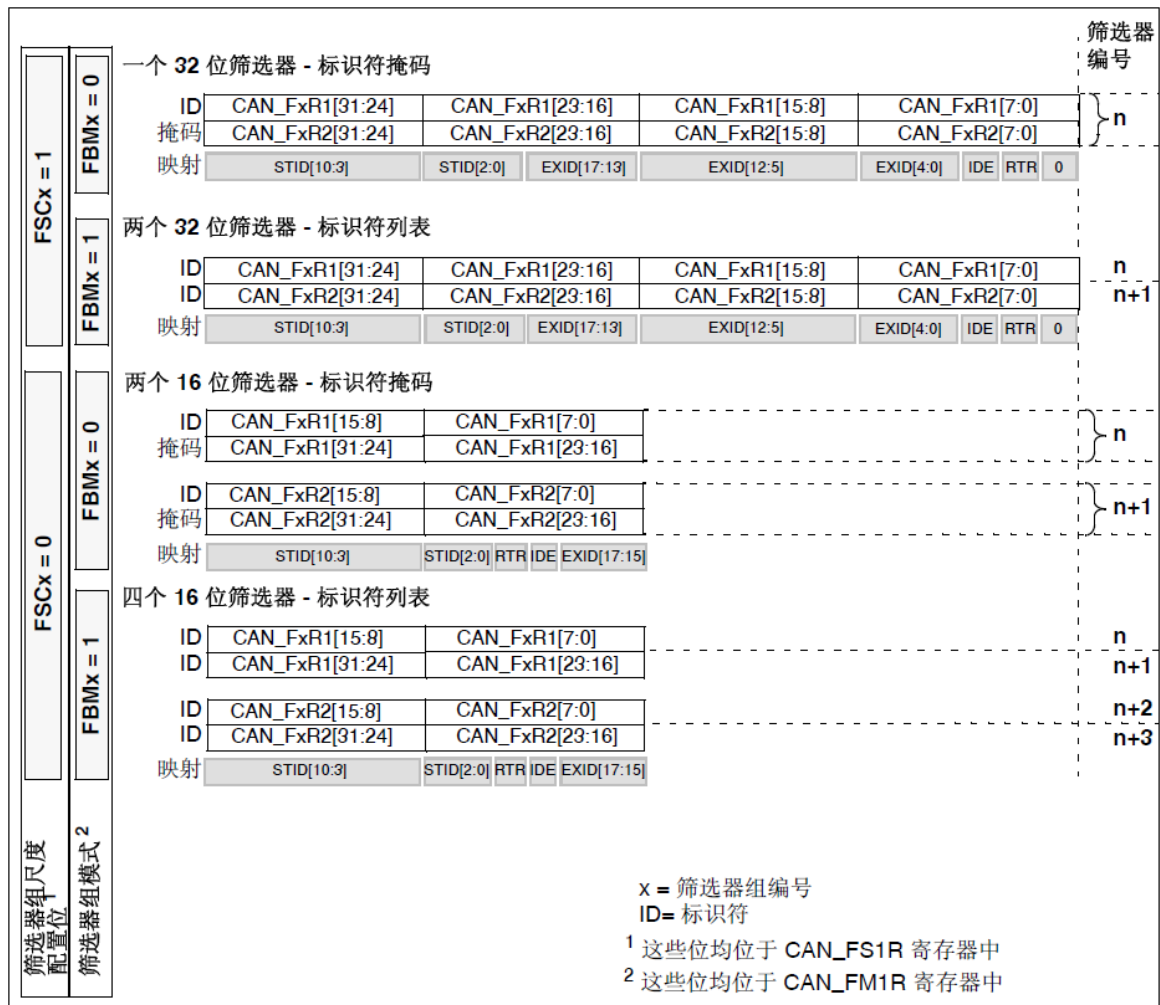


图 32.1.11 过滤器组位宽模式设置

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。

应用程序不用的过滤器组，应该保持在禁用状态。

过滤器组中的每个过滤器，都被编号为(叫做过滤器号，图 32.1.11 中的 n)从 0 开始，到某个最大数值一取决于过滤器组的模式和位宽的设置。

举个简单的例子，我们设置过滤器组 0 工作在：1 个 32 位过滤器-标识符屏蔽模式，然后设置 CAN_F0R1=0xFFFF0000，CAN_F0R2=0xFF00FF00。其中存放到 CAN_F0R1 的值就是期望收到的 ID，即我们希望收到的 ID（STID+EXTID+IDE+RTR）最好是：0xFFFF0000。而 0xFF00FF00 就是设置我们需要必须关心的 ID，表示收到的 ID，其位[31:24]和位[15:8]这 16 个位的必须和 CAN_F0R1 中对应的位一模一样，而另外的 16 个位则不关心，可以一样，也可以不一样，都认为是正确的 ID，即收到的 ID 必须是 0xFFxx00xx，才算是正确的(x 表示不关心)。

关于标识符过滤的详细介绍，请参考《STM32F4xx 中文参考手册》的 24.7.4 节（616 页）。接下来，我们看看 STM32F4 的 CAN 发送和接收的流程。

CAN 发送流程

CAN 发送流程为：程序选择 1 个空置的邮箱（TME=1）→设置标识符（ID），数据长度和发送数据→设置 CAN_TiRxR 的 TXRQ 位为 1，请求发送→邮箱挂号（等待成为最高优先级）→预定发送（等待总线空闲）→发送→邮箱空置。整个流程如图 32.1.12 所示：

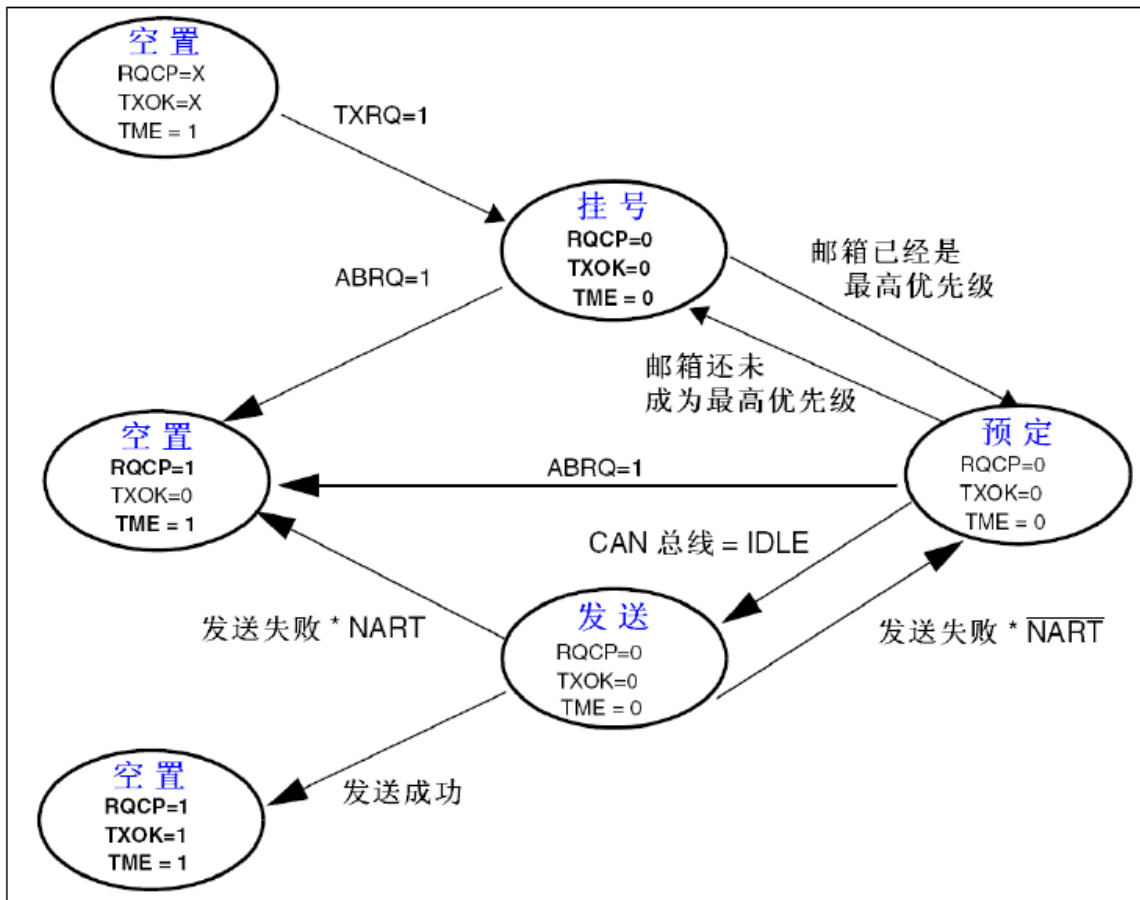


图 32.1.12 发送邮箱

上图中，还包含了很多其他处理，终止发送（ABRQ=1）和发送失败处理等。通过这个流程图，我们大致了解了 CAN 的发送流程，后面的数据发送，我们基本就是按照此流程来走。接下来再看看 CAN 的接收流程。

CAN 接收流程

CAN 接收到的有效报文，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。这里的有效报文是指那些正确被接收的（直到 EOF 都没有错误）且通过了标识符过滤的报文。前面我们知道 CAN 的接收有 2 个 FIFO，我们每个滤波器组都可以设置其关联的 FIFO，通过 CAN_FFA1R 的设置，可以将滤波器组关联到 FIFO0/FIFO1。

CAN 接收流程为：FIFO 空→收到有效报文→挂号_1（存入 FIFO 的一个邮箱，这个由硬件控制，我们不需要理会）→收到有效报文→挂号_2→收到有效报文→挂号_3→收到有效报文→溢出。

这个流程里面，我们没有考虑从 FIFO 读出报文的情况，实际情况是：我们必须在 FIFO 溢出之前，读出至少 1 个报文，否则下个报文到来，将导致 FIFO 溢出，从而出现报文丢失。每读出 1 个报文，相应的挂号就减 1，直到 FIFO 空。CAN 接收流程如图 32.1.13 所示：

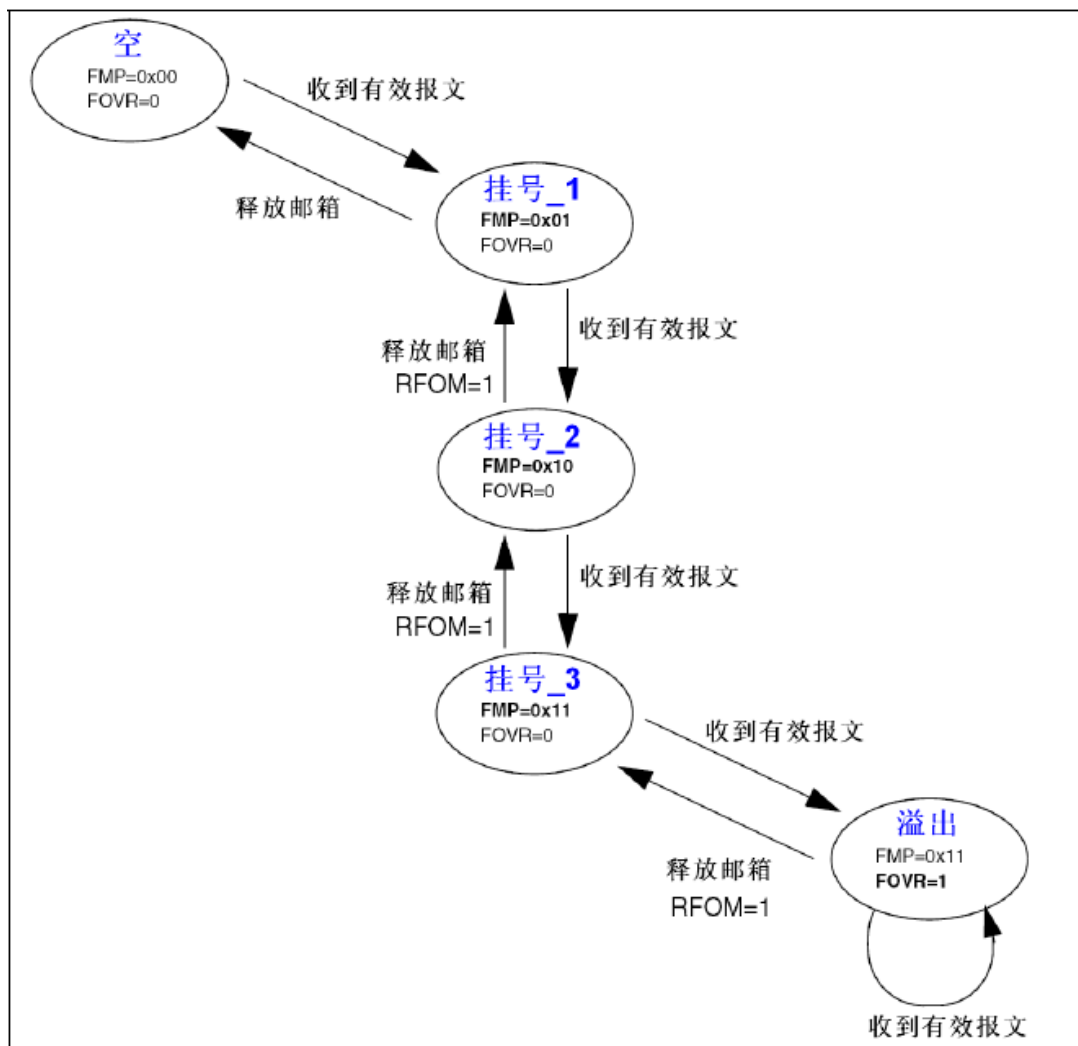


图 32.1.13 FIFO 接收报文

FIFO 接收到的报文数，我们可以通过查询 CAN_RFRxR 的 FMP 寄存器来得到，只要 FMP 不为 0，我们就可以从 FIFO 读出收到的报文。

接下来，我们简单看看 STM32F4 的 CAN 位时间特性，STM32F4 的 CAN 位时间特性和之前我们介绍的，稍有点区别。STM32F4 把传播时间段和相位缓冲段 1 (STM32F4 称之为时间段 1) 合并了，所以 STM32F4 的 CAN 一个位只有 3 段：同步段 (SYNC_SEG)、时间段 1 (BS1) 和时间段 2 (BS2)。STM32F4 的 BS1 段可以设置为 1~16 个时间单元，刚好等于我们上面介绍的传播时间段和相位缓冲段 1 之和。STM32F4 的 CAN 位时序如图 32.1.14 所示：

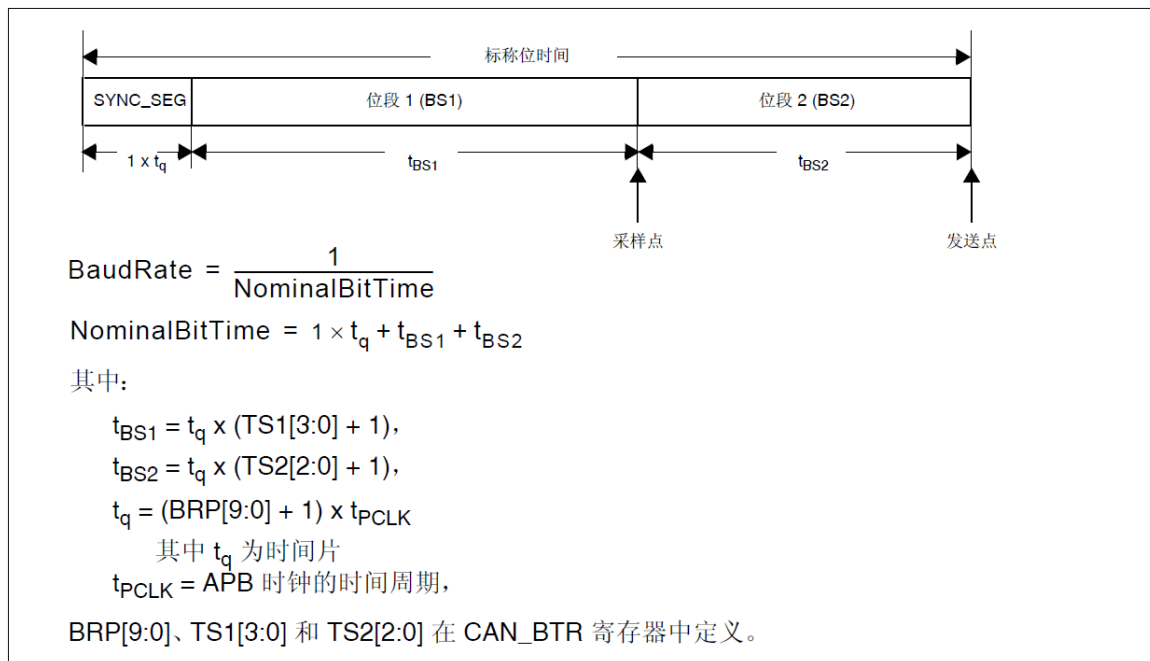


图 32.1.14 STM32F4 CAN 位时序

图中还给出了 CAN 波特率的计算公式，我们只需要知道 BS1 和 BS2 的设置，以及 APB1 的时钟频率（一般为 42Mhz），就可以方便的计算出波特率。比如设置 TS1=6、TS2=5 和 BRP=5，在 APB1 频率为 42Mhz 的条件下，即可得到 CAN 通信的波特率=42000/[(7+6+1)*6]=500Kbps。

接下来，我们介绍一下本章需要用到的一些比较重要的寄存器。首先，来看 CAN 的主控制寄存器（CAN_MCR），该寄存器各位描述如图 32.1.15：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															DBF
															rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESET	Reserved							TTCM	ABOM	AWUM	NART	RFLM	TXFP	SLEEP	INRQ
rs								rw	rw	rw	rw	rw	rw	rw	rw

图 32.1.15 寄存器 CAN_MCR 各位描述

该寄存器的详细描述，请参考《STM32F4xx 中文参考手册》24.9.2 节（625 页），这里我们仅介绍下 INRQ 位，该位用来控制初始化请求。

软件对该位清 0，可使 CAN 从初始化模式进入正常工作模式：当 CAN 在接收引脚检测到连续的 11 个隐性位后，CAN 就达到同步，并为接收和发送数据作好准备了。为此，硬件相应地对 CAN_MSR 寄存器的 INAK 位清’ 0’。

软件对该位置 1 可使 CAN 从正常工作模式进入初始化模式：一旦当前的 CAN 活动(发送或接收)结束，CAN 就进入初始化模式。相应地，硬件对 CAN_MSR 寄存器的 INAK 位置’ 1’。

所以我们在 CAN 初始化的时候，先要设置该位为 1，然后进行初始化（尤其是 CAN_BTR 的设置，该寄存器，必须在 CAN 正常工作之前设置），之后再设置该位为 0，让 CAN 进入正常工作模式。

第二个，我们介绍 CAN 位时序寄存器（CAN_BTR），该寄存器用于设置分频、Tbs1、Tbs2 以及 Tsjuw 等非常重要的参数，直接决定了 CAN 的波特率。另外该寄存器还可以设置 CAN 的工作模式，该寄存器各位描述如图 32.1.16 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SILM	LBKM	Reserved				SJW[1:0]		Res.	TS2[2:0]			TS1[3:0]			
rw	rw					rw	rw		rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						BRP[9:0]									
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31 **SILM**: 静默模式 (调试) (Silent mode (debug))

0: 正常工作
1: 静默模式

位 30 **LBKM**: 环回模式 (调试) (Loop back mode (debug))

0: 禁止环回模式
1: 使能环回模式

位 29:26 保留, 必须保持复位值。

位 25:24 **SJW[1:0]**: 再同步跳转宽度 (Resynchronization jump width)

这些位定义 CAN 硬件在执行再同步时最多可以将位加长或缩短的时间片数目。

$$t_{RJW} = t_{CAN} \times (SJW[1:0] + 1)$$

位 23 保留, 必须保持复位值。

位 22:20 **TS2[2:0]**: 时间段 2 (Time segment 2)

这些位定义时间段 2 中的时间片数目。

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

位 19:16 **TS1[3:0]**: 时间段 1 (Time segment 1)

这些位定义时间段 1 中的时间片数目。

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

位 15:10 保留, 必须保持复位值。

位 9:0 **BRP[9:0]**: 波特率预分频器 (Baud rate prescaler)

这些位定义一个时间片的长度。

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

图 32.1.16 寄存器 CAN_BTR 各位描述

STM32F4 提供了两种测试模式, 环回模式和静默模式, 当然他们组合还可以组合成环回静默模式。这里我们简单介绍下环回模式。

在环回模式下, bxCAN 把发送的报文当作接收的报文并保存(如果可以通过接收过滤)在接收邮箱里。也就是环回模式是一个自发自收的模式, 如图 32.1.17 所示:

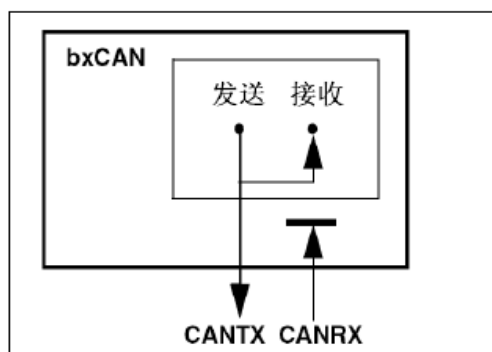


图 32.1.17 CAN 环回模式

环回模式可用于自测试。为了避免外部的影响, 在环回模式下 CAN 内核忽略确认错误(在数据/远程帧的确认位时刻, 不检测是否有显性位)。在环回模式下, bxCAN 在内部把 Tx 输出回馈到 Rx 输入上, 而完全忽略 CANRX 引脚的实际状态。发送的报文可以在 CANTX 引脚上检测到。

第三个, 我们介绍 CAN 发送邮箱标识符寄存器 (CAN_TiRxR) (x=0~3), 该寄存器各位描

述如图 32.1.18 所示:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
STID[10:0]/EXID[28:18]											EXID[17:13]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXID[12:0]													IDE	RTR	TXRQ
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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)
标准标识符或扩展标识符的 MSB (取决于 IDE 位的值)。

位 20:3 **EXID[17:0]**: 扩展标识符 (Extended identifier)
扩展标识符的 LSB。

位 2 **IDE**: 标识符扩展 (Identifier extension)
此位用于定义邮箱中消息的标识符类型。
0: 标准标识符。
1: 扩展标识符。

位 1 **RTR**: 远程发送请求 (Remote transmission request)
0: 数据帧
1: 遥控帧

位 0 **TXRQ**: 发送邮箱请求 (Transmit mailbox request)
由软件置 1, 用于请求发送相应邮箱的内容。
邮箱变为空后, 此位由硬件清零。

图 32.1.18 寄存器 CAN_TlRxR 各位描述

该寄存器主要用来设置标识符 (包括扩展标识符), 另外还可以设置帧类型, 通过 TXRQ 值 1, 来请求邮箱发送。因为有 3 个发送邮箱, 所以寄存器 CAN_TlRxR 有 3 个。

第四个, 我们介绍 CAN 发送邮箱数据长度和时间戳寄存器 (CAN_TDTxR) (x=0~2), 该寄存器我们本章仅用来设置数据长度, 即最低 4 个位。比较简单, 这里就不详细介绍了。

第五个, 我介绍的是 CAN 发送邮箱低字节数据寄存器 (CAN_TDLxR) (x=0~2), 该寄存器各位描述如图 32.1.19 所示:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DATA3[7:0]								DATA2[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA1[7:0]								DATA0[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:24 **DATA3[7:0]**: 数据字节 3 (Data byte 3)
消息的数据字节 3。

位 23:16 **DATA2[7:0]**: 数据字节 2 (Data byte 2)
消息的数据字节 2。

位 15:8 **DATA1[7:0]**: 数据字节 1 (Data byte 1)
消息的数据字节 1。

位 7:0 **DATA0[7:0]**: 数据字节 0 (Data byte 0)
消息的数据字节 0。
一条消息可以包含 0 到 8 个数据字节, 从字节 0 开始。

图 32.1.19 寄存器 CAN_TDLxR 各位描述

该寄存器用来存储将要发送的数据, 这里只能存储低 4 个字节, 另外还有一个寄存器 CAN_TDHxR, 该寄存器用来存储高 4 个字节, 这样总共就可以存储 8 个字节。CAN_TDHxR 的各位描述同 CAN_TDLxR 类似, 我们就不单独介绍了。

第六个,我们介绍 CAN 接收 FIFO 邮箱标识符寄存器 (CAN_RIxR) (x=0/1),该寄存器各位描述同 CAN_TIxR 寄存器几乎一模一样,只是最低位为保留位,该寄存器用于保存接收到的报文标识符等信息,我们可以通过读该寄存器获取相关信息。

同样的,CAN 接收 FIFO 邮箱数据长度和时间戳寄存器 (CAN_RDTxR)、CAN 接收 FIFO 邮箱低字节数据寄存器 (CAN_RDLxR)和 CAN 接收 FIFO 邮箱高字节数据寄存器 (CAN_RDHxR) 分别和发送邮箱的: CAN_TDTxR、CAN_TDLxR 以及 CAN_TDHxR 类似,这里我们就不单独一一介绍了。详细介绍,请参考《STM32F4xx 中文参考手册》24.9.3 节(635 页)。

第七个,我们介绍 CAN 过滤器模式寄存器 (CAN_FM1R),该寄存器各位描述如图 32.1.20 所示:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				FBM27	FBM26	FBM25	FBM24	FBM23	FBM22	FBM21	FBM20	FBM19	FBM18	FBM17	FBM16
				r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FBM15	FBM14	FBM13	FBM12	FBM11	FBM10	FBM9	FBM8	FBM7	FBM6	FBM5	FBM4	FBM3	FBM2	FBM1	FBM0
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

位 31:28 保留,必须保持复位值。

位 27:0 **FBMx**: 筛选器模式 (Filter mode)

筛选器 x 的寄存器的模式

0: 筛选器存储区 x 的两个 32 位寄存器处于标识符屏蔽模式。

1: 筛选器存储区 x 的两个 32 位寄存器处于标识符列表模式。

图 32.1.20 寄存器 CAN_FM1R 各位描述

该寄存器用于设置各滤波器组的工作模式,对 28 个滤波器组的工作模式,都可以通过该寄存器设置,不过该寄存器必须在过滤器处于初始化模式下 (CAN_FMR 的 FINIT 位=1),才可以进行设置。

第八个,我们介绍 CAN 过滤器位宽寄存器(CAN_FS1R),该寄存器各位描述如图 32.1.21 所示:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				FSC27	FSC26	FSC25	FSC24	FSC23	FSC22	FSC21	FSC20	FSC19	FSC18	FSC17	FSC16
				r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FSC15	FSC14	FSC13	FSC12	FSC11	FSC10	FSC9	FSC8	FSC7	FSC6	FSC5	FSC4	FSC3	FSC2	FSC1	FSC0
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

位 31:28 保留,必须保持复位值。

位 27:0 **FSCx**: 筛选器尺度配置 (Filter scale configuration)

这些位定义了筛选器 13-0 的尺度配置。

0: 双 16 位尺度配置

1: 单 32 位尺度配置

图 32.1.21 寄存器 CAN_FS1R 各位描述

该寄存器用于设置各滤波器组的位宽,对 28 个滤波器组的位宽设置,都可以通过该寄存器实现。该寄存器也只能在过滤器处于初始化模式下进行设置。

第九个,我们介绍 CAN 过滤器 FIFO 关联寄存器 (CAN_FFA1R),该寄存器各位描述如图 32.1.22 所示:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				FFA27	FFA26	FFA25	FFA24	FFA23	FFA22	FFA21	FFA20	FFA19	FFA18	FFA17	FFA16
				r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FFA15	FFA14	FFA13	FFA12	FFA11	FFA10	FFA9	FFA8	FFA7	FFA6	FFA5	FFA4	FFA3	FFA2	FFA1	FFA0
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

位 31:28 保留，必须保持复位值。

位 27:0 **FFAx**: 筛选器 x 的筛选器 FIFO 分配 (Filter FIFO assignment for filter x)

通过此筛选器的消息将存储在指定的 FIFO 中。

0: 筛选器分配到 FIFO 0

1: 筛选器分配到 FIFO 1

图 32.1.22 寄存器 CAN_FFA1R 各位描述

该寄存器设置报文通过滤波器组之后，被存入的 FIFO，如果对应位为 0，则存放到 FIFO0；如果为 1，则存放到 FIFO1。该寄存器也只能在过滤器处于初始化模式下配置。

第十个，我们介绍 CAN 过滤器激活寄存器 (CAN_FA1R)，该寄存器各位对应滤波器组和前面的几个寄存器类似，这里就不列出了，对对应位置 1，即开启对应的滤波器组；置 0 则关闭该滤波器组。

最后，我们介绍 CAN 的过滤器组 i 的寄存器 x (CAN_FiRx) (i=0~27; x=1/2)。该寄存器各位描述如图 32.1.23 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FB31	FB30	FB29	FB28	FB27	FB26	FB25	FB24	FB23	FB22	FB21	FB20	FB19	FB18	FB17	FB16
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FB15	FB14	FB13	FB12	FB11	FB10	FB9	FB8	FB7	FB6	FB5	FB4	FB3	FB2	FB1	FB0
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

位 31:0 **FB[31:0]**: 筛选器位 (Filter bits)

标识符

寄存器的每一位用于指定预期标识符相应位的级别。

0: 需要显性位

1: 需要隐性位

掩码

寄存器的每一位用于指定相关标识符寄存器的位是否必须与预期标识符的相应位匹配。

0: 无关，不使用此位进行比较。

1: 必须匹配，传入标识符的此位必须与筛选器相应标识符寄存器中指定的级别相同。

图 32.1.23 寄存器 CAN_FiRx 各位描述

每个滤波器组的 CAN_FiRx 都由 2 个 32 位寄存器构成，即：CAN_FiR1 和 CAN_FiR2。根据过滤器位宽和模式的不同设置，这两个寄存器的功能也不尽相同。关于过滤器的映射，功能描述和屏蔽寄存器的关联，请参见图 32.1.11。

关于 CAN 的介绍，就到此结束了。接下来，我们看看本章我们将实现的功能，及 CAN 的配置步骤。

本章，我们通过 KEY_UP 按键选择 CAN 的工作模式(正常模式/环回模式)，然后通过 KEY0 控制数据发送，并通过查询的办法，将接收到的数据显示在 LCD 模块上。如果是环回模式，我们用一个开发板即可测试。如果是正常模式，我们就需要 2 个探索者 STM32F4 开发板，并且将他们的 CAN 接口对接起来，然后一个开发板发送数据，另外一个开发板将接收到的数据显示在 LCD 模块上。

最后，我们来看看本章的 CAN 的初始化配置步骤：

1) 配置相关引脚的复用功能 (AF9)，使能 CAN 时钟。

我们要用 CAN，第一步就要使能 CAN 的时钟，CAN 的时钟通过 APB1ENR 的第 25 位来设置。其次要设置 CAN 的相关引脚为复用输出，这里我们需要设置 PA11 (CAN1_RX) 和 PA12 (CAN1_TX) 为复用功能 (AF9)，并使能 PA 口的时钟。具体配置过程如下：

```
//使能相关时钟
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE); //使能 PORTA 时钟
RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE); //使能 CAN1 时钟

//初始化 GPIO
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11 | GPIO_Pin_12;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //复用功能
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //100MHz
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化 PA11, PA12

//引脚复用映射配置
GPIO_PinAFConfig(GPIOA, GPIO_PinSource11, GPIO_AF_CAN1); //PA11 复用为 CAN1
GPIO_PinAFConfig(GPIOA, GPIO_PinSource12, GPIO_AF_CAN1); //PA12 复用为 CAN1
```

这里需要提醒一下，CAN 发送接受引脚是哪些 IO 口，可以在中文参考手册引脚表里面查找。

2) 设置 CAN 工作模式及波特率等。

这一步通过先设置 CAN_MCR 寄存器的 INRQ 位，让 CAN 进入初始化模式，然后设置 CAN_MCR 的其他相关控制位。再通过 CAN_BTR 设置波特率和工作模式 (正常模式/环回模式) 等信息。最后设置 INRQ 为 0，退出初始化模式。

在库函数中，提供了函数 CAN_Init() 用来初始化 CAN 的工作模式以及波特率，CAN_Init() 函数体中，在初始化之前，会设置 CAN_MCR 寄存器的 INRQ 为 1 让其进入初始化模式，然后初始化 CAN_MCR 寄存器和 CRN_BTR 寄存器之后，会设置 CAN_MCR 寄存器的 INRQ 为 0 让其退出初始化模式。所以我们在调用这个函数的前后不需要再进行初始化模式设置。下面我们来看看 CAN_Init() 函数的定义：

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

第一个参数就是 CAN 标号，这里我们的芯片只有一个 CAN，所以就是 CAN1。

第二个参数是 CAN 初始化结构体指针，结构体类型是 CAN_InitTypeDef，下面我们来看看这个结构体的定义：

```
typedef struct
{
    uint16_t CAN_Prescaler;
    uint8_t CAN_Mode;
    uint8_t CAN_SJW;
    uint8_t CAN_BS1;
    uint8_t CAN_BS2;
    FunctionalState CAN_TTCM;
    FunctionalState CAN_ABOM;
    FunctionalState CAN_AWUM;
    FunctionalState CAN_NART;
    FunctionalState CAN_RFLM;
```

```
FunctionalState CAN_TAFP;
} CAN_InitTypeDef;
```

这个结构体看起来成员变量比较多，实际上参数可以分为两类。前面 5 个参数是用来设置寄存器 CAN_BTR，用来设置模式以及波特率相关的参数，这在前面有讲解过，设置模式的参数是 CAN_Mode，我们实验中用到回环模式 CAN_Mode_LoopBack 和常规模式 CAN_Mode_Normal，大家还可以选择静默模式以及静默回环模式测试。其他设置波特率相关的参数 CAN_Prescaler，CAN_SJW，CAN_BS1 和 CAN_BS2 分别用来设置波特率分频器，重新同步跳跃宽度以及时间段 1 和时间段 2 占用的时间单元数。后面 6 个成员变量用来设置寄存器 CAN_MCR，也就是设置 CAN 通信相关的控制位。大家可以去翻翻中文参考手册中这两个寄存器的描述，非常详细，我们在前面也有讲解到。初始化实例为：

```
CAN_InitStructure.CAN_TTCM=DISABLE;           //非时间触发通信模式
CAN_InitStructure.CAN_ABOM=DISABLE;           //软件自动离线管理
CAN_InitStructure.CAN_AWUM=DISABLE;           //睡眠模式通过软件唤醒
CAN_InitStructure.CAN_NART=ENABLE;            //禁止报文自动传送
CAN_InitStructure.CAN_RFLM=DISABLE;           //报文不锁定,新的覆盖旧的
CAN_InitStructure.CAN_TAFP=DISABLE;           //优先级由报文标识符决定
CAN_InitStructure.CAN_Mode= CAN_Mode_LoopBack; //模式设置 1,回环模式;
CAN_InitStructure.CAN_SJW=CAN_SJW_1tq; //重新同步跳跃宽度为个时间单位
CAN_InitStructure.CAN_BS1=CAN_BS1_8tq; //时间段 1 占用 8 个时间单位
CAN_InitStructure.CAN_BS2=CAN_BS2_7tq; //时间段 2 占用 7 个时间单位
CAN_InitStructure.CAN_Prescaler=5; //分频系数(Fdiv)
CAN_Init(CAN1, &CAN_InitStructure); // 初始化 CAN1
```

3) 设置滤波器。

本章，我们将使用滤波器组 0，并工作在 32 位标识符屏蔽位模式下。先设置 CAN_FMR 的 FINIT 位，让过滤器组工作在初始化模式下，然后设置滤波器组 0 的工作模式以及标识符 ID 和屏蔽位。最后激活滤波器，并退出滤波器初始化模式。

在库函数中，提供了函数 CAN_FilterInit() 用来初始化 CAN 的滤波器相关参数，CAN_Init() 函数体中，在初始化之前，会设置 CAN_FMR 寄存器的 INRQ 为 INIT 让其进入初始化模式，然后初始化 CAN 滤波器相关的寄存器之后，会设置 CAN_FMR 寄存器的 FINIT 为 0 让其退出初始化模式。所以我们在调用这个函数的前后不需要再进行初始化模式设置。下面我们来看看 CAN_FilterInit() 函数的定义：

```
void CAN_FilterInit(CAN_FilterInitTypeDef* CAN_FilterInitStruct);
```

这个函数只有一个入口参数就是 CAN 滤波器初始化结构体指针，结构体类型为 CAN_FilterInitTypeDef，下面我们看看类型定义：

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

```
uint8_t CAN_FilterScale;  
FunctionalState CAN_FilterActivation;  
} CAN_FilterInitTypeDef;
```

结构体一共有 9 个成员变量,第 1 个至第 4 个是用来设置过滤器的 32 位 id 以及 32 位 mask id,分别通过 2 个 16 位来组合的,这个在前面有讲解过它们的意义。

第 5 个成员变量 CAN_FilterFIFOAssignment 用来设置 FIFO 和过滤器的关联关系,我们的实验是关联的过滤器 0 到 FIFO0, 值为 CAN_Filter_FIFO0。

第 6 个成员变量 CAN_FilterNumber 用来设置初始化的过滤器组,取值范围为 0~13。

第 7 个成员变量 FilterMode 用来设置过滤器组的模式,取值为标识符列表模式

CAN_FilterMode_IdList 和标识符屏蔽位模式 CAN_FilterMode_IdMask。

第 8 个成员变量 FilterScale 用来设置过滤器的位宽为 2 个 16 位 CAN_FilterScale_16bit 还是 1 个 32 位 CAN_FilterScale_32bit。

第 9 个成员变量 CAN_FilterActivation 就很明了了,用来激活该过滤器。

过滤器初始化参考实例代码:

```
CAN_FilterInitStructure.CAN_FilterNumber=0;    //过滤器 0  
CAN_FilterInitStructure.CAN_FilterMode=CAN_FilterMode_IdMask;  
CAN_FilterInitStructure.CAN_FilterScale=CAN_FilterScale_32bit; //32 位  
CAN_FilterInitStructure.CAN_FilterIdHigh=0x0000; ///32 位 ID  
CAN_FilterInitStructure.CAN_FilterIdLow=0x0000;  
CAN_FilterInitStructure.CAN_FilterMaskIdHigh=0x0000; //32 位 MASK  
CAN_FilterInitStructure.CAN_FilterMaskIdLow=0x0000;  
CAN_FilterInitStructure.CAN_FilterFIFOAssignment=CAN_Filter_FIFO0; // FIFO0  
CAN_FilterInitStructure.CAN_FilterActivation=ENABLE; //激活过滤器 0  
CAN_FilterInit(&CAN_FilterInitStructure); //滤波器初始化
```

4) 发送接受消息

在初始化 CAN 相关参数以及过滤器之后,接下来就是发送和接收消息了。库函数中提供了发送和接收消息的函数。发送消息的函数是:

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

这个函数比较好理解,第一个参数是 CAN 标号,我们使用 CAN1。第二个参数是相关消息结构体 CanTxMsg 指针类型,CanTxMsg 结构体的成员变量用来设置标准标识符,扩展标识符,消息类型和消息帧长度等信息。

接受消息的函数是:

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

前面两个参数也比较好理解,CAN 标号和 FIFO 号。第二个参数 RxMessage 是用来存放接受到的消息信息。结构体 CanRxMsg 和结构体 CanTxMsg 比较接近,分别用来定义发送消息和描述接受消息,大家可以对照看一下,也比较好理解。

5) CAN 状态获取

对于 CAN 发送消息的状态,挂起消息数目等等之类的传输状态信息,库函数提供了一些列的函数,包括 CAN_TransmitStatus()函数, CAN_MessagePending()函数, CAN_GetFlagStatus()函数等等,大家可以根据需要来调用。

至此,CAN 就可以开始正常工作了。如果用到中断,就还需要进行中断相关的配置,本章因为没用到中断,所以就不作介绍了。

32.2 硬件设计

本章要用到的硬件资源如下：

- 1) 指示灯 DS0
- 2) KEY0 和 KEY_UP 按键
- 3) TFTLCD 模块
- 4) CAN
- 5) CAN 收发芯片 JTA1050

前面 3 个之前都已经详细介绍过了，这里我们介绍 STM32F4 与 TJA1050 连接关系，如图 32.2.1 所示：

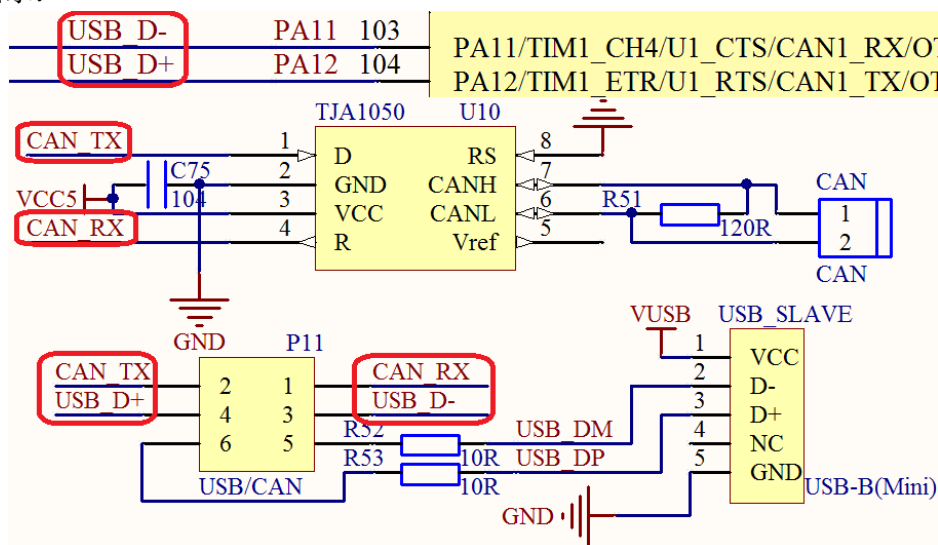


图 32.2.1 STM32F4 与 TJA1050 连接电路图

从上图可以看出：STM32F4 的 CAN 通过 P11 的设置，连接到 TJA1050 收发芯片，然后通过接线端子（CAN）同外部的 CAN 总线连接。图中可以看出，在探索者 STM32F4 开发板上面是带有 120Ω 的终端电阻的，如果我们的开发板不是作为 CAN 的终端的话，需要把这个电阻去掉，以免影响通信。另外，需要注意：CAN1 和 USB 共用了 PA11 和 PA12，所以他们不能同时使用。

这里还要注意，我们要设置好开发板上 P11 排针的连接，通过跳线帽将 PA11 和 PA12 分别连接到 CRX（CAN_RX）和 CTX（CAN_TX）上面，如图 32.2.2 所示：

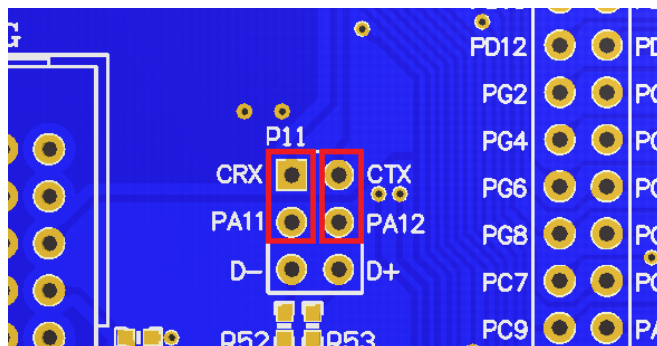


图 32.2.2 硬件连接示意图

最后，我们用 2 根导线将两个开发板 CAN 端子的 CAN_L 和 CAN_L，CAN_H 和 CAN_H 连接起来。这里注意不要接反了（CAN_L 接 CAN_H），接反了会导致通讯异常！！

32.3 软件设计

打开 CAN 通信实验的工程可以看到，我们增加了文件 can.c 以及头文件 can.h，同时 CAN 相关的固件库函数和定义分布在文件 stm32f4xx_can.c 和头文件 stm32f4xx_can.h 中。

打开 can.c 文件，代码如下：

```
//CAN 初始化
//tsjw:重新同步跳跃时间单元. @ref CAN_synchronisation_jump_width
//tbs2:时间段 2 的时间单元. @ref CAN_time_quantum_in_bit_segment_2
//tbs1:时间段 1 的时间单元. @ref CAN_time_quantum_in_bit_segment_1
//brp :波特率分频器.范围:1~1024;(实际要加 1,也就是 1~1024) tq=(brp)*tpclk1
//mode: @ref CAN_operating_mode
u8 CAN1_Mode_Init(u8 tsjw,u8 tbs2,u8 tbs1,u16 brp,u8 mode)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    CAN_InitTypeDef      CAN_InitStructure;
    CAN_FilterInitTypeDef CAN_FilterInitStructure;
#ifdef CAN1_RX0_INT_ENABLE
    NVIC_InitTypeDef  NVIC_InitStructure;
#endif
    //使能相关时钟
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE); //使能 PORTA 时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE); //使能 CAN1 时钟
    //初始化 GPIO
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11| GPIO_Pin_12;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //复用功能
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
    GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化 PA11,PA12

    //引脚复用映射配置
    GPIO_PinAFConfig(GPIOA,GPIO_PinSource11,GPIO_AF_CAN1); //PA11 复用为 CAN1
    GPIO_PinAFConfig(GPIOA,GPIO_PinSource12,GPIO_AF_CAN1); //PA12 复用为 CAN1

    //CAN 单元设置
    CAN_InitStructure.CAN_TTCM=DISABLE; //非时间触发通信模式
    CAN_InitStructure.CAN_ABOM=DISABLE; //软件自动离线管理
    CAN_InitStructure.CAN_AWUM=DISABLE; //睡眠模式通过软件唤醒
    CAN_InitStructure.CAN_NART=ENABLE; //禁止报文自动传送
    CAN_InitStructure.CAN_RFLM=DISABLE; //报文不锁定,新的覆盖旧的
    CAN_InitStructure.CAN_TXFP=DISABLE; //优先级由报文标识符决定
    CAN_InitStructure.CAN_Mode= mode; //模式设置
    CAN_InitStructure.CAN_SJW=tsjw; //重新同步跳跃宽度
```



```

CAN_InitStructure.CAN_BS1=tbs1; //Tbs1 范围 CAN_BS1_1tq ~CAN_BS1_16tq
CAN_InitStructure.CAN_BS2=tbs2; //Tbs2 范围 CAN_BS2_1tq ~ CAN_BS2_8tq
CAN_InitStructure.CAN_Prescaler=brp; //分频系数(Fdiv)为 brp+1
CAN_Init(CAN1, &CAN_InitStructure); // 初始化 CAN1

//配置过滤器
CAN_FilterInitStructure.CAN_FilterNumber=0; //过滤器 0
CAN_FilterInitStructure.CAN_FilterMode=CAN_FilterMode_IdMask;
CAN_FilterInitStructure.CAN_FilterScale=CAN_FilterScale_32bit; //32 位
CAN_FilterInitStructure.CAN_FilterIdHigh=0x0000; ///32 位 ID
CAN_FilterInitStructure.CAN_FilterIdLow=0x0000;
CAN_FilterInitStructure.CAN_FilterMaskIdHigh=0x0000; //32 位 MASK
CAN_FilterInitStructure.CAN_FilterMaskIdLow=0x0000;
CAN_FilterInitStructure.CAN_FilterFIFOAssignment=CAN_Filter_FIFO0;
CAN_FilterInitStructure.CAN_FilterActivation=ENABLE; //激活过滤器 0
CAN_FilterInit(&CAN_FilterInitStructure); //滤波器初始化

#if CAN1_RX0_INT_ENABLE
    CAN_ITConfig(CAN1,CAN_IT_FMP0,ENABLE); //FIFO0 消息挂号中断允许.
    NVIC_InitStructure.NVIC_IRQChannel = CAN1_RX0_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1; // 主优先级为 1
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0; // 次优先级为 0
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
#endif
    return 0;
}
#if CAN1_RX0_INT_ENABLE //使能 RX0 中断
//中断服务函数
void CAN1_RX0_IRQHandler(void)
{
    CanRxMsg RxMessage;
    int i=0;
    CAN_Receive(CAN1, 0, &RxMessage);
    for(i=0;i<8;i++)
        printf("rxbuf[%d]:%d\r\n",i,RxMessage.Data[i]);
}
#endif

//can 发送一组数据(固定格式:ID 为 0X12,标准帧,数据帧)
//len:数据长度(最大为 8) msg:数据指针,最大为 8 个字节.
//返回值:0,成功; 其他,失败;
u8 CAN1_Send_Msg(u8* msg,u8 len)

```

```

{
    u8 mbox;
    u16 i=0;
    CanTxMsg TxMessage;
    TxMessage.StdId=0x12;    // 标准标识符为 0
    TxMessage.ExtId=0x12;    // 设置扩展标识符（29 位）
    TxMessage.IDE=0;         // 使用扩展标识符
    TxMessage.RTR=0;         // 消息类型为数据帧，一帧 8 位
    TxMessage.DLC=len;       // 发送两帧信息
    for(i=0;i<len;i++)
        TxMessage.Data[i]=msg[i];    // 第一帧信息
    mbox= CAN_Transmit(CAN1, &TxMessage);
    i=0;
    while((CAN_TransmitStatus(CAN1, mbox)==CAN_TxStatus_Failed)&&(i<0XFFF))i++;
    if(i>=0XFFF)return 1;
    return 0;
}

//can 口接收数据查询
//buf:数据缓存区;
//返回值:0,无数据被收到; 其他,接收的数据长度;
u8 CAN1_Receive_Msg(u8 *buf)
{
    u32 i;
    CanRxMsg RxMessage;
    if( CAN_MessagePending(CAN1,CAN_FIFO0)==0)return 0;//没有接收到数据,直接退出
    CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);//读取数据
    for(i=0;i<RxMessage.DLC;i++)
        buf[i]=RxMessage.Data[i];
    return RxMessage.DLC;//返回接受到的数据长度
}

```

此部分代码总共 3 个函数，首先是：CAN_Mode_Init 函数。该函数用于 CAN 的初始化，该函数带有 5 个参数，可以设置 CAN 通信的波特率和工作模式等，在该函数中，我们就是按 32.1 节末尾的介绍来初始化的，本章中，我们设计滤波器组 0 工作在 32 位标识符屏蔽模式，从设计值可以看出，该滤波器是不会对任何标识符进行过滤的，因为所有的标识符位都被设置成不需要关心，这样设计，主要是方便大家实验。

第二个函数，Can_Send_Msg 函数。该函数用于 CAN 报文的发送，主要是设置标识符 ID 等信息，写入数据长度和数据，并请求发送，实现一次报文的发送。

第三个函数，Can_Receive_Msg 函数。用来接受数据并且将接受到的数据存放到 buf 中。

can.c 里面，还包含了中断接收的配置，通过 can.h 的 CAN1_RX0_INT_ENABLE 宏定义，来配置是否使能中断接收，本章我们不开启中断接收的。其他函数我们就不一一介绍了，都比较简单，大家自行理解即可。

can.h 头文件中，CAN1_RX0_INT_ENABLE 用于设置是否使能中断接收，本章我们不用中断接收，故设置为 0。最后我们看看主函数，代码如下：

```
int main(void)
```

```

{
    u8 key, i=0, t=0, cnt=0, u8 canbuf[8], res;
    u8 mode=1; //CAN 工作模式; 0, 普通模式; 1, 环回模式
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    LCD_Init(); //LCD 初始化
    KEY_Init(); //按键初始化
    CAN1_Mode_Init(CAN_SJW_1tq, CAN_BS2_6tq, CAN_BS1_7tq, 6,
        CAN_Mode_LoopBack); //CAN 初始化环回模式, 波特率 500Kbps
    POINT_COLOR=RED; //设置字体为红色
    LCD_ShowString(30, 50, 200, 16, 16, "Explorer STM32F4");
    LCD_ShowString(30, 70, 200, 16, 16, "CAN TEST");
    LCD_ShowString(30, 90, 200, 16, 16, "ATOM@ALIENTEK");
    LCD_ShowString(30, 110, 200, 16, 16, "2014/5/7");
    LCD_ShowString(30, 130, 200, 16, 16, "LoopBack Mode");
    LCD_ShowString(30, 150, 200, 16, 16, "KEY0:Send WK_UP:Mode"); //显示提示信息
    POINT_COLOR=BLUE; //设置字体为蓝色
    LCD_ShowString(30, 170, 200, 16, 16, "Count:"); //显示当前计数值
    LCD_ShowString(30, 190, 200, 16, 16, "Send Data:"); //提示发送的数据
    LCD_ShowString(30, 250, 200, 16, 16, "Receive Data:"); //提示接收到的数据
    while(1)
    {
        key=KEY_Scan(0);
        if(key==KEY0_PRES) //KEY0 按下, 发送一次数据
        {
            for(i=0; i<8; i++)
            {
                canbuf[i]=cnt+i; //填充发送缓冲区
                if(i<4) LCD_ShowxNum(30+i*32, 210, canbuf[i], 3, 16, 0X80); //显示数据
                else LCD_ShowxNum(30+(i-4)*32, 230, canbuf[i], 3, 16, 0X80); //显示数据
            }
            res=CAN1_Send_Msg(canbuf, 8); //发送 8 个字节
            if(res) LCD_ShowString(30+80, 190, 200, 16, 16, "Failed"); //提示发送失败
            else LCD_ShowString(30+80, 190, 200, 16, 16, "OK "); //提示发送成功
        }
        else if(key==WKUP_PRES) //WK_UP 按下, 改变 CAN 的工作模式
        {
            mode=!mode;
            CAN1_Mode_Init(CAN_SJW_1tq, CAN_BS2_6tq, CAN_BS1_7tq, 6, mode);
            //CAN 普通模式初始化, 普通模式, 波特率 500Kbps
            POINT_COLOR=RED; //设置字体为红色
            if(mode==0) //普通模式, 需要 2 个开发板
            {

```

```

        LCD_ShowString(30,130,200,16,16,"Normal Mode ");
    }else //回环模式,一个开发板就可以测试了.
    {
        LCD_ShowString(30,130,200,16,16,"LoopBack Mode");
    }
    POINT_COLOR=BLUE;//设置字体为蓝色
}
key=CAN1_Receive_Msg(canbuf);
if(key)//接收到有数据
{
    LCD_Fill(30,270,160,310,WHITE);//清除之前的显示
    for(i=0;i<key;i++)
    {
        if(i<4)LCD_ShowxNum(30+i*32,270,canbuf[i],3,16,0X80);    //显示数据
        else LCD_ShowxNum(30+(i-4)*32,290,canbuf[i],3,16,0X80);    //显示数据
    }
}
t++; delay_ms(10);
if(t==20)
{
    LED0=!LED0;//提示系统正在运行
    t=0;cnt++;
    LCD_ShowxNum(30+48,170,cnt,3,16,0X80);    //显示数据
}
}
}

```

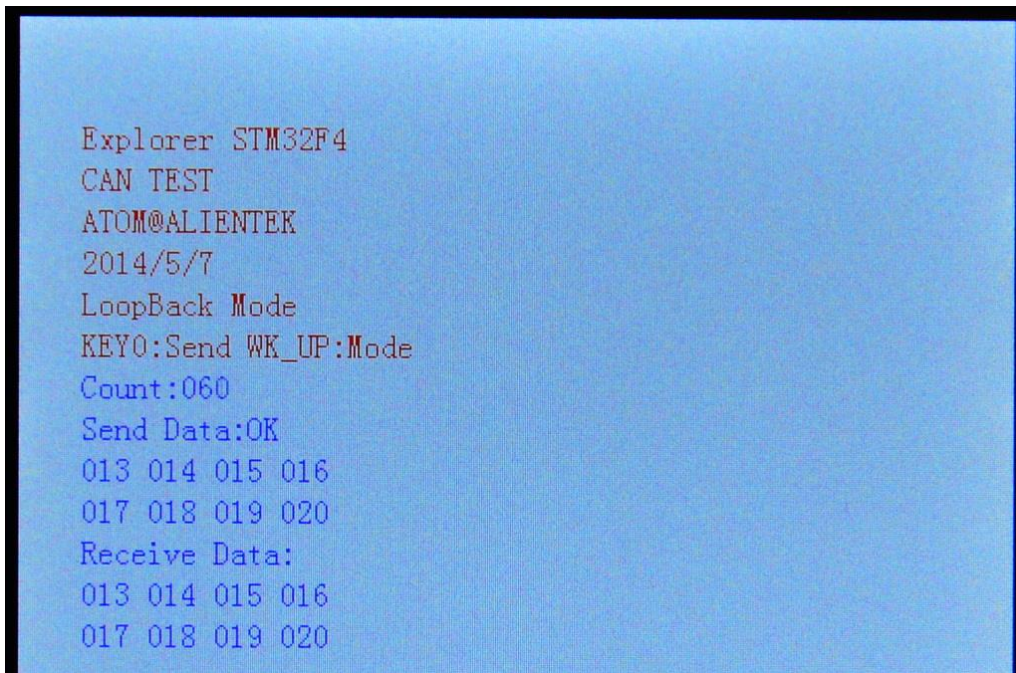
此部分代码，我们主要关注下 CAN1_Mode_Init 初始化代码：

```
CAN1_Mode_Init(CAN_SJW_1tq,CAN_BS2_6tq,CAN_BS1_7tq,6,mode);
```

该函数用于设置波特率和 CAN 的模式，根据前面的波特率计算公式，我们知道这里的波特率被初始化为 500Kbps。mode 参数用于设置 CAN 的工作模式（普通模式/环回模式），通过 KEY_UP 按键，可以随时切换模式。cnt 是一个累加数，一旦 KEY0 按下，就以这个数位基准连续发送 8 个数据。当 CAN 总线收到数据的时候，就将收到的数据直接显示在 LCD 屏幕上。

32.4 下载验证

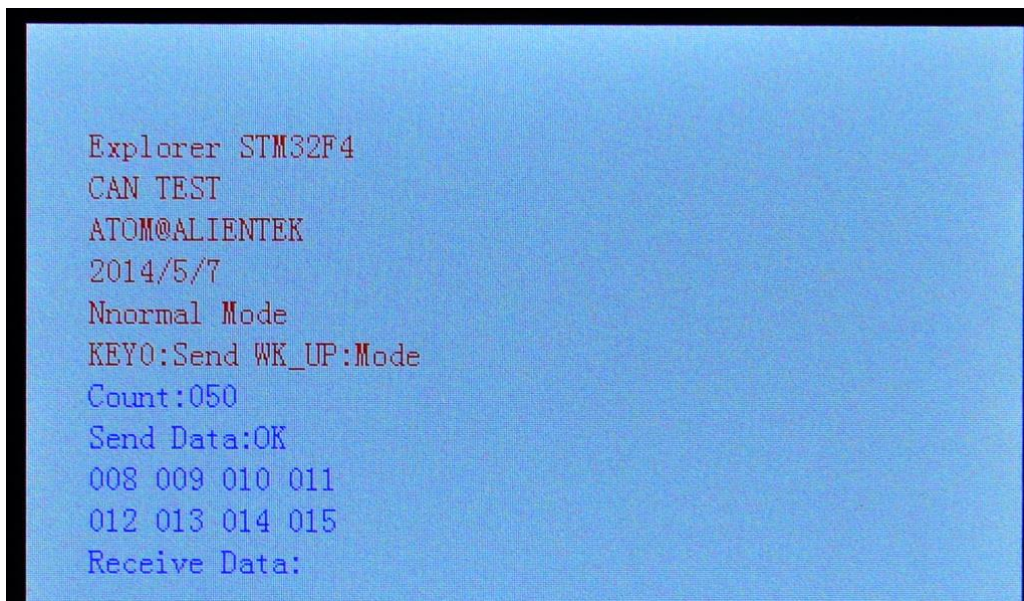
在代码编译成功之后，我们通过下载代码到 ALIENTEK 探索者 STM32F4 开发板上，得到如图 32.4.1 所示：



```
Explorer STM32F4
CAN TEST
ATOM@ALIENTEK
2014/5/7
LoopBack Mode
KEY0:Send WK_UP:Mode
Count:060
Send Data:OK
013 014 015 016
017 018 019 020
Receive Data:
013 014 015 016
017 018 019 020
```

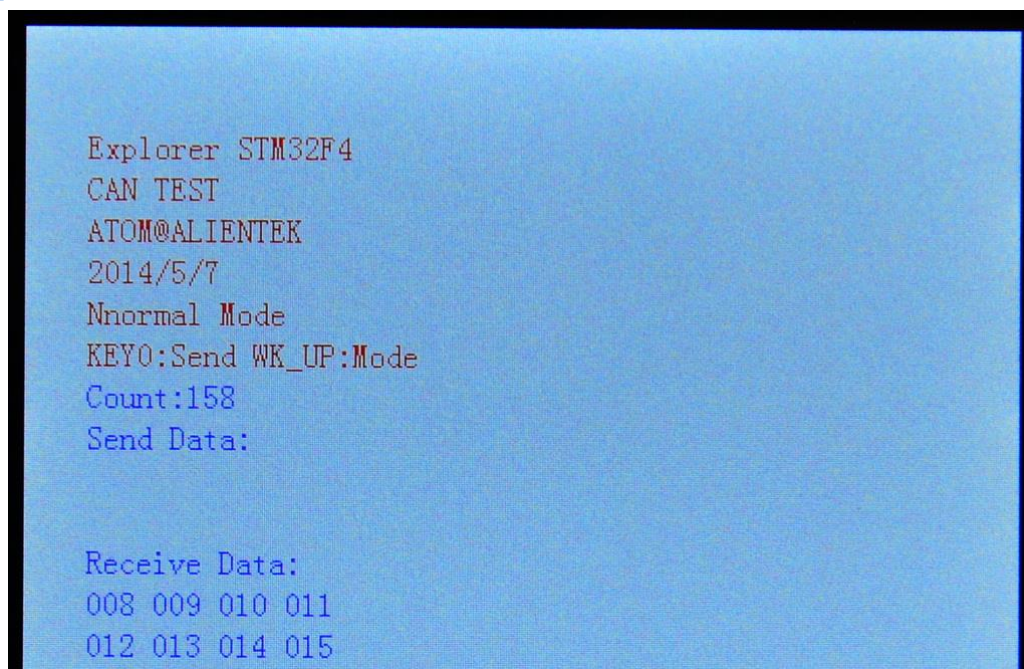
图 32.4.1 程序运行效果图

伴随 DS0 的不停闪烁，提示程序在运行。默认我们是设置的环回模式，此时，我们按下 KEY0 就可以在 LCD 模块上面看到自发自收的数据（如上图所示），如果我们选择普通模式（通过 KEY_UP 按键切换），就必须连接两个开发板的 CAN 接口，然后就可以互发数据了。如图 32.4.2 和图 32.4.3 所示：



```
Explorer STM32F4
CAN TEST
ATOM@ALIENTEK
2014/5/7
Normal Mode
KEY0:Send WK_UP:Mode
Count:050
Send Data:OK
008 009 010 011
012 013 014 015
Receive Data:
```

图 32.4.2 CAN 普通模式发送数据



```
Explorer STM32F4
CAN TEST
ATOM@ALIENTEK
2014/5/7
Nnormal Mode
KEY0:Send WK_UP:Mode
Count:158
Send Data:

Receive Data:
008 009 010 011
012 013 014 015
```

图 32.4.3 CAN 普通模式接收数据

图 32.4.2 来自开发板 A，发送了 8 个数据，图 32.4.3 来自开发板 B，收到了来自开发板 A 的 8 个数据。