AN0095
入门指南

AT32 MCU CAN入门指南

前言

CAN（Controller Area Network）是一种实现各节点之间实时、可靠的数据通信的分布式串行通信协议，支持 CAN 协议 2.0A 和 2.0B。本文介绍了 CAN 标准协议，AT32 CAN 的使用流程以及基于 AT32 的几个 CAN 使用例程。

注：本应用笔记对应的代码是基于雅特力提供的V2.x.x 板级支持包（BSP）而开发，对于其他版本 BSP，需要注意使用上的区别。

支持型号列表：

支持型号	AT32F403A 系列 AT32F407 系列 AT32F403 系列 AT32F413 系列 AT32F415 系列 AT32F421 系列 AT32F425 系列 AT32F435 系列 AT32F437 系列 AT32WB415 系列 AT32F402 系列 AT32F405 系列 AT32F423 系列 AT32L021 系列

目录

表目录

图目录

1 CAN 概述

CAN 是Controller Area Network 的缩写（以下称为CAN），它的设计目标是以最小的CPU负荷来高效处理大量的报文。1986 年德国电气商BOSCH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。

CAN协议特点：

- **多主控制：**

  在总线空闲时，所有节点均可发送信息。如果出现两个及以上节点同时开始发送信息时，总线会根据标识符（Identifier 以下称为 ID）进行仲裁，ID 越小则优先级高，则仲裁优胜，仲裁优胜的节点继续发送，仲裁失利的节点立即转入接收状态。需注意，ID 并不是表示节点地址，而是指示所发送的报文的优先级。

- **系统的灵活性：**

  如上所述，与 CAN 总线的各节点没有类似于“地址”的信息。因此在总线上增减节点时，连接在总线上的其它节点的软硬件设计均不受影响。

- **高可靠性：**

  CAN 协议具有错误检测、错误通知、故障封闭和错误恢复功能。CAN 总线上的任意节点均可检测错误（错误检测）；检测到错误后向总线发送错误帧以通知其他节点（错误通知）；同时每个节点内部有一个错误计数功能，每次检测到错误之后，错误计数值累加，当某节点持续错误导致计数连续累加，直到大于 256 后，此故障节点从总线上断开，避免影响其他节点（故障封闭）。且发送节点如果在发送信息的过程中检测到错误，待错误结束后会自动重发此信息直到成功发送（错误恢复）。

- **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）。

- **可连接节点多：**

  CAN 总线是可同时连接多个节点。节点数量理论上是没有限制的。但实际上节点数量受总线时间延迟及电气负载的限制。降低通信速度，可连接的节点数增加；提高通信速度，则可连接的节点数减少。

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。本文例程使用的是1Mbps的通信速率，使用的是ISO11898标准。

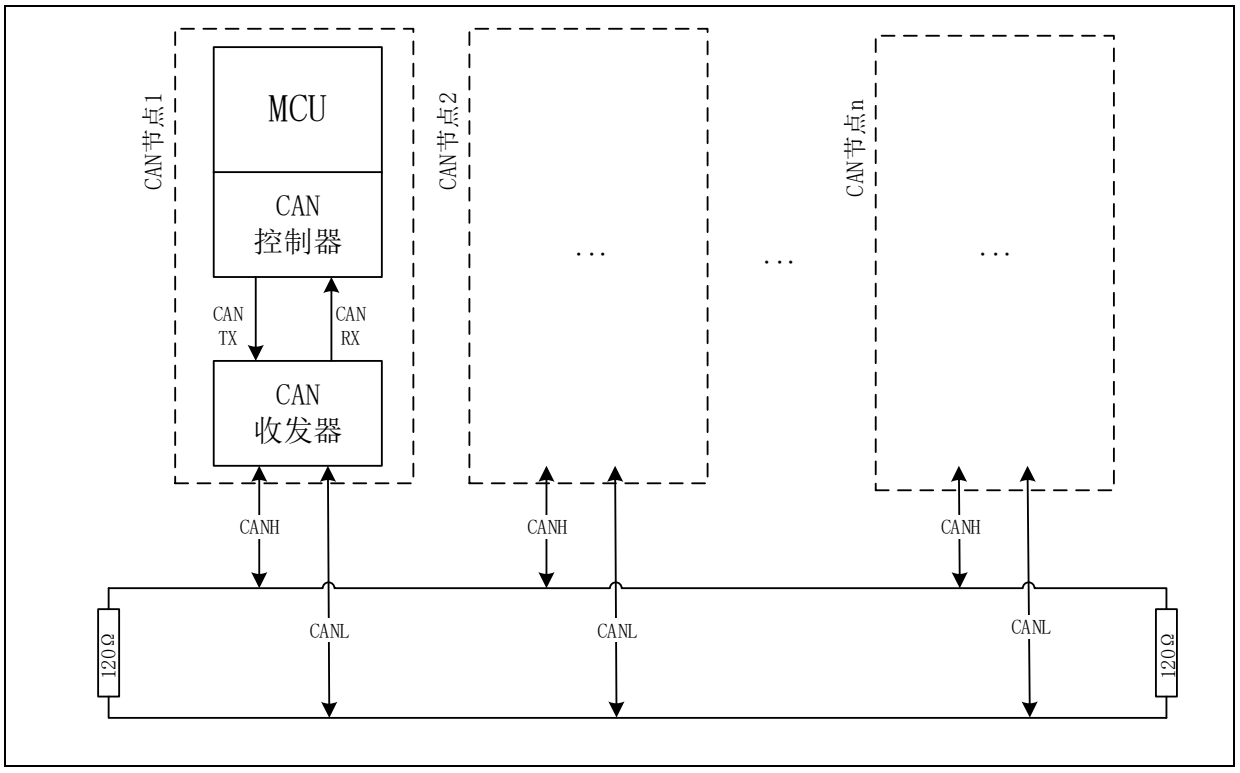
2 CAN 协议介绍

本章主要介绍 CAN 的网络拓扑结构、总线物理层特性、帧类型、帧结构、位填充机制、位格式、同步机制、仲裁机制、错误处理机制等。另外还有 CAN 协议的更多细节请参考 BOSCH CAN 协议，本文不再详述。

2.1 CAN 网络拓扑结构

如下图 1：CAN 总线由两条差分线 CANH 和 CANL 组成，各个节点通过较短的支线接入 CAN 总线。各节点从通信协议而言是没有主从和地址区分的，每个节点均可以平等的收发数据。另外，在 CAN 总线的两端各有一个 120Ω 的终端电阻，来做阻抗匹配，以减少回波反射。

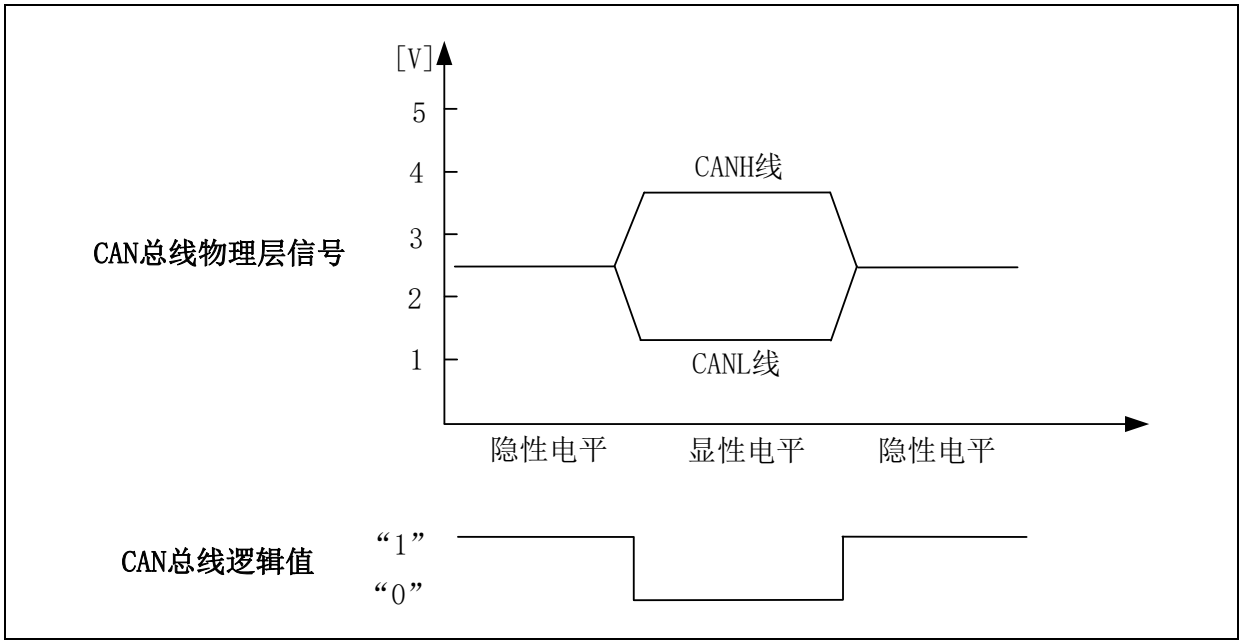
图 1 CAN 网络拓扑结构



2.2 CAN 总线物理层特性

如下图 2：显性电平对应逻辑“0”，CANH 和 CANL 压差 2.5V 左右。而隐性电平对应逻辑“1”，CANH 和 CANL 压差为 0V。在总线上，显性电平具有优先权，只要有一个节点输出显性电平，总线上即为显性电平。而隐形电平则具有包容的意味，只有所有的节点都输出隐性电平，总线上才为隐性电平。

图 2 CAN 总线电平特性



2.3 帧类型

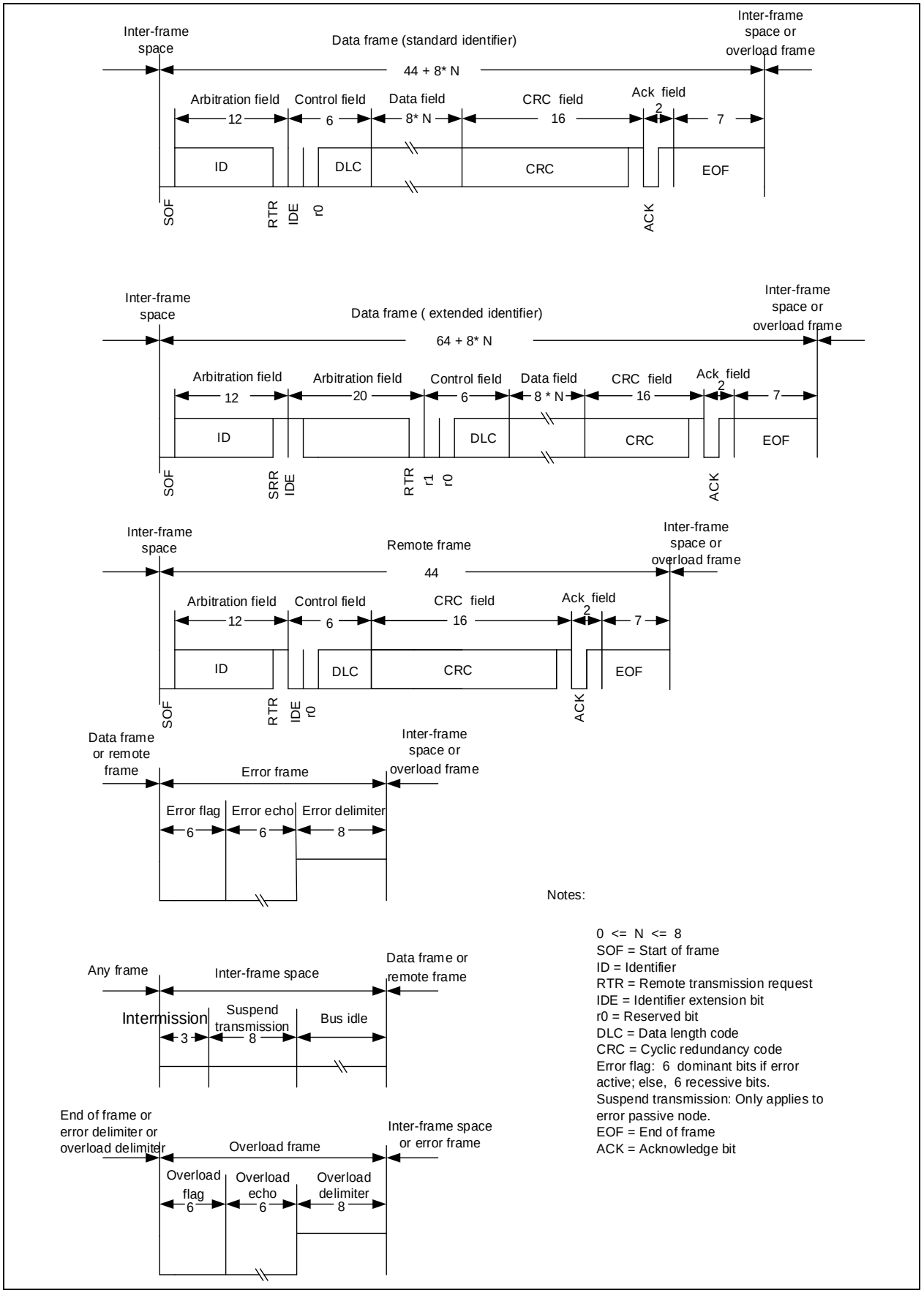
如下表 1，CAN 包含了以下 5 种帧类型。其中数据帧和远程帧由用户控制收发；错误帧、过载帧和间隔帧是 CAN 总线上各节点硬件根据对应状态发送，用户不能也无需控制。

表 1 CAN 帧类型

帧类型	描述
数据帧	数据帧携带数据从发送节点至接收节点
远程帧	某节点发出远程帧，请求其他节点发送具有同一识别符的数据帧
错误帧	任何节点检测到一总线错误就发出错误帧
过载帧	过载帧用以在先行的和后续的数据帧（或远程帧）之间提供一附加的延时
间隔帧	又称帧间间隔，用于将上述各种类型的帧隔开

如下图 3，包含了各类型帧结构示意。

图 3 CAN 各帧类型



Notes:

0 <= N <= 8
SOF = Start of frame
ID = Identifier
RTR = Remote transmission request
IDE = Identifier extension bit
r0 = Reserved bit
DLC = Data length code
CRC = Cyclic redundancy code
Error flag: 6 dominant bits if error active; else, 6 recessive bits.
Suspend transmission: Only applies to error passive node.
EOF = End of frame
ACK = Acknowledge bit

2.4 帧结构

本文仅对标准数据帧进行详细介绍，其他帧类型可参考图 3 与标准数据帧进行对比理解。

一帧标准数据帧包含如下部分：

帧起始：为 1bit 显性位。由于 CAN 总线空闲时是隐性电平，帧起始的显性位用于提示总线上的节点“一帧信息传输开始了”。

仲裁段：表示该帧优先级的段，包含标识符和帧类型（数据/远程帧）。

控制段：表示数据的字节数、标识符类型（标准/扩展标识符）及保留位的段。

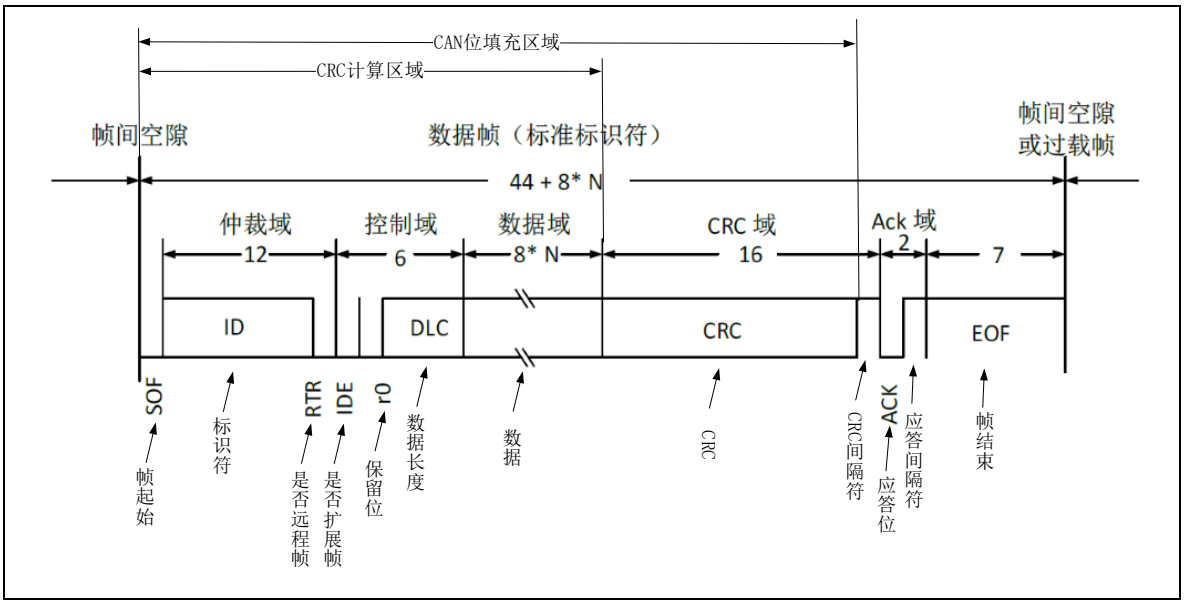
数据段：数据，一帧可发送 0~8 个字节的数据（数据长度根据控制段的 DLC 决定）。

CRC 段：发送节点将 CRC 计算区域（不包含填充位）进行 CRC 计算后放入 CRC 段发送。接收节点也对 CRC 计算区域进行 CRC 计算，并与收到的 CRC 域进行对比，若 CRC 对比结果有误则向总线发送错误帧，若对比结果正确则随后发送应答。

ACK 段：含应答位（ACK SLOT）和应答间隔符（ACK DELIMITER）。发送节点在 ACK 段均发送隐性电平；接收节点如果在接收过程中没有检测到错误，则在应答位输出 1bit 显性电平，以通知发送节点“这帧数据被正确的接收了”。

帧结束：表示数据帧结束的段，为 7bit 隐性电平。

图 4 CAN 标准数据帧



2.5 位填充

由于 CAN 总线只有 CANH/CANL 两条差分线，没有 CLK 线来做同步，所以 CAN 是直接通过数据流中间的跳变沿来做同步的（参考下文同步机制）。而为了避免数据流中出现大段没有跳变沿的情况，CAN 加入了“位填充”机制。

即发送器只要检测到位流里有 5 个连续相同值的位，便自动在位流里插入一相反电平的填充位。例如，原始数据流为“0000000111110001…”，经过位填充后实际输出到 CAN 总线的数据流为“000001001111100001…”，加下划线的位即为填充位。

位填充的范围为帧起始（SOF）~CRC 域（不含 CRC 间隔符），参考上图 4。

2.6 位格式

AT32 的 CAN 一个 bit 可分为 3 段：

- 同步段（SYNC_SEG）
- 位段 1（BIT SEGMENT 1），包括 CAN 标准里的 PROP_SEG 和 PHASE_SEG1，记为 BSEG1。
- 位段 2（BIT SEGMENT 2），即 CAN 标准里的 PHASE_SEG2，记为 BSEG2。

这些段又由 Time Quantum（以下称为 Tq）的最小时间单位构成。

1 位分为 3 个段，每个段又由若干个 Tq 构成，这称为位时序。

1 位由多少个 Tq 构成、每个段又由多少个 Tq 构成等，可以任意设定位时序。用户通过设定位时序和 Tq 长度来设定 CAN 的波特率和采样点。关于波特率和采样点设置，后文详细介绍。

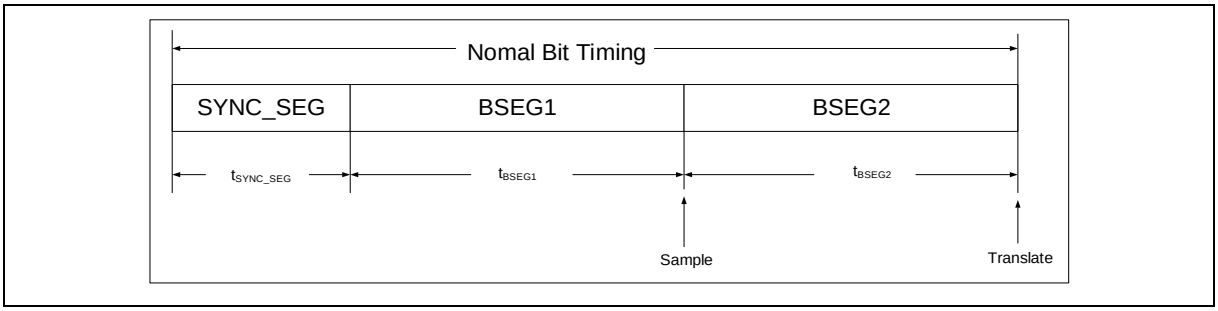
各段作用及 AT32 的 CAN 可配置的 Tq 数见下表 2：

表 2 位的各段的作用

名称（CAN 标准）	名称（AT32 格式）	作用	Tq 数
同步段（SYNC_SEG）	同步段（SYNC_SEG）	同步段用于同步总线上不同的节点。理想情况下，跳变沿出现在此段。	1Tq
传播段（PROP SEG）	位段 1（BSEG1）	传播段用于补偿网络内的物理延时时间。它是总线上输入比较器延时和输出驱动器延时总和的两倍。	1~16Tq
相位缓冲段 1（PHASE SEG1）		相位缓冲段用于补偿边沿阶段的错误。这两个段可以通过重同步跳跃加长或缩短。	
相位缓冲段 2（PHASE SEG2）	位段 2（BSEG2）		1~8Tq

如下图 5，同步段、位段 1 和位段 2 组成一个 bit。BSEG1 和 BSEG2 段交界处为采样点，即接收节点采样的时间点。

图 5 位时序



2.7 同步机制

硬同步（HARD SYHCHRONIZATION）：

硬同步后，内部的位时间从同步段重新开始。因此，硬同步强迫由硬同步引起的沿处于重新开始的位时间同步段之内。即下图 6 的理想跳变沿情况。

重新同步跳转宽度（RESYHCHRONIZATION JUMP WIDTH）：

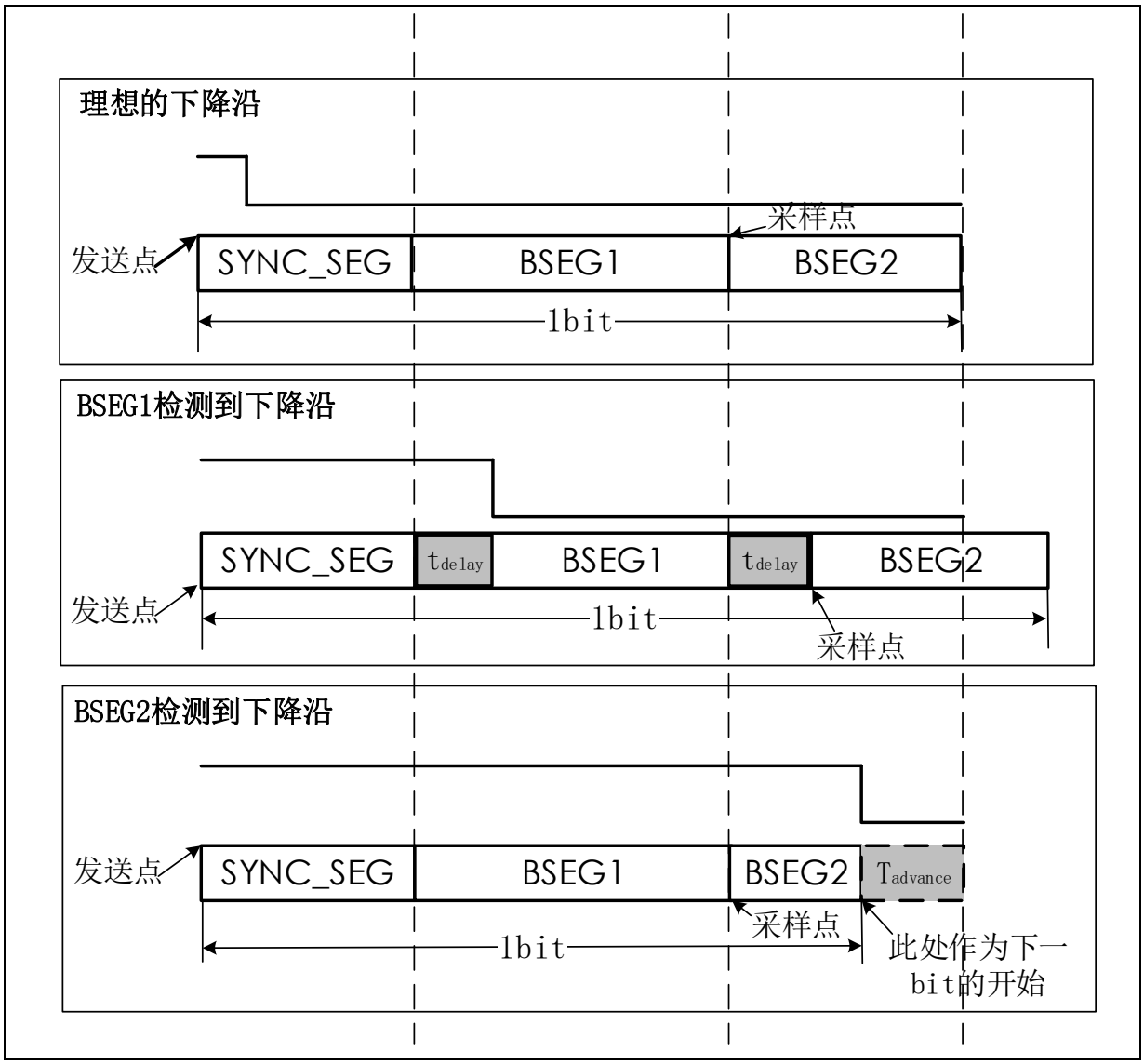
重新同步的结果，使位段 1 增长，或使位段 2 缩短。位段增长或缩短的数量有一个上限，此上限由重新同步跳转宽度给定。重新同步跳转宽度应设置于 1~4Tq 之间。

如下图 6：

当在 BSEG1 段检测到下降沿，则 BSEG1 段增长 Tdelay，当前 bit 整体增长 Tdelay，其中 Tdelay<=重新同步跳转宽度。

当在 BSEG2 段检测到下降沿，则 BSEG2 段缩短 Tadvance，当前 bit 整体缩短 Tadvance，其中 Tadvance<=重新同步跳转宽度。

图 6 重同步跳跃

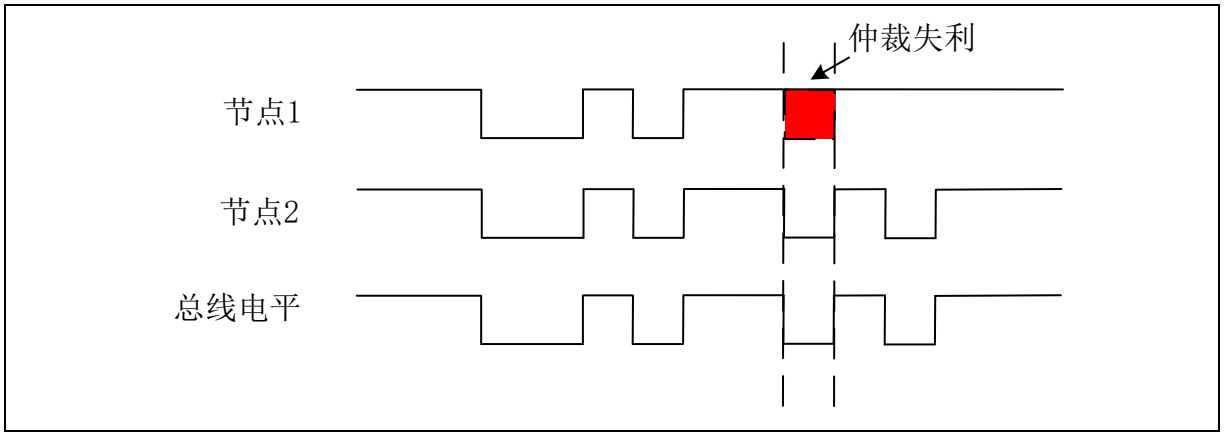


2.8 仲裁机制

只要总线空闲，任何单元都可以开始发送报文。如果 2 个或 2 个以上的单元同时开始传送报文，那么就会有总线访问冲突。通过对 ID 进行逐位仲裁可以解决这个冲突。仲裁的机制确保了报文和时间均不损失。当具有相同 ID 的数据帧和远程帧同时初始化时，数据帧优先于远程帧。仲裁期间，每一个发送节点都对发送位的电平与被监控的总线电平进行比较。如果电平相同，则这个节点可以继续发送。如果发送的是一“隐性”电平而监测到是一“显性”电平（见总线电平），那么该节点就失去了仲裁，必须立即退出发送状态并转入接收状态。

例如下图 7，节点 1 和节点 2 同时发送一帧数据，ID 段前几 bit 相同。直到红色处，节点 1 发送隐性电平“1”，节点 2 发送显性电平“0”。此时节点 2 仲裁优胜，继续发送，总线电平和节点 2 发送值一致；而节点 1 仲裁失利，在下一 bit 转入接收，后续节点 1 的发送引脚保持隐性电平。

图 7 仲裁机制



2.9 错误处理机制

2.9.1 错误类型

CAN 协议定义了以下 5 种不同的错误类型：

- **位错误（Bit Error）**

  单元在发送位的同时也对总线进行监视。如果所发送的位值与所监视的位值不相符合，则在此位时间里检测到一个位错误。AT32 将位错误细分为显性位错误（发送显性位但检测到隐性位）和隐性位错误（发送隐性位但检测到显性位）。CAN 节点在发送状态会出现此类错误。
  但是在仲裁场（ARBITRATION FIELD）的填充位流期间或应答间隙（ACK SLOT）发送一“隐性”位的情况是例外的—— 此时，当监视到一“显性”位时，不会发出位错误。当发送器发送一个被动错误标志但检测到“显性”位时，也不视为位错误。

- **位填充错误（Stuff Error）**

  如果在使用一帧报文的位填充区域（参考图 4 的“位填充区域”）检测到 6 个连续相同的位电平时，将检测到一个位填充错误。CAN 节点在接收状态会出现此类错误。

- **CRC 错误（CRC Error）**

  CRC 序列包括发送器的 CRC 计算结果。接收器计算 CRC 的方法与发送器相同。如果计算结果与接收到 CRC 序列的结果不相符，则检测到一个 CRC 错误。CAN 节点在接收状态会出现此类错误。

- **格式错误（Form Error）**

  当一个固定形式的位场含有 1 个或多个非法位，则检测到一个格式错误。例如在 CRC 间隔符/ACK 间隔符的位场检测到显性位，则会检测到格式错误。例外：接收器的帧末尾最后一位期间的显性位不被当作帧错误。CAN 节点在接收状态会出现此类错误。

- **应答错误（Acknowledgment Error）**

  只要在应答位（ACK SLOT）期间所监测的位不为“显性”，则发送器会检测到一个应答错误。CAN 节点在发送状态会出现此类错误。

2.9.2 错误状态

CAN 节点检测到错误之后，根据不同状态和错误类型会对发送错误计数器（TEC[7:0]）/接收错误计

数器（REC[7:0]）进行加 1 或加 8（具体增加规则请参考 BOSCH CAN 协议），每正确的发送/接收一帧数据后，发送/接收错误计数器减 1。因此发送/接收错误计数器值表明了 CAN 节点和网络的稳定程度。根据发送/接收错误计数器值，一个节点的状态会处于以下三种之一：

- **错误主动**

“错误主动”的节点可以正常地参与总线通讯并在错误被检测到时发出主动错误标志（6 个显性位）。见下图 8，TEC<128 且 REC<128 即为错误主动状态。

- **错误被动**

“错误被动”的节点可参与总线接收和发送数据/远程帧。但检测到错误时只能发送错误被动标志（6 个隐形位）。见下图 8，255≥TEC>128 且 255≥REC>128 即为错误被动状态。

- **离线**

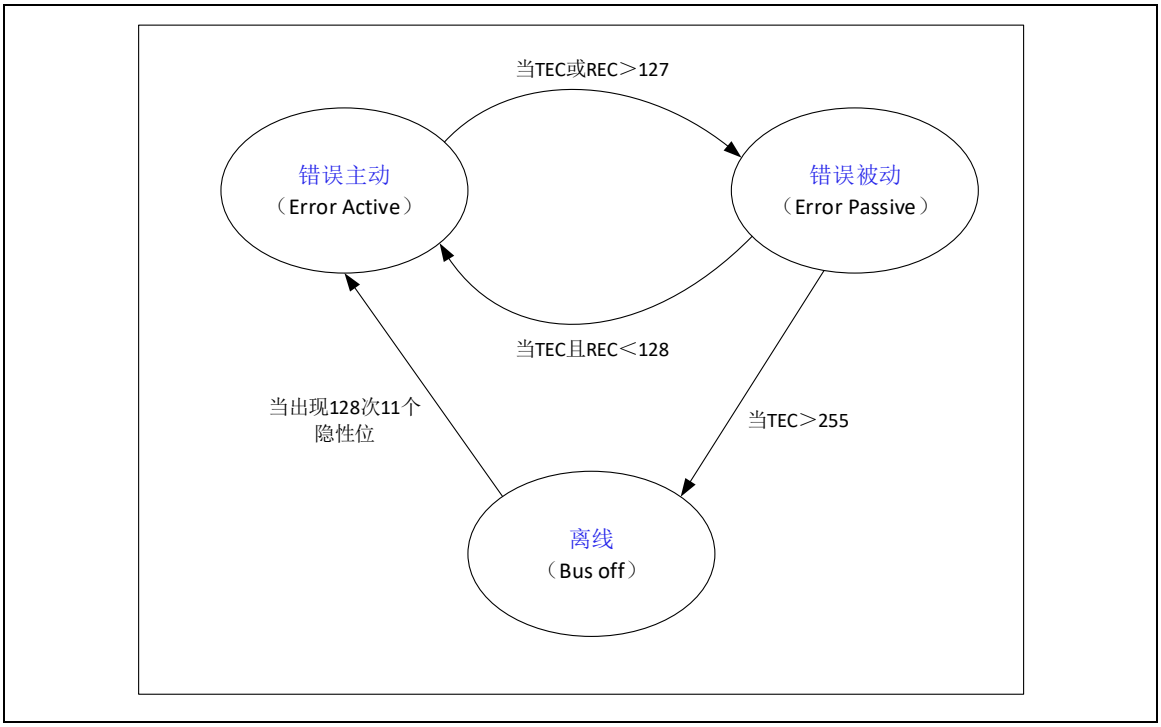
“离线”的节点相当于直接从 CAN 总线断开，不能收/发任何信息。见下图 8， TEC>255 即为离线状态。

AT32 离线管理：AT32 CAN 从离线状态恢复分两种情况：

1） 当 CAN 主控制寄存器（CAN_MCTRL）AEBOEN 位为‘0’时，需要软件请求进入冻结模式，再请求退出冻结模式，然后在通信模式下等待 CAN 节点 RX 检测到 128 次 11 个连续隐性位，随后该节点会从离线状态恢复。

2） 当 AEBOEN 位为‘1’时，通信模式下 CAN 节点 RX 检测到 128 次 11 个连续隐性位，就自动从离线状态恢复。

图 8 错误状态



3 AT32 的 CAN

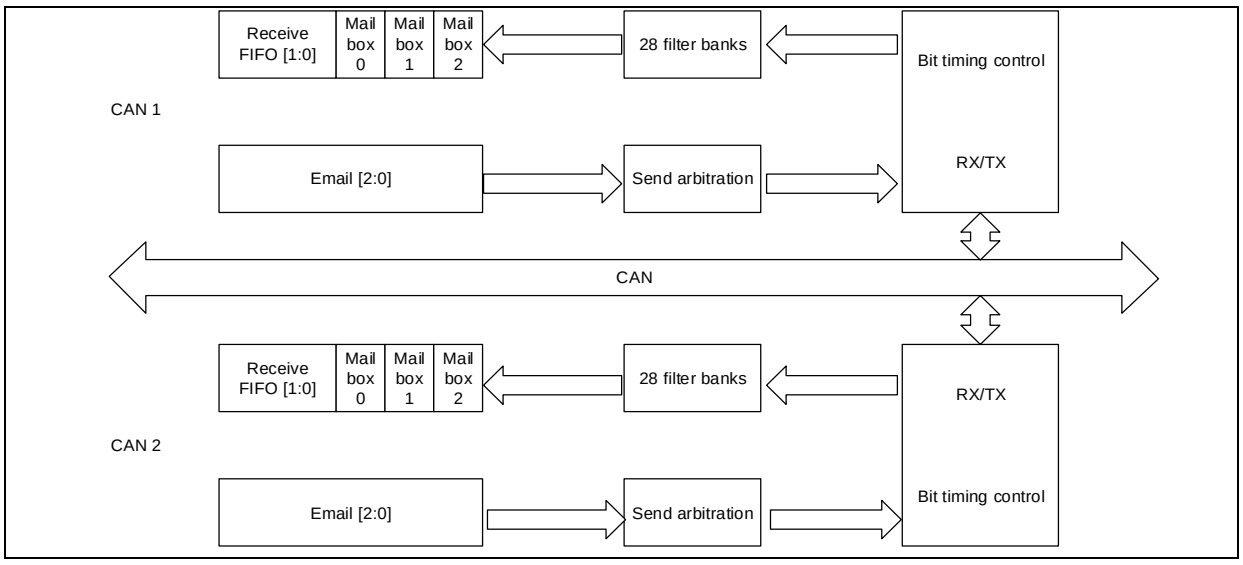
AT32 的 CAN 支持标准 CAN 协议 2.0A 和 2.0B。且在兼容标准 CAN 协议的基础上增加了一些功能和可配置选项。其中 CAN 2.0A 和 2.0B 的主要差别在于：CAN 2.0A 仅支持 11bit ID，即只支持标准帧；CAN 2.0B 支持 11bit/29bit ID，即支持标准帧和扩展帧。

本章节主要介绍 AT32 CAN 的主要设计结构和使用，介绍了 AT32 CAN 的正常通信流程，包括发送流程、接收流程、报文过滤、波特率及采样点设置等。其他 AT32 CAN 相关设计，例如错误管理、中断管理等，请参考 RM 相关章节。

3.1 整体功能介绍

随着 CAN 网络节点和报文数量的增加，需要一个增强的过滤机制处理各种类型的报文，减少接收报文的处理时间，采用 FIFO 的方案，使得 CPU 可以长时间处理应用层任务而不会丢失报文。同时发送报文由硬件控制发送优先级顺序。基于以上考虑，CAN 控制器提供 28 组位宽可配置的标识符过滤器组，2 个接收 FIFO，每个 FIFO 都可以存放 3 个完整的报文。共有 3 个发送邮箱，发送调度器决定发送优先级顺序。整个收发过程完全由硬件管理，无需占用 CPU 资源。

图 9 AT32 CAN 整体功能介绍



3.2 CAN 发送流程

CAN 发送流程见下图 10 和以下的步骤：

用户使用时只需操作 1）~3）。4）~7）由硬件自动完成，无需用户代码参与，不占用 CPU 资源。

1）程序选择 1 个空置的邮箱（发送邮箱空标志 TMxEF=1）

2）将需要发送的报文写入对应的空邮箱。报文内容包含： ID、帧类型、数据长度和发送数据等

3）请求发送：将 CAN_TMIx 的 TMSR 位置 1

4）邮箱挂号（等待成为最高优先级）

5）预定发送（等待总线空闲）

6）发送

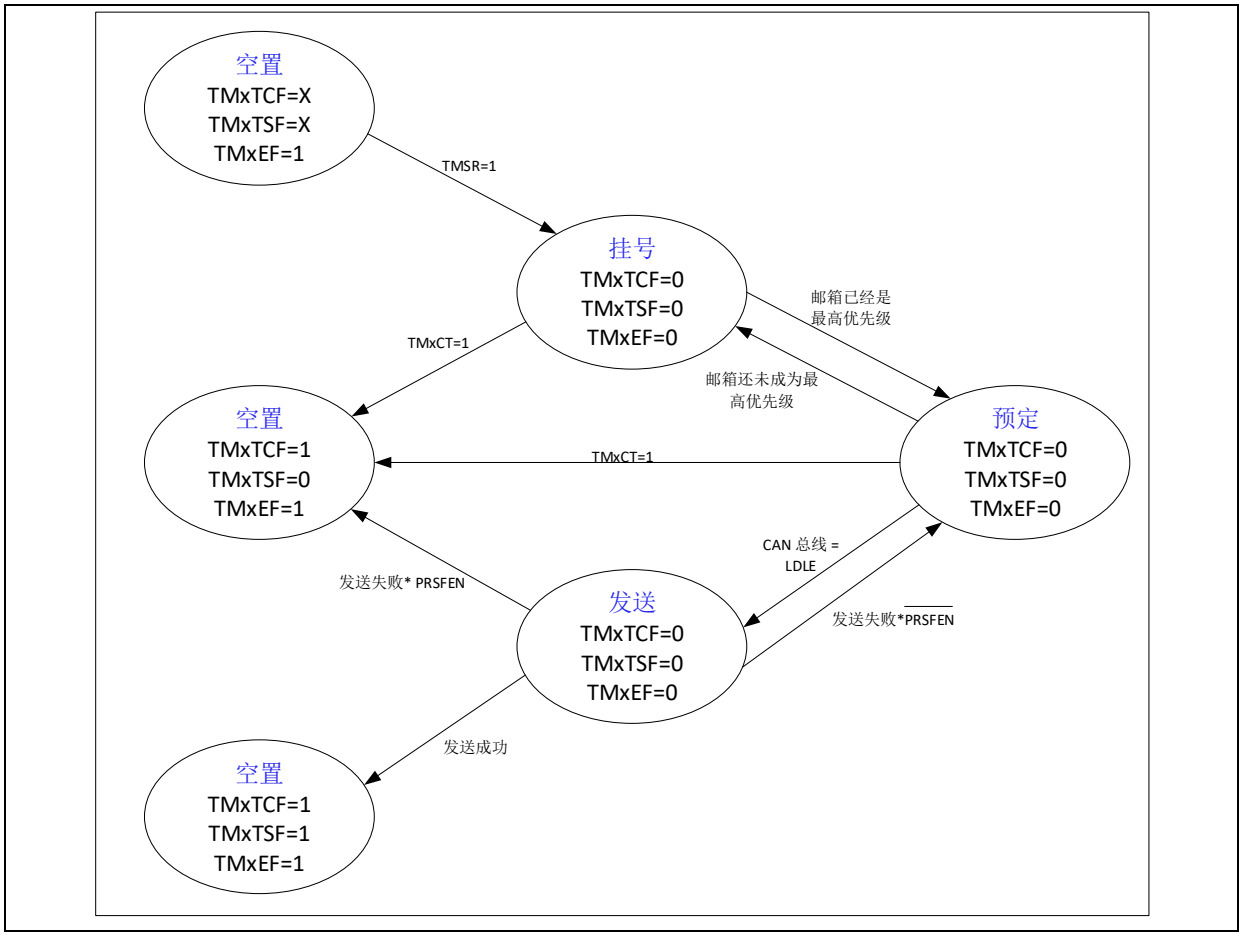
7）邮箱空置

注：以上步骤 1）~7）只简单介绍正常发送流程，下图 10 中还包含取消发送、发送失败、自动/不自动重传等情况，可参考 RM 文件报文发送一节，这里不再详述。

下图 10 中标志位和操作位说明如下：

TMxTCF：请求完成标志位（发送/中止请求）
TMxTSF：发送成功标志位
TMxEF：发送邮箱空标志位
TMSR：请求发送
TMxCT：中止发送
PRSFEN：禁止自动重传(PRSFEN =1 时，禁止自动重传；PRSFEN=0 时，自动重传直到发送成功)

图 10 CAN 发送流程



3.3 CAN 接收流程

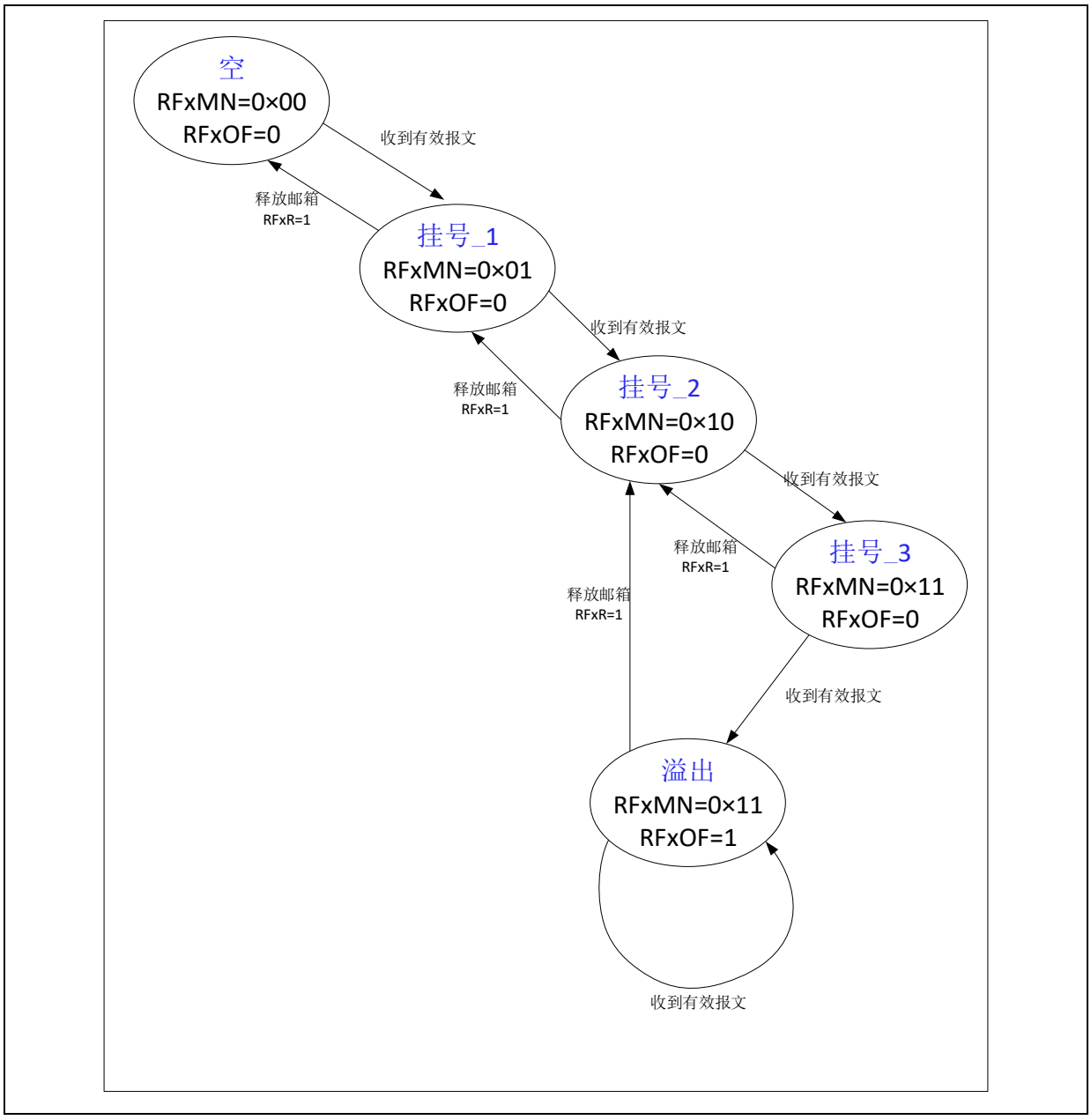
CAN 常用接收流程如下，即下图 11 的“空”和“挂号_1”两个状态间循环：
1）FIFO 空
2）收到有效报文
3）进入“挂号_1”状态（FIFO 内有 1 条有效报文的状态）
4）读取有效报文：读取接收邮箱寄存器（CAN_RFIx，CAN_RFCx，CAN_RFDTLx，CAN_RFDTHx）。
5）释放邮箱：CAN_RFx 寄存器 RFxR 位置 1。
注：用户使用时只需操作 4）~5）。1）~3）由硬件自动完成，无需用户代码参与，不占用 CPU 资源。

有效报文：
当报文被正确接收（直到 EOF 域的最后一位都没有错误），且通过了标识符过滤，那么该报文被认为是有效报文。过滤器相关介绍见下一节。

而如果接收过程中用户不参与操作（即不去读取有效报文和释放邮箱），硬件流程如下：
1）收到有效报文
2）进入“挂号_1”状态（FIFO 内有 1 条有效报文的状态）
3）收到有效报文
4）进入“挂号_2”状态（FIFO 内有 2 条有效报文的状态）
5）收到有效报文
6）进入“挂号_3”状态（FIFO 内有 3 条有效报文的状态）
7）收到有效报文
8）进入“溢出”状态（FIFO 内有 3 条有效报文，丢失了一条报文，溢出标志置起）

下图 11 中标志位和操作位说明如下：
RFxMN：FIFO 内有效报文数量（取值 0~3）
RFxOF：溢出标志位
RFxR：释放邮箱

图 11 CAN 接收流程



3.4 过滤器

在 CAN 协议里，报文的 ID 不代表节点的地址，而是跟报文的内容相关的。因此，发送者以广播的形式把报文发送给所有的接收者。节点在接收报文时，根据 ID 的值决定软件是否需要该报文；如果需要，就存到接收 FIFO 里，用户可通过软件读取接收邮箱寄存器获取该报文；如果不需要，报文就被丢弃且无需软件的干预。

为满足这一需求，AT32 CAN 控制器为应用程序提供了 28 个硬件过滤器组（AT32F435 系列有 28 个过滤器组，0~27；但 AT32F403A 等系列只有 14 个过滤器组，0~13。具体请参考相应型号的 RM），以便只接收那些软件需要的报文。用户配置好需要的 ID 后，整个过滤过程无需软件参与，不占用 CPU 资源。

过滤器的位宽

每个过滤器组由 2 个 32bit 的寄存器，CAN_FiFB1 和 CAN_FiFB2 组成。通过配置 CAN_FBWCFG 寄存器的 FBWSELx 位，可以设置 2 个 16 位宽或者 1 个 32 位宽的过滤器。

32 位宽的过滤器寄存器 CAN_FiFBx 包括：一组 SID[10：0]、EID[17：0]、IDT 和 RTR 位。

16 位宽的过滤器寄存器 CAN_FiFBx 包括：两组 SID[10：0]、IDT、RTR 和 EID[17：15]位。

过滤器模式

通过设置 CAN_FMCFG 寄存器的 FMSELx 位可以设置过滤器寄存器工作在标识符掩码模式或者标识符列表模式，掩码模式用来指定 ID 的哪些位需要与预设 ID 相同，哪些位无需比较，列表模式表示 ID 的每个位都必须与预设 ID 一致。

两种模式与过滤器位宽配合使用，可以有以下四种过滤方式：

图 12 32 位宽标识符掩码模式

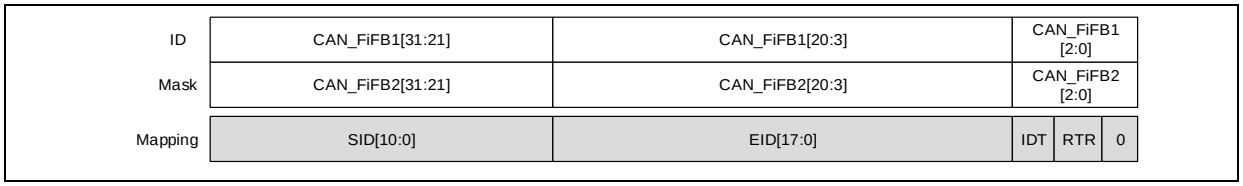


图 13 32 位宽标识符列表模式

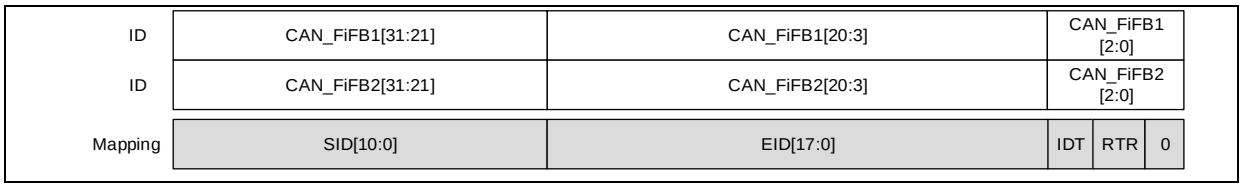


图 14 16 位宽标识符掩码模式

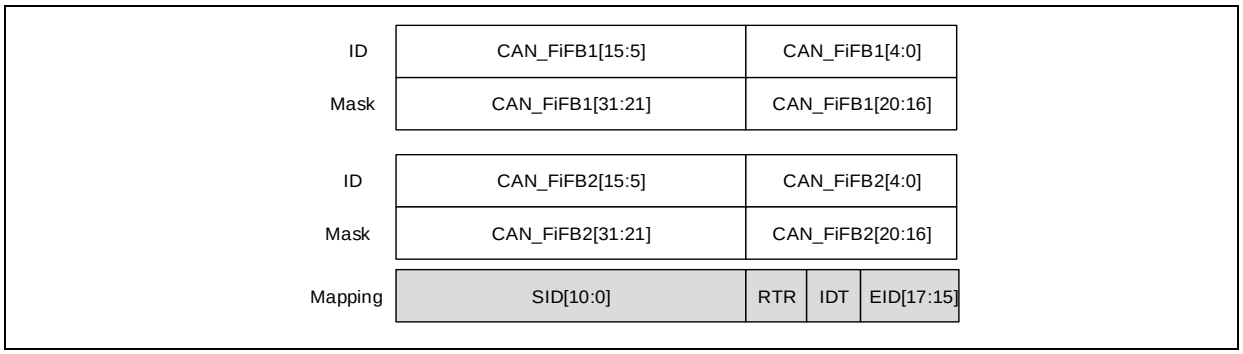
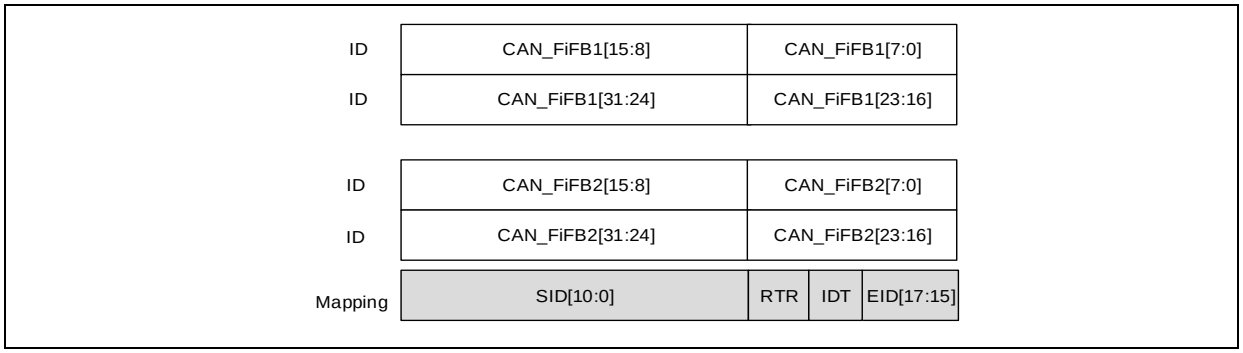


图 15 16 位宽标识符列表模式



更多 CAN 过滤器说明，例如 CAN 过滤器匹配序号，优先级规则等可参考 RM 文件报文过滤一节，这里不再详述。过滤器配置流程见后文案例介绍 -- CAN 接收过滤器使用。

3.5 CAN 波特率及采样点计算

如前文 CAN 位格式一节所述，CAN 的一个 bit 被分为几段。其中第一段同步段（SYNC_SEG）固定为 1Tq，1Tq 的长度由 CAN 位时序寄存器 （CAN_BTMG）的分频系数 BRDIV[11：0]位定义；位段

1（BSEG1）通过配置 CAN 位时序寄存器的 BTS1[3：0]位，可设定为 1~16Tq；位段 2（BSEG2）通过配置 CAN 位时序寄存器的 BTS2[2：0]位，可设定为 1~8Tq。用户通过配置 CAN 时序寄存器，可设置 CAN 波特率和采样点，整个 CAN 总线上各节点的波特率和采样点一致最佳，不过由于各节点主频可能不一样，所以比较难保证波特率和采样点均一致。用户使用时应首先保证波特率一致，采样点尽量保持在较小的偏差范围内，这样 CAN 总线可以支持更多的节点和更长的线路。

3.5.1 波特率计算公式

$$BaudRate = \frac{1}{\text{Nomal Bit Timimg}}$$

$$Nomal\ Bit\ Timing = t_{SYNC\_SEG} + t_{BSEG1} + t_{BSEG2}$$

其中

$$t_{SYNC\_SEG} = 1\ x\ t_q$$

$$t_{BSEG1} = (1 + \text{BTS1}[3:0])\ x\ t_q$$

$$t_{BSEG2} = (1 + \text{BTS2}[2:0])\ x\ t_q$$

$$t_q = (1 + \text{BRDIV}[11:0])\ x\ t_{pclk}$$

例如，bsp 例程 project\at_start_f437\examples\can\communication_mode：
APB 时钟：APB1_CLK = 144MHZ
CAN 分频系数：BRDIV = 12

此时$1\text{Tq} = \frac{1}{144\text{MHZ}/12} = \frac{1}{12}us$

同步段：SYNC_SEG = 1Tq（固定不变，无需用户配置）
位段 1：BSEG1 = 8Tq（BTS1[3：0] = 7）
位段 2：BSEG2 = 3Tq（BTS2[2：0] = 2）
此时$Nomal\ Bit\ Timimg = 1\text{Tq} * (\text{SYNC}_{\text{SEG}} + \text{BSEG1} + \text{BSEG12}) = 1us$

此时$BaudRate = \frac{1}{\text{Nomal Bit Timimg}} = \frac{1}{1\text{us}} = 1M\text{bps}$

3.5.2 采样点计算公式

$$sample\ point = \frac{\text{SYNC\_SEG} + \text{BSEG1}}{\text{SYNC\_SEG} + \text{BSEG1} + \text{BSEG2}}$$

举例同上：

此时$sample\ point = \frac{1+8}{1+8+3} = 75\%$

关于采样点设置，CAN 协议并没有明确规定，但根据各厂商 CAN 设备使用习惯，采样点设置建议如下表 3：

表 3 采样点设置建议

波特率	采样点建议
>800Kbps	75%
>500Kbps	80%
<=500Kbps	87.5%

3.5.3 波特率计算工具

为方便用户波特率设定，本文介绍一个 AT 专用波特率计算工具：Artery_CAN_BitRate_Configuration。

使用步骤如下：

1） **波特率设定**：高速 CAN 波特率最大为 1M，各厂商 CAN 设备常用波特率为 125K、250K、333K、500K、1M 等。用户可根据需要设定波特率。参考下图 16“波特率(Kbit/S)”。

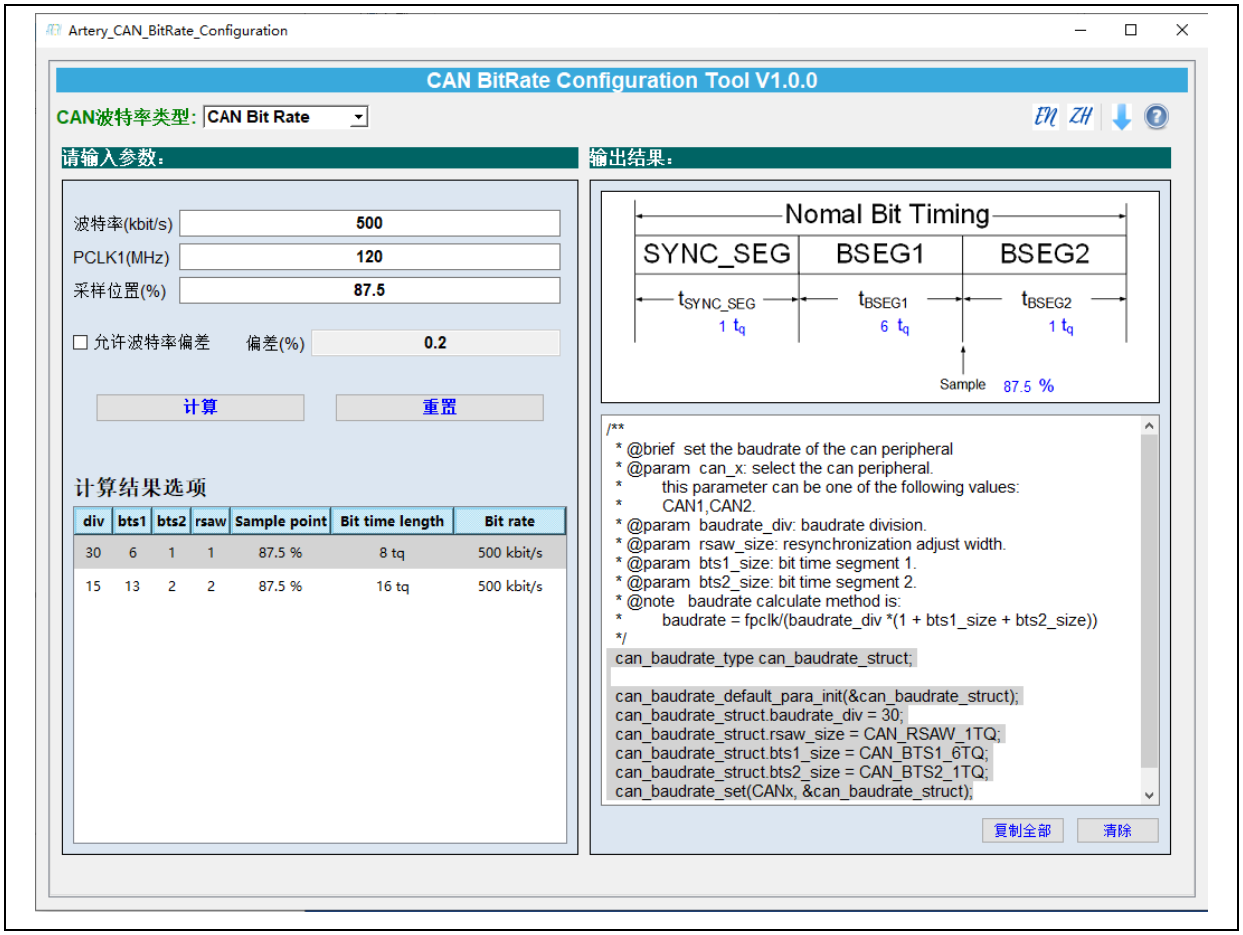
2） **CAN 时钟源频率设定**：参考下图 16“PCLK1(MHZ)”。

3） **采样位置设置**：设置完波特率后，计算工具会自动填入一个推荐的采样位置值。若实际项目中无具体限定，可保持默认设定；若项目中有具体限定，根据需求更改即可。参考下图 16“采样位置(%)”。

4） **波特率偏差设定**：建议在不勾选“允许波特率偏差”项，仅在没有符合要求的计算结果时，再勾选此项。由于同一 CAN 网络的节点波特率有误差会增大通信错误几率，建议“偏差”值设置尽量小。参考下图 16“允许波特率偏差”和“偏差”。

5） **波特率配置结果选择**：根据以上设定即可计算出多组结果。在页面左下角选择一项计算结果，即会在页面右下角显示对应软件代码配置，点击“复制全部”即可获得对应代码。

图 16 波特率配置工具



4 案例 1 CAN 正常通信—normal 模式

注：所有project都是基于keil 5而建立，若用户需要在其他编译环境上使用，请参考AT32xxx_Firmware_Library_V2.x.x\project\at_start_xxx\templates中各种编译环境（例如IAR6/7,keil 4/5）进行简单修改即可。

4.1 功能简介

实现两个 CAN 节点之间收发通信。

4.2 资源准备

1) 硬件环境：

两套对应产品型号的 AT-START BOARD + CAN 电平转化器

程序设计以 bsp demo 为例：

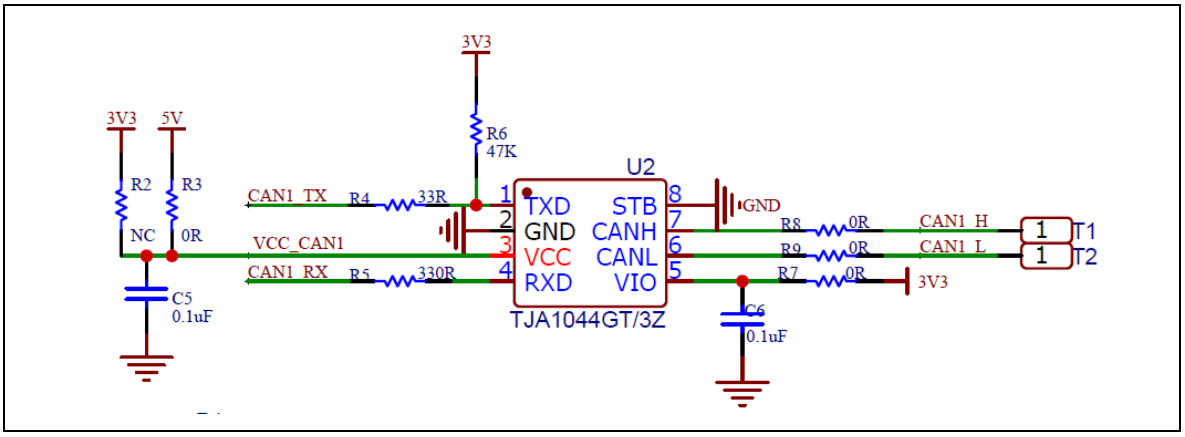
CAN1_TX(PB9)连接电平转换器的 TXD；

CAN1_RX(PB8)连接电平转换器的 RXD；

两个 CAN 节点的电平转换器的 CANH 和 CANL 分别相连。

CAN 电平转换器硬件设计可参考下图：

图 17 CAN 电平转换器硬件设计



2) 软件环境：

project\at_start_f435\examples\can\communication_mode

4.3 软件设计

1) 配置流程

- 配置 CAN1 TX 和 RX pin 对应的 GPIO 引脚
- 配置 CAN 基础选项
- 配置 CAN 波特率
- 配置 CAN 过滤器
- 配置 CAN 中断

2) 代码介绍

- main 函数代码描述

```
int main(void)
{
  system_clock_config(); /* 系统时钟配置 */
  at32_board_init();/*AT-START BOARD 基本配置，例如 LED 初始化，delay 初始化等 */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4); /*中断组优先级配置 */
  can_gpio_config();/*CAN1 对应 GPIO 引脚配置 */
  can_configuration(); /* CAN 配置：包括 CAN 基础选项、波特率、过滤器配置、中断配置 */
  while(1)
  {
    can_transmit_data(); /* CAN1 发送一帧标准数据帧 */
    at32_led_toggle(LED4); /* 翻转 LED4 */
    delay_sec(1); /* delay 1 秒 */
  }
}
```

■ CAN 配置函数代码描述

```
static void can_configuration(void)
{
  can_base_type can_base_struct;
  can_baudrate_type can_baudrate_struct;
  can_filter_init_type can_filter_init_struct;

  /* enable the can clock */
  crm_periph_clock_enable(CRM_CAN1_PERIPH_CLOCK, TRUE); /* 使能 CAN 时钟 */

  /* can 基础配置 */
  can_default_para_init(&can_base_struct); /* 初始化 CAN 基础配置结构体 */
  can_base_struct.mode_selection = CAN_MODE_COMMUNICATE; /* CAN 模式：正常通信模式 */
  can_base_struct.ttc_enable = FALSE; /* CAN 时间触发模式（时间戳）：关闭 */
  can_base_struct.aebo_enable = TRUE; /* 自动退出离线功能：开启 */
  can_base_struct.aed_enable = TRUE; /* 自动退出睡眠功能：开启 */
  can_base_struct.prsf_enable = FALSE; /* 禁止自动重传：关闭（即开启自动重传—同 CAN 标准协议） */
  can_base_struct.mdrsel_selection = CAN_DISCARDING_FIRST_RECEIVED; /* 报文溢出丢弃规则：丢弃之前收到的报文，保留最新收到的报文 */
  can_base_struct.mmssr_selection = CAN_SENDING_BY_ID; /* 报文发送优先级：根据 ID（ID 小的先发送） */
  can_base_init(CAN1, &can_base_struct); /* 将以上配置写入 CAN 主控制寄存器 */

  /* can 波特率配置：
  can baudrate, set boudrate = pclk/(baudrate_div *(1 + bts1_size + bts2_size))
  此处 pclk=144M；
  因此 boudrate = 144/（12*（1+8+3））=1Mbps
   */
  can_baudrate_struct.baudrate_div = 12; /* CAN 分频系数：12 */
```

```
  can_baudrate_struct.rsaw_size = CAN_RSAW_1TQ; /* CAN 同步跳跃宽度：1Tq */
  can_baudrate_struct.bts1_size = CAN_BTS1_8TQ; /* CAN 位段 1 长度：8Tq */
  can_baudrate_struct.bts2_size = CAN_BTS2_3TQ; /* CAN 位段 2 长度：3Tq */
  can_baudrate_set(CAN1, &can_baudrate_struct); /* 将以上配置写入 CAN 位时序寄存器 */
  /* can 过滤器配置 */
  can_filter_init_struct.filter_activate_enable = TRUE; /* 使能该过滤器 */
  can_filter_init_struct.filter_mode = CAN_FILTER_MODE_ID_MASK; /* 过滤器模式：掩码模式 */
  can_filter_init_struct.filter_fifo = CAN_FILTER_FIFO0; /* 使用哪个 FIFO（FIFO1/FIFO1）：FIFO0 */
  can_filter_init_struct.filter_number = 0; /* 使用哪个过滤器组（0~27）：0 */
  can_filter_init_struct.filter_bit = CAN_FILTER_32BIT; /* 过滤器位宽：32bit 宽度 */
  can_filter_init_struct.filter_id_high = 0; /* 过滤器 ID 高位设定：0 */
  can_filter_init_struct.filter_id_low = 0; /* 过滤器 ID 低位设定：0 */
  can_filter_init_struct.filter_mask_high = 0; /* 过滤器掩码高位设定：0（即所有位都不关心，所有 ID
都可以通过） */
  can_filter_init_struct.filter_mask_low = 0; /* 过滤器掩码低位设定：0（即所有位都不关心，所有 ID
都可以通过） */
  can_filter_init(CAN1, &can_filter_init_struct); /* 将以上配置写入过滤器相关寄存器 */

  /* can 中断配置 */
  nvic_irq_enable(CAN1_SE_IRQn, 0x00, 0x00); /* 中断优先级配置及使能：CAN1 状态变化/错误中
断 */
  nvic_irq_enable(CAN1_RX0_IRQn, 0x00, 0x00); /* 中断优先级配置及使能：CAN1 FIFO0 接收中断
*/
  can_interrupt_enable(CAN1, CAN_RF0MIEN_INT, TRUE); /* CAN1 FIFO0 非空中断使能：FIFO0 接
收到一帧有效数据即产生中断 */
  can_interrupt_enable(CAN1, CAN_ETRIEN_INT, TRUE); /* 错误类型记录中断使能：ETR[2:0]不为
0 时产生中断 */
  can_interrupt_enable(CAN1, CAN_EOIEN_INT, TRUE); /* CAN1 错误中断使能：所有错误中断的总
开关 */
}
```

■ CAN 发送函数代码描述

```
static void can_transmit_data(void)
{
  uint8_t transmit_mailbox;
  can_tx_message_type tx_message_struct;
  tx_message_struct.standard_id = 0x400; /* 设置发送数据帧的 ID=0x400 */
  tx_message_struct.extended_id = 0;
  tx_message_struct.id_type = CAN_ID_STANDARD; /* 发送数据帧类型（标准/扩展）：标准数据帧
*/
  tx_message_struct.frame_type = CAN_TFT_DATA; /* 发送帧类型（远程/数据）：数据帧 */
  tx_message_struct.dlc = 8; /* 发送数据长度（0~8）：8 */
  tx_message_struct.data[0] = 0x11; /* 第 1byte 数据：0x11 */
  tx_message_struct.data[1] = 0x22; /* 第 2byte 数据：0x22 */
  tx_message_struct.data[2] = 0x33; /* 第 3byte 数据：0x33 */
```

```
  tx_message_struct.data[3] = 0x44; /* 第 4byte 数据：0x44 */
  tx_message_struct.data[4] = 0x55; /* 第 5byte 数据：0x55 */
  tx_message_struct.data[5] = 0x66; /* 第 6byte 数据：0x66 */
  tx_message_struct.data[6] = 0x77; /* 第 7byte 数据：0x77 */
  tx_message_struct.data[7] = 0x88; /* 第 8byte 数据：0x88 */
  transmit_mailbox = can_message_transmit(CAN1, &tx_message_struct); /* 将以上待发送报文写入发送邮箱并请求发送 */
  while(can_transmit_status_get(CAN1, (can_tx_mailbox_num_type)transmit_mailbox) != CAN_TX_STATUS_SUCCESSFUL); /* 等待该邮箱发送成功—对应邮箱发送成功标志置起 */
}
```

■ CAN 接收中断函数代码描述

```
void CAN1_RX0_IRQHandler(void)
{
  can_rx_message_type rx_message_struct;
  if(can_flag_get(CAN1,CAN_RF0MN_FLAG) != RESET) /* 判断 FIFO0 非空（报文数量>0） */
  {
    can_message_receive(CAN1, CAN_RX_FIFO0, &rx_message_struct); /* 读取一帧报文：包含 ID, 数据长度，数据等信息 */
    if(rx_message_struct.standard_id == 0x400) /* 判断收到的报文是否为 ID=0x400 的标准帧 */
      at32_led_toggle(LED2); /* 如果收到报文为 ID=0x400 的标准帧，则翻转 LED2 */
    else
      at32_led_toggle(LED3); /* 否则翻转 LED3 */
  }
}
```

■ GPIO 配置函数代码描述

```
static void can_gpio_config(void)
{
  gpio_init_type gpio_init_struct;
  /* enable the gpio clock */
  crm_periph_clock_enable(CRM_GPIOB_PERIPH_CLOCK, TRUE); /* 使能对应 GPIOB 时钟 */
  gpio_default_para_init(&gpio_init_struct); /* 初始化 GPIO 配置结构体 */

  /* configure the can tx, rx pin */
  gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER; /* GPIO 驱动能力：强 */
  gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL; /* FPIO 输出模式：推挽输出 */
  gpio_init_struct.gpio_mode = GPIO_MODE_MUX; /* GPIO 模式：复用模式 */
  gpio_init_struct.gpio_pins = GPIO_PINS_9 | GPIO_PINS_8; /* GPIO 引脚：pin8 & pin9 */
  gpio_init_struct.gpio_pull = GPIO_PULL_NONE; /* GPIO 上下拉：无上下拉 */
  gpio_init(GPIOB, &gpio_init_struct); /* 将以上配置写入对应寄存器 */

  gpio_pin_mux_config(GPIOB, GPIO_PINS_SOURCE9, GPIO_MUX_9); /* 配置 GPIOB_pin9 mux9 功能—CAN_TX */
```

```
gpio_pin_mux_config(GPIOB, GPIO_PINS_SOURCE8, GPIO_MUX_9); /* 配置 GPIOB_pin8 mux9 功能—CAN_RX */
```

4.4 实验效果

- 如若数据传输无误，两块 AT-START BOARD 的 LED2 均会闪烁以指示收到 ID=0x400 的标准帧数据；LED4 会持续闪烁以指示程序正常运行。

5 案例 2 CAN 接收过滤器使用

注：所有project都是基于keil 5而建立，若用户需要在其他编译环境上使用，请参考AT32xxx_Firmware_Library_V2.x.x\project\at_start_xxx\templates中各种编译环境（例如IAR6/7,keil 4/5）进行简单修改即可。

5.1 功能简介

实现报文过滤：接收需要的报文，丢弃不需要的报文。

5.2 资源准备

1) 硬件环境：

两套对应产品型号的 AT-START BOARD + CAN 电平转化器

程序设计以 bsp demo 为例：

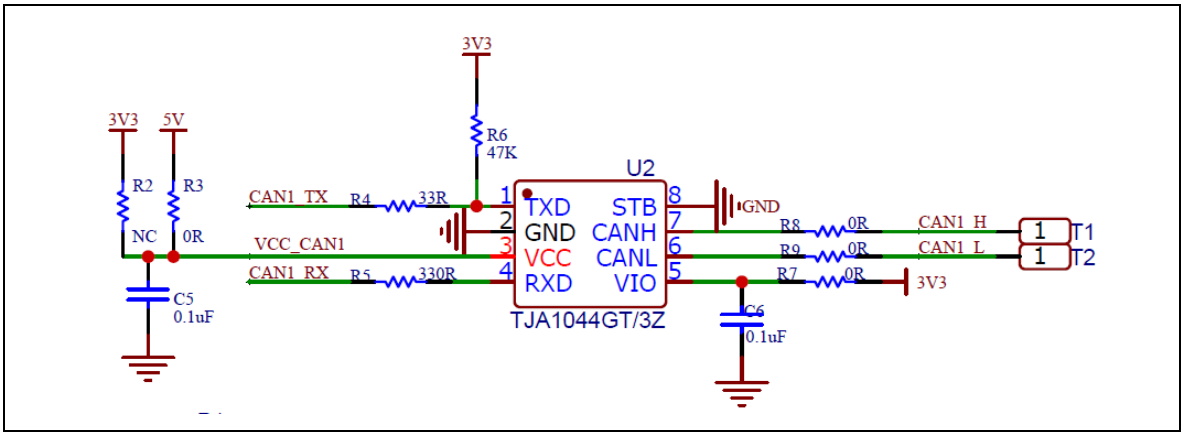
CAN1_TX(PB9)连接电平转换器的 TXD；

CAN1_RX(PB8)连接电平转换器的 RXD；

两个 CAN 节点的电平转换器的 CANH 和 CANL 分别相连。

CAN 电平转换器硬件设计可参考下图：

图 18 CAN 电平转换器硬件设计



2) 软件环境：

project\at_start_f435\examples\can\ filter

5.3 软件设计

1) 配置流程

- 配置 CAN1 TX 和 RX pin 对应的 GPIO 引脚
- 配置 CAN 基础选项
- 配置 CAN 波特率
- 配置 CAN 过滤器
- 配置 CAN 中断

2) 代码介绍

- 设定可通过过滤的 ID

```
/* extended identifier */
#define FILTER_EXT_ID1                  ((uint32_t)0x18F5F100)
#define FILTER_EXT_ID2                  ((uint32_t)0x18F5F200)

/* standard identifier */
#define FILTER_STD_ID1                  ((uint16_t)0x04F6)
#define FILTER_STD_ID2                  ((uint16_t)0x04F7)
```

- main 函数代码描述

```
int main(void)
{
  system_clock_config(); /* 系统时钟配置 */
  at32_board_init(); /* AT-START BOARD 基本配置，例如 LED 初始化，delay 初始化等 */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4); /* 中断组优先级配置 */
  can_gpio_config(); /* CAN1 对应 GPIO 配置 */
  can_configuration(); /* CAN 配置：包括 CAN 基础选项、波特率、过滤器配置、中断配置 */
can_transmit_data(); /* CAN1 发送 4 帧数据，ID 分别为：FILTER_EXT_ID1，FILTER_EXT_ID2，
FILTER_STD_ID1，FILTER_STD_ID2. */
  while(1)
  {
    if(test_result == 4)
    {
      at32_led_toggle(LED2); /* 如果收到 4 帧数据，则翻转 LED2/3/4 */
      at32_led_toggle(LED3);
      at32_led_toggle(LED4);
      delay_sec(1); /* delay 1 秒 */
    }
  }
}
```

- CAN 配置函数代码描述

```
static void can_configuration(void)
{
  can_base_type can_base_struct;
  can_baudrate_type can_baudrate_struct;
  can_filter_init_type can_filter_init_struct;

  /* enable the can clock */
  crm_periph_clock_enable(CRM_CAN1_PERIPH_CLOCK, TRUE); /* 使能 CAN 时钟 */

  /* can 基础配置 */
  can_default_para_init(&can_base_struct); /* 初始化 CAN 基础配置结构体 */
  can_base_struct.mode_selection = CAN_MODE_COMMUNICATE; /* CAN 模式：正常通信模式 */
  can_base_struct.ttc_enable = FALSE; /* CAN 时间触发模式（时间戳）：关闭 */
  can_base_struct.aebo_enable = TRUE; /* 自动退出离线功能：开启 */
```

```
  can_base_struct.aed_enable = TRUE; /* 自动退出睡眠功能：开启 */
  can_base_struct.prsf_enable = FALSE; /* 禁止自动重传：关闭（即开启自动重传—同 CAN 标准协议） */
  can_base_struct.mdrsel_selection = CAN_DISCARDING_FIRST_RECEIVED; /* 报文溢出丢弃规则：丢弃之前收到的报文，保留新收到的报文 */
  can_base_struct.mmssr_selection = CAN_SENDING_BY_ID; /* 报文发送优先级：根据 ID（ID 小的先发送） */
  can_base_init(CAN1, &can_base_struct); /* 将以上配置写入 CAN 主控制寄存器 */

  /* can 波特率配置：
  can baudrate, set boudrate = pclk/(baudrate_div *(1 + bts1_size + bts2_size))
  此处 pclk=144M；
  因此 boudrate = 144/（12*（1+8+3））=1Mbps
   */
  can_baudrate_struct.baudrate_div = 12; /* CAN 分频系数：12 */
  can_baudrate_struct.rsaw_size = CAN_RSAW_1TQ; /* CAN 同步跳跃宽度：1Tq */
  can_baudrate_struct.bts1_size = CAN_BTS1_8TQ; /* CAN 位段 1 长度：8Tq */
  can_baudrate_struct.bts2_size = CAN_BTS2_3TQ; /* CAN 位段 2 长度：3Tq */
  can_baudrate_set(CAN1, &can_baudrate_struct); /* 将以上配置写入 CAN 位时序寄存器 */
  /* can 过滤器配置 */
  /* can filter 0 config */
  can_filter_init_struct.filter_activate_enable = TRUE; /* 使能该过滤器 */
  can_filter_init_struct.filter_mode = CAN_FILTER_MODE_ID_LIST; /* 过滤器模式：列表模式 */
  can_filter_init_struct.filter_fifo = CAN_FILTER_FIFO0; /* 使用哪个 FIFO（FIFO1/FIFO1）：FIFO0 */
  can_filter_init_struct.filter_number = 0; /* 使用哪个过滤器组（0~27）：0 */
  can_filter_init_struct.filter_bit = CAN_FILTER_32BIT; /* 过滤器位宽：32bit 宽度 */
  can_filter_init_struct.filter_id_high = (((FILTER_EXT_ID1 << 3) >> 16) & 0xFFFF); /* 配置可通过过滤器的扩展 ID：FILTER_EXT_ID1 */
  can_filter_init_struct.filter_id_low = ((FILTER_EXT_ID1 << 3) & 0xFFFF) | 0x04;
  can_filter_init_struct.filter_mask_high = ((FILTER_EXT_ID2 << 3) >> 16) & 0xFFFF; /* 配置可通过过滤器的扩展 ID：FILTER_EXT_ID2 */
  can_filter_init_struct.filter_mask_low = ((FILTER_EXT_ID2 << 3) & 0xFFFF) | 0x04;
  can_filter_init(CAN1, &can_filter_init_struct); /* 将以上配置写入过滤器相关寄存器 */
  /* can filter 1 config */
  can_filter_init_struct.filter_activate_enable = TRUE; /* 使能该过滤器 */
  can_filter_init_struct.filter_mode = CAN_FILTER_MODE_ID_LIST; /* 过滤器模式：列表模式 */
  can_filter_init_struct.filter_fifo = CAN_FILTER_FIFO0; /* 使用哪个 FIFO（FIFO1/FIFO1）：FIFO0 */
  can_filter_init_struct.filter_number = 1; /* 使用哪个过滤器组（0~27）：1 */
  can_filter_init_struct.filter_bit = CAN_FILTER_32BIT; /* 过滤器位宽：32bit 宽度 */
  can_filter_init_struct.filter_id_high = FILTER_STD_ID1 << 5; /* 配置可通过过滤器的标准 ID：FILTER_STD_ID1 */
  can_filter_init_struct.filter_id_low = 0;
  can_filter_init_struct.filter_mask_high = FILTER_STD_ID2 << 5; /* 配置可通过过滤器的标准 ID：FILTER_STD_ID2 */
  can_filter_init_struct.filter_mask_low = 0;
  can_filter_init(CAN1, &can_filter_init_struct); /* 将以上配置写入过滤器相关寄存器 */
```

```
  /* can 中断配置 */
  nvic_irq_enable(CAN1_SE_IRQn, 0x00, 0x00); /* 中断优先级配置及使能：CAN1 状态变化/错误中断 */
  nvic_irq_enable(CAN1_RX0_IRQn, 0x00, 0x00); /* 中断优先级配置及使能：CAN1 FIFO0 接收中断 */
  can_interrupt_enable(CAN1, CAN_RF0MIEN_INT, TRUE); /* CAN1 FIFO0 非空中断使能：FIFO0 接收到一帧有效数据即产生中断 */
  can_interrupt_enable(CAN1, CAN_ETRIEN_INT, TRUE); /* 错误类型记录中断使能：ETR[2:0]不为 0 时产生中断 */
  can_interrupt_enable(CAN1, CAN_EOIEN_INT, TRUE); /* CAN1 错误中断使能：所有错误中断的总开关 */
}
```

■ CAN 发送函数代码描述

```
static void can_transmit_data(void)
{
  uint8_t transmit_mailbox;
  can_tx_message_type tx_message_struct;

  /* transmit FILTER_STD_ID1 */
  tx_message_struct.standard_id = FILTER_STD_ID1;/* 设置发送标准数据帧的 ID= FILTER_STD_ID1 */
  tx_message_struct.extended_id = 0;
  tx_message_struct.id_type = CAN_ID_STANDARD; /* 发送数据帧类型（标准/扩展）：标准数据帧 */
  tx_message_struct.frame_type = CAN_TFT_DATA; /* 发送帧类型（远程/数据）：数据帧 */
  tx_message_struct.dlc = 8; /* 发送数据长度（0~8）：8 */
  tx_message_struct.data[0] = 0x11; /* 第 1byte 数据：0x11 */
  tx_message_struct.data[1] = 0x22; /* 第 2byte 数据：0x22 */
  tx_message_struct.data[2] = 0x33; /* 第 3byte 数据：0x33 */
  tx_message_struct.data[3] = 0x44; /* 第 4byte 数据：0x44 */
  tx_message_struct.data[4] = 0x55; /* 第 5byte 数据：0x55 */
  tx_message_struct.data[5] = 0x66; /* 第 6byte 数据：0x66 */
  tx_message_struct.data[6] = 0x77; /* 第 7byte 数据：0x77 */
  tx_message_struct.data[7] = 0x88; /* 第 8byte 数据：0x88 */
  transmit_mailbox = can_message_transmit(CAN1, &tx_message_struct); /* 将以上待发送数据写入发送邮箱并请求发送 */
  while(can_transmit_status_get(CAN1, (can_tx_mailbox_num_type)transmit_mailbox) != CAN_TX_STATUS_SUCCESSFUL); /* 等待该邮箱发送成功—对应邮箱发送成功标志置起 */

  /* transmit FILTER_STD_ID2 */
  tx_message_struct.standard_id = FILTER_STD_ID2;/* 设置发送标准数据帧的 ID= FILTER_STD_ID2 */
  tx_message_struct.extended_id = 0;
  tx_message_struct.id_type = CAN_ID_STANDARD; /* 发送数据帧类型（标准/扩展）：标准数据帧
```

```
*/
  tx_message_struct.frame_type = CAN_TFT_DATA; /* 发送帧类型（远程/数据）：数据帧 */
  tx_message_struct.dlc = 8; /* 发送数据长度（0~8）：8 */
  tx_message_struct.data[0] = 0x11; /* 第 1byte 数据：0x11 */
  tx_message_struct.data[1] = 0x22; /* 第 2byte 数据：0x22 */
  tx_message_struct.data[2] = 0x33; /* 第 3byte 数据：0x33 */
  tx_message_struct.data[3] = 0x44; /* 第 4byte 数据：0x44 */
  tx_message_struct.data[4] = 0x55; /* 第 5byte 数据：0x55 */
  tx_message_struct.data[5] = 0x66; /* 第 6byte 数据：0x66 */
  tx_message_struct.data[6] = 0x77; /* 第 7byte 数据：0x77 */
  tx_message_struct.data[7] = 0x88; /* 第 8byte 数据：0x88 */
  transmit_mailbox = can_message_transmit(CAN1, &tx_message_struct); /* 将以上待发送数据写入
发送邮箱并请求发送 */
  while(can_transmit_status_get(CAN1, (can_tx_mailbox_num_type)transmit_mailbox) !=
CAN_TX_STATUS_SUCCESSFUL); /* 等待该邮箱发送成功—对应邮箱发送成功标志置起 */

  /* transmit FILTER_EXT_ID1 */
  tx_message_struct.standard_id = 0;
  tx_message_struct.extended_id = FILTER_EXT_ID1;/* 设置发送扩展数据帧的 ID=
FILTER_EXT_ID1 */
  tx_message_struct.id_type = CAN_ID_EXTENDED; /* 发送数据帧类型（标准/扩展）：扩展数据帧
*/
  tx_message_struct.frame_type = CAN_TFT_DATA; /* 发送帧类型（远程/数据）：数据帧 */
  tx_message_struct.dlc = 8; /* 发送数据长度（0~8）：8 */
  tx_message_struct.data[0] = 0x11; /* 第 1byte 数据：0x11 */
  tx_message_struct.data[1] = 0x22; /* 第 2byte 数据：0x22 */
  tx_message_struct.data[2] = 0x33; /* 第 3byte 数据：0x33 */
  tx_message_struct.data[3] = 0x44; /* 第 4byte 数据：0x44 */
  tx_message_struct.data[4] = 0x55; /* 第 5byte 数据：0x55 */
  tx_message_struct.data[5] = 0x66; /* 第 6byte 数据：0x66 */
  tx_message_struct.data[6] = 0x77; /* 第 7byte 数据：0x77 */
  tx_message_struct.data[7] = 0x88; /* 第 8byte 数据：0x88 */
  transmit_mailbox = can_message_transmit(CAN1, &tx_message_struct); /* 将以上待发送数据写入
发送邮箱并请求发送 */
  while(can_transmit_status_get(CAN1, (can_tx_mailbox_num_type)transmit_mailbox) !=
CAN_TX_STATUS_SUCCESSFUL); /* 等待该邮箱发送成功—对应邮箱发送成功标志置起 */

  /* transmit FILTER_EXT_ID2 */
  tx_message_struct.standard_id = 0;
  tx_message_struct.extended_id = FILTER_EXT_ID2;/* 设置发送扩展数据帧的 ID=
FILTER_EXT_ID2 */
  tx_message_struct.id_type = CAN_ID_EXTENDED; /* 发送数据帧类型（标准/扩展）：扩展数据帧
*/
  tx_message_struct.frame_type = CAN_TFT_DATA; /* 发送帧类型（远程/数据）：数据帧 */
  tx_message_struct.dlc = 8; /* 发送数据长度（0~8）：8 */
  tx_message_struct.data[0] = 0x11; /* 第 1byte 数据：0x11 */
```

```
  tx_message_struct.data[1] = 0x22; /* 第 2byte 数据：0x22 */
  tx_message_struct.data[2] = 0x33; /* 第 3byte 数据：0x33 */
  tx_message_struct.data[3] = 0x44; /* 第 4byte 数据：0x44 */
  tx_message_struct.data[4] = 0x55; /* 第 5byte 数据：0x55 */
  tx_message_struct.data[5] = 0x66; /* 第 6byte 数据：0x66 */
  tx_message_struct.data[6] = 0x77; /* 第 7byte 数据：0x77 */
  tx_message_struct.data[7] = 0x88; /* 第 8byte 数据：0x88 */
  transmit_mailbox = can_message_transmit(CAN1, &tx_message_struct); /* 将以上待发送数据写入发送邮箱并请求发送 */
  while(can_transmit_status_get(CAN1, (can_tx_mailbox_num_type)transmit_mailbox) != CAN_TX_STATUS_SUCCESSFUL); /* 等待该邮箱发送成功—对应邮箱发送成功标志置起 */
}
```

■ CAN 接收中断函数代码描述

```
void CAN1_RX0_IRQHandler(void)
{
  can_rx_message_type rx_message_struct;
  if(can_flag_get(CAN1,CAN_RF0MN_FLAG) != RESET) /* 判断 FIFO0 非空（报文数量>0） */
  {
    if(test_result == 4)
    {
      test_result = 0;
    }
    can_message_receive(CAN1, CAN_RX_FIFO0, &rx_message_struct); /* 读取一帧报文：包含 ID, 数据长度，数据等信息 */
    if((rx_message_struct.id_type == CAN_ID_STANDARD) && (rx_message_struct.standard_id == FILTER_STD_ID1))
      test_result++; /* 如果收到 ID= FILTER_STD_ID1 的标准帧，test_result 标志加 1 */
    else if((rx_message_struct.id_type == CAN_ID_STANDARD) && (rx_message_struct.standard_id == FILTER_STD_ID2))
      test_result++;/* 如果收到 ID= FILTER_STD_ID2 的标准帧，test_result 标志加 1 */
    else if((rx_message_struct.id_type == CAN_ID_EXTENDED) && (rx_message_struct.extended_id == FILTER_EXT_ID1))
      test_result++;/* 如果收到 ID= FILTER_EXT_ID1 的扩展帧，test_result 标志加 1 */
    else if((rx_message_struct.id_type == CAN_ID_EXTENDED) && (rx_message_struct.extended_id == FILTER_EXT_ID2))
      test_result++;/* 如果收到 ID= FILTER_EXT_ID2 的扩展帧，test_result 标志加 1 */
  }
}
```

■ GPIO 配置函数代码描述

```
static void can_gpio_config(void)
{
  gpio_init_type gpio_init_struct;
  /* enable the gpio clock */
```

```
  crm_periph_clock_enable(CRM_GPIOB_PERIPH_CLOCK, TRUE); /* 使能对应 GPIOB 时钟 */
  gpio_default_para_init(&gpio_init_struct); /* 初始化 GPIO 配置结构体 */

  /* configure the can tx, rx pin */
  gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER; /* GPIO 驱动能力：强 */
  gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL; /* FPIO 输出模式：推挽输出 */
  gpio_init_struct.gpio_mode = GPIO_MODE_MUX; /* GPIO 模式：复用模式 */
  gpio_init_struct.gpio_pins = GPIO_PINS_9 | GPIO_PINS_8; /* GPIO 引脚：pin8 & pin9 */
  gpio_init_struct.gpio_pull = GPIO_PULL_NONE; /* GPIO 上下拉：无上下拉 */
  gpio_init(GPIOB, &gpio_init_struct); /* 将以上配置写入对应寄存器 */

  gpio_pin_mux_config(GPIOB, GPIO_PINS_SOURCE9, GPIO_MUX_9); /* 配置 GPIOB_pin9 mux9 功能—CAN_TX */
  gpio_pin_mux_config(GPIOB, GPIO_PINS_SOURCE8, GPIO_MUX_9); /* 配置 GPIOB_pin8 mux9 功能—CAN_RX */
```

5.4 实验效果

- 如若数据传输无误，AT-START BOARD 的 LED2/3/4 会翻转一次，以指示收到 ID=FILTER_EXT_ID1，FILTER_EXT_ID2，FILTER_STD_ID1，FILTER_STD_ID2 的 4 帧数据。

6 案例 3 CAN 调试—loopback 模式

注：所有project都是基于keil 5而建立，若用户需要在其他编译环境上使用，请参考AT32xxx_Firmware_Library_V2.x.x\project\at_start_xxx\templates中各种编译环境（例如IAR6/7,keil 4/5）进行简单修改即可。

6.1 功能简介

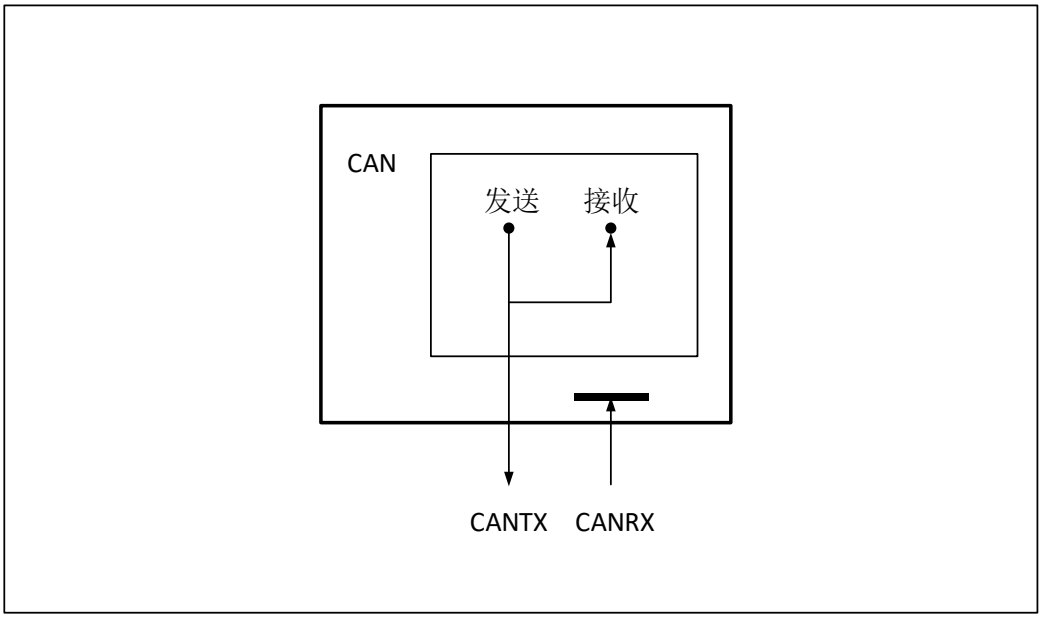
实现单板的环回模式通信。

环回模式（loopback mode）：

环回模式可用于自测试。在环回模式下，CAN 在内部把 TX 输出回馈到 RX 输入上，而完全忽略 CAN_RX 引脚的实际状态。因此此模式下 CAN 对应的 GPIO 引脚可以不配置，而如果对应的 GPIO 引脚配置了，发送的报文可以在 CAN_TX 引脚上检测到。见下图 19。另外，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）。

平转换器硬件设计可参考下图：

图 19 CAN loopback 模式



6.2 资源准备

1) 硬件环境：

一块对应产品型号的 AT-START BOARD

2) 软件环境：

project\at_start_f435\examples\can\loopback_mode

6.3 软件设计

1) 配置流程

- 配置 CAN1 TX 和 RX pin 对应 GPIO（loopback 模式下，此项可忽略不配置）
- 配置 CAN 基础选项
- 配置 CAN 波特率
- 配置 CAN 过滤器

- 配置 CAN 中断

2) 代码介绍

- main 函数代码描述

```
int main(void)
{
  system_clock_config(); /* 系统时钟配置 */
  at32_board_init(); /* AT-START BOARD 基本配置，例如 LED 初始化，delay 初始化等 */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4); /* 中断组优先级配置 */
  can_gpio_config(); /* CAN1 对应 GPIO 配置（loopback 模式下，此项可忽略不配。如果配置了，
CAN_TX pin 也会输出数据，但 CAN_RX pin 不会接收外部数据，参考图 16） */
  can_configuration(); /* CAN 配置：包括 CAN 基础选项、波特率、过滤器配置、中断配置 */
  while(1)
  {
    can_transmit_data(); /* CAN1 发送一帧标准数据帧 */
    at32_led_toggle(LED4); /* 翻转 LED4 */
    delay_sec(1); /* delay 1 秒 */
  }
}
```

- CAN 配置函数代码描述

```
static void can_configuration(void)
{
  can_base_type can_base_struct;
  can_baudrate_type can_baudrate_struct;
  can_filter_init_type can_filter_init_struct;

  /* enable the can clock */
  crm_periph_clock_enable(CRM_CAN1_PERIPH_CLOCK, TRUE); /* 使能 CAN 时钟 */

  /* can 基础配置 */
  can_default_para_init(&can_base_struct); /* 初始化 CAN 基础配置结构体 */
  can_base_struct.mode_selection = CAN_MODE_LOOPBACK; /* CAN 模式：环回模式 */
  can_base_struct.ttc_enable = FALSE; /* CAN 时间触发模式（时间戳）：关闭 */
  can_base_struct.aebo_enable = TRUE; /* 自动退出离线功能：开启 */
  can_base_struct.aed_enable = TRUE; /* 自动退出睡眠功能：开启 */
  can_base_struct.prsf_enable = FALSE; /* 禁止自动重传：关闭（即开启自动重传—同 CAN 标准协
议） */
  can_base_struct.mdrsel_selection = CAN_DISCARDING_FIRST_RECEIVED; /* 报文溢出丢弃规
则：丢弃之前收到的报文 */
  can_base_struct.mmssr_selection = CAN_SENDING_BY_ID; /* 报文发送优先级：根据 ID（ID 小的
先发送） */
  can_base_init(CAN1, &can_base_struct); /* 将以上配置写入 CAN 主控制寄存器 */

  /* can 波特率配置：
  can baudrate, set boudrate = pclk/(baudrate_div *(1 + bts1_size + bts2_size))
```

```
  此处 pclk=144M；
  因此 boudrate = 144/（12*（1+8+3））=1Mbps
   */
  can_baudrate_struct.baudrate_div = 12; /* CAN 分频系数：12 */
  can_baudrate_struct.rsaw_size = CAN_RSAW_1TQ; /* CAN 同步跳跃宽度：1Tq */
  can_baudrate_struct.bts1_size = CAN_BTS1_8TQ; /* CAN 位段 1 长度：8Tq */
  can_baudrate_struct.bts2_size = CAN_BTS2_3TQ; /* CAN 位段 2 长度：3Tq */
  can_baudrate_set(CAN1, &can_baudrate_struct); /* 将以上配置写入 CAN 位时序寄存器 */
  /* can 过滤器配置 */
  can_filter_init_struct.filter_activate_enable = TRUE; /* 使能该过滤器 */
  can_filter_init_struct.filter_mode = CAN_FILTER_MODE_ID_MASK; /* 过滤器模式：掩码模式 */
  can_filter_init_struct.filter_fifo = CAN_FILTER_FIFO0; /* 使用哪个 FIFO（FIFO1/FIFO1）：FIFO0 */
  can_filter_init_struct.filter_number = 0; /* 使用哪个过滤器组（0~27）：0 */
  can_filter_init_struct.filter_bit = CAN_FILTER_32BIT; /* 过滤器位宽：32bit 宽度 */
  can_filter_init_struct.filter_id_high = 0; /* 过滤器 ID 高位设定：0 */
  can_filter_init_struct.filter_id_low = 0; /* 过滤器 ID 低位设定：0 */
  can_filter_init_struct.filter_mask_high = 0; /* 过滤器掩码高位设定：0（即所有位都不关心，所有 ID 都可以通过） */
  can_filter_init_struct.filter_mask_low = 0; /* 过滤器掩码低位设定：0（即所有位都不关心，所有 ID 都可以通过） */
  can_filter_init(CAN1, &can_filter_init_struct); /* 将以上配置写入过滤器相关寄存器 */

  /* can 中断配置 */
  nvic_irq_enable(CAN1_SE_IRQn, 0x00, 0x00); /* 中断优先级配置及使能：CAN1 状态变化/错误中断 */
  nvic_irq_enable(CAN1_RX0_IRQn, 0x00, 0x00); /* 中断优先级配置及使能：CAN1 FIFO0 接收中断 */
  can_interrupt_enable(CAN1, CAN_RF0MIEN_INT, TRUE); /* CAN1 FIFO0 非空中断使能：FIFO0 接收到一帧有效数据即产生中断 */
  can_interrupt_enable(CAN1, CAN_ETRIEN_INT, TRUE); /* 错误类型记录中断使能：ETR[2:0]不为 0 时产生中断 */
  can_interrupt_enable(CAN1, CAN_EOIEN_INT, TRUE); /* CAN1 错误中断使能：所有错误中断的总开关 */
}
```

■ CAN 发送函数代码描述

```
static void can_transmit_data(void)
{
  uint8_t transmit_mailbox;
  can_tx_message_type tx_message_struct;
  tx_message_struct.standard_id = 0x400; /* 设置发送数据帧的 ID=0x400 */
  tx_message_struct.extended_id = 0;
  tx_message_struct.id_type = CAN_ID_STANDARD; /* 发送数据帧类型（标准/扩展）：标准数据帧 */
  tx_message_struct.frame_type = CAN_TFT_DATA; /* 发送帧类型（远程/数据）：数据帧 */
```

```
  tx_message_struct.dlc = 8; /* 发送数据长度（0~8）：8 */
  tx_message_struct.data[0] = 0x11; /* 第 1byte 数据：0x11 */
  tx_message_struct.data[1] = 0x22; /* 第 2byte 数据：0x22 */
  tx_message_struct.data[2] = 0x33; /* 第 3byte 数据：0x33 */
  tx_message_struct.data[3] = 0x44; /* 第 4byte 数据：0x44 */
  tx_message_struct.data[4] = 0x55; /* 第 5byte 数据：0x55 */
  tx_message_struct.data[5] = 0x66; /* 第 6byte 数据：0x66 */
  tx_message_struct.data[6] = 0x77; /* 第 7byte 数据：0x77 */
  tx_message_struct.data[7] = 0x88; /* 第 8byte 数据：0x88 */
  transmit_mailbox = can_message_transmit(CAN1, &tx_message_struct); /* 将以上待发送数据写入发送邮箱并请求发送 */
  while(can_transmit_status_get(CAN1, (can_tx_mailbox_num_type)transmit_mailbox) != CAN_TX_STATUS_SUCCESSFUL); /* 等待该邮箱发送成功—对应邮箱发送成功标志置起 */
}
```

■ CAN 接收中断函数代码描述

```
void CAN1_RX0_IRQHandler(void)
{
  can_rx_message_type rx_message_struct;
  if(can_flag_get(CAN1,CAN_RF0MN_FLAG) != RESET) /* 判断 FIFO0 非空（报文数量>0） */
  {
    can_message_receive(CAN1, CAN_RX_FIFO0, &rx_message_struct); /* 读取一帧报文：包含 ID, 数据长度，数据等信息 */
    if(rx_message_struct.standard_id == 0x400) /* 判断收到的报文是否为 ID=0x400 的标准帧 */
      at32_led_toggle(LED2); /* 如果收到报文为 ID=0x400 的标准帧，则翻转 LED2 */
    else
      at32_led_toggle(LED3); /* 否则翻转 LED3 */
  }
}
```

■ GPIO 配置函数代码描述

```
static void can_gpio_config(void)
{
  gpio_init_type gpio_init_struct;
  /* enable the gpio clock */
  crm_periph_clock_enable(CRM_GPIOB_PERIPH_CLOCK, TRUE); /* 使能对应 GPIOB 时钟 */
  gpio_default_para_init(&gpio_init_struct); /* 初始化 GPIO 配置结构体 */

  /* configure the can tx, rx pin */
  gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER; /* GPIO 驱动能力：强 */
  gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL; /* FPIO 输出模式：推挽输出 */
  gpio_init_struct.gpio_mode = GPIO_MODE_MUX; /* GPIO 模式：复用模式 */
  gpio_init_struct.gpio_pins = GPIO_PINS_9 | GPIO_PINS_8; /* GPIO 引脚：pin8 & pin9 */
  gpio_init_struct.gpio_pull = GPIO_PULL_NONE; /* GPIO 上下拉：无上下拉 */
```

```
  gpio_init(GPIOB, &gpio_init_struct); /* 将以上配置写入对应寄存器 */

  gpio_pin_mux_config(GPIOB, GPIO_PINS_SOURCE9, GPIO_MUX_9); /* 配置 GPIOB_pin9 mux9 功能—CAN_TX */
  gpio_pin_mux_config(GPIOB, GPIO_PINS_SOURCE8, GPIO_MUX_9); /* 配置 GPIOB_pin8 mux9 功能—CAN_RX */
```

6.4 实验效果

- AT-START BOARD 的 LED2 会闪烁以指示收到自己发送的 ID=0x400 的标准帧数据；LED4 会持续闪烁以指示程序正常运行。

7 文档版本历史

表 4 文档版本历史

日期	版本	变更
2021.5.20	2.0.0	最初版本
2022.7.19	2.0.1	1.更改CAN波特率计算工具章节，改为使用Artery_CAN_BitRate_Configuration工具。

重要通知 - 请仔细阅读

买方自行负责对本文所述雅特力产品和服务的选择和使用，雅特力概不承担与选择或使用本文所述雅特力产品和服务相关的任何责任。

无论之前是否有过任何形式的表示，本文档不以任何方式对任何知识产权进行任何明示或默示的授权或许可。如果本文档任何部分涉及任何第三方产品或服务，不应被视为雅特力授权使用此类第三方产品或服务，或许可其中的任何知识产权，或者被视为涉及以任何方式使用任何此类第三方产品或服务或其中任何知识产权的保证。

除非在雅特力的销售条款中另有说明，否则，雅特力对雅特力产品的使用和/或销售不做任何明示或默示的保证，包括但不限于有关适销性、适合特定用途（及其依据任何司法管辖区的法律的对应情况），或侵犯任何专利、版权或其他知识产权的默示保证。

雅特力产品并非设计或专门用于下列用途的产品：（A）对安全性有特别要求的应用，例如：生命支持、主动植入设备或对产品功能安全有要求的系统；（B）航空应用；（C）航天应用或航天环境；（D）武器，且/或（E）其他可能导致人身伤害、死亡及财产损害的应用。如果采购商擅自将其用于前述应用，即使采购商向雅特力发出了书面通知，风险及法律责任仍将由采购商单独承担，且采购商应独力负责在前述应用中满足所有法律和法规要求。

经销的雅特力产品如有不同于本文档中提出的声明和/或技术特点的规定，将立即导致雅特力针对本文所述雅特力产品或服务授予的任何保证失效，并且不应以任何形式造成或扩大雅特力的任何责任。