应用笔记

ACM32H5 系列芯片

FDCAN 应用笔记

版本：V0.2

日期：2024-12-11

上海航芯电子科技股份有限公司

1. 概述

本应用手册适用于 ACM32H5 系列芯片 FDCAN 模块。它描述了与 FDCAN 模块相关的设置和功能使用方法，以便在应用程序中进行优化设计。

本应用手册需要结合用户手册中对应的 FDCAN 模块部分、SDK 中相关的 demo 及 HAL 库驱动一起阅读。

2. CAN-FD 协议概述

随着总线技术在汽车电子领域越来越广泛和深入的应用，特别是自动驾驶技术的迅速发展，汽车电子对总线宽度和数据传输速率的要求也越来也高，传统 CAN（1MBit/s，8Bytes Payload）已难以满足日益增加的需求。因此在 2012 年，Bosch 发布了新的 CAN FD 标准（CAN with Flexible Data Rate），CAN FD 继承了 CAN 的绝大多数特性，如同样的物理层，双线串行通信协议，基于非破坏性仲裁技术，分布式实时控制，可靠的错误处理和检测机制等，同时 CAN FD 弥补了 CAN 在总线带宽和数据长度方面的不足。
2015 年 6 月 30 日，国际标准化组织（ISO）已经正式认可 CAN FD，并无反对票通过 ISO 11898-1 作为国际标准草案。下表为 CAN-FD 和 CAN2.0 之间的主要差异：

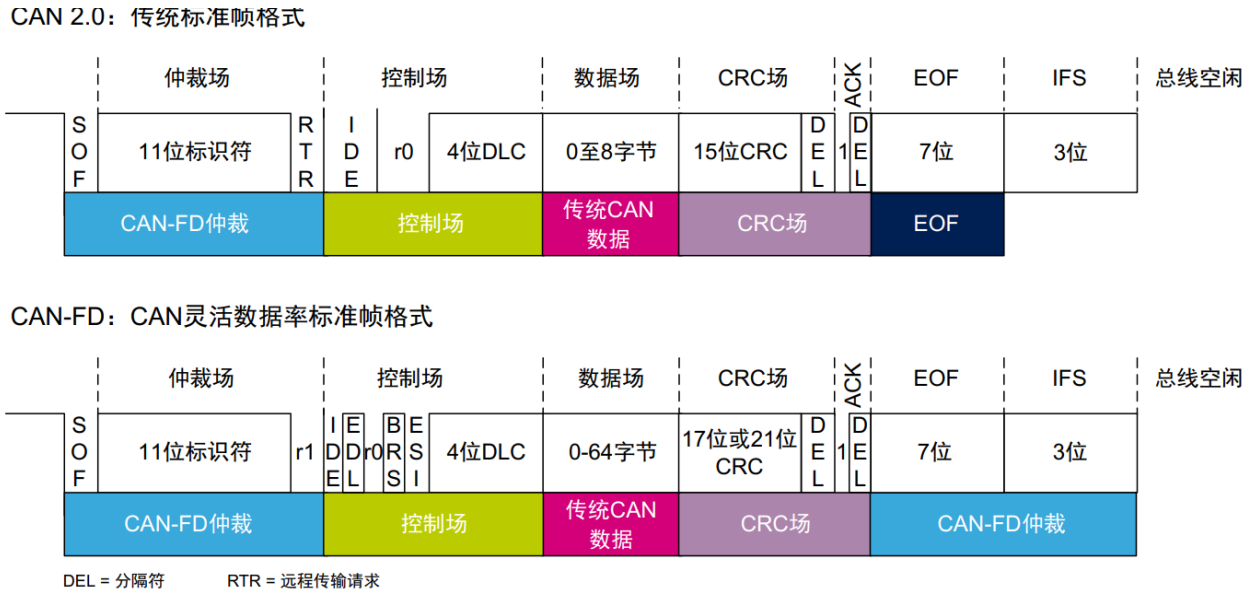
表 2-1 CAN-FD 与 CAN 2.0 之间的主要差异

特征	CAN 2.0	CAN-FD
兼容性	不支持 CAN-FD	支持 CAN 2.0 A/B
最大比特率（Mbit/s）	比特率：最高 1M	仲裁段比特率：最高 1M 数据段比特率：最高 8M
数据长度（DLC）	编码为 0 至 8	编码为 0 至 64①
支持 BRS	无	有
支持 FDF	无	有
CRC 位校验码	CRC 计算中不包括填充位	CRC 计算中包括填充位
远程帧支持	有	无

① CAN-FD DLC 编码，8 字节长度后面为非连续编码

CAN-FD 的帧格式相对 CAN2.0 也有差异：

图 2-1 CAN2.0 和 CAN-FD 帧结构对比



DLC 数据长度编码如下：

表 2-2 CAN-FD 与 CAN 2.0 DLC 编码

DLC（二进制）	帧类型	数据长度

0000~1000	CAN2.0 和 CAN-FD	0~8
1001~1111	CAN2.0	8
1001	CAN-FD	12
1010	CAN-FD	16
1011	CAN-FD	20
1100	CAN-FD	24
1101	CAN-FD	32
1110	CAN-FD	48
1111	CAN-FD	64

3. H5 FDCAN 的实现

3.1. 主要特性

- 支持 CAN2.0B 和 CAN-FD
- CAN2.0B 最多支持 1M 的波特率
- FDCAN 的波特率可以自由设置
- 可编程的分频器（1 到 256 分频）
- 接收 FIFO 可以容纳 16 帧数据
- 发送 FIFO 分为高优先级缓存（1 个 PTB）被低优先级缓存（16 个 STB）
- STB 支持 FIFO 模式和优先级模式
- 支持 16 组 29 比特的过滤组
- 支持 single-shot 发送模式
- 支持监听和回环模式
- 支持收发器低功耗模式
- 支持时间戳功能，支持 ISO11898-4 时间戳和 CiA603 时间戳
- 支持 AUTOSAR 和 SAEJ1939

3.2. 过滤器实现

由于 CAN 节点可以接收到总线上的所有帧，因此使用过滤器可以有效的减轻 CPU 的负载，只处理需要的相关帧数据。H5 FDCAN 共有 16 个过滤器组，每组过滤器共包含两个 32 位的寄存器，可自由配置指定过滤器组的过滤方式(code 方式/mask 方式)和过滤长度(半字 HalfWord/字 Word)。

过滤方式说明：

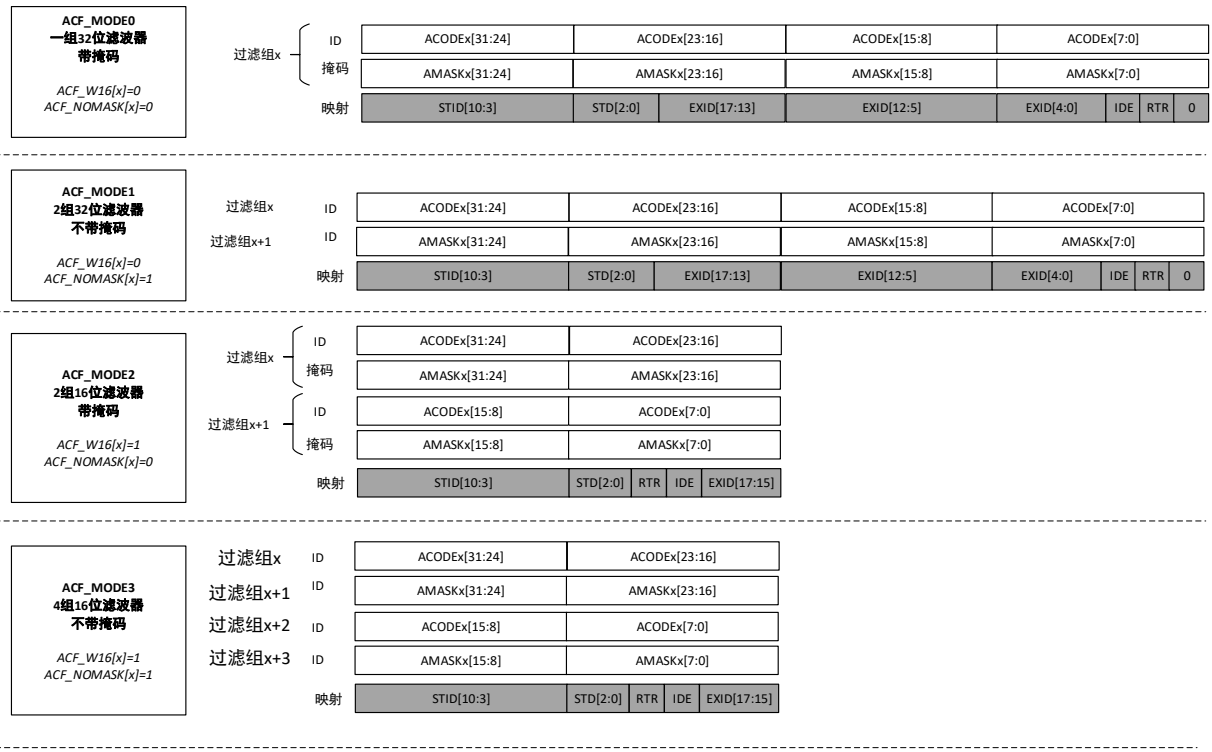
- code 方式：在 code 方式下，过滤器组中的 ACODE 寄存器和 AMASK 寄存器，都用于存放指定的 ID 数据，当且仅当接收到的帧 ID 与 ACODE/AMASK 中的数据完全一致才会杯存放在 RXFIFO 中。
- mask 方式：在 mask 方式下，过滤器组中的 ACODE 寄存器用来存放指定的 ID 数据，AMASK 寄存器用来存放屏蔽位数据，值为’0’表示对应 ACODE 中的对应 bit 位。当且仅当接收的 ID 数据与 AMASK 相与的结果等于 ACODE 与 AMASK 相与的结果，才会将该 ID 帧数据存放到 RXFIFO 中

过滤器长度说明：

- 半字模式(HalfWord)：该模式下，32 位的过滤器组会当成两个过滤器组使用，其中高 16 位位一组，低 16 位为一组
- 字模式(Word)：该模式下，32 为的过滤去组会当成一个过滤器组使用

根据上述两种配置的不同组合，过滤器组共有 4 种工作方式。对应的过滤器结构如下图所示：

图 3-1 FDCAN 过滤器结构示意图



从上图可以看出，H5 FDCAN 的过滤器有四种类型：

- 一个 32 位 code 带 mask 的过滤器类型
- 两个 32 位 code 不带 mask 的过滤器类型
- 两个 16 位 code 带 mask 的过滤器类型
- 四个 16 位 code 不带 mask 的过滤器类型

这 16 个过滤器组都可以独立的配置成以上四种类型中的其中一个。

3.3. 收发缓存

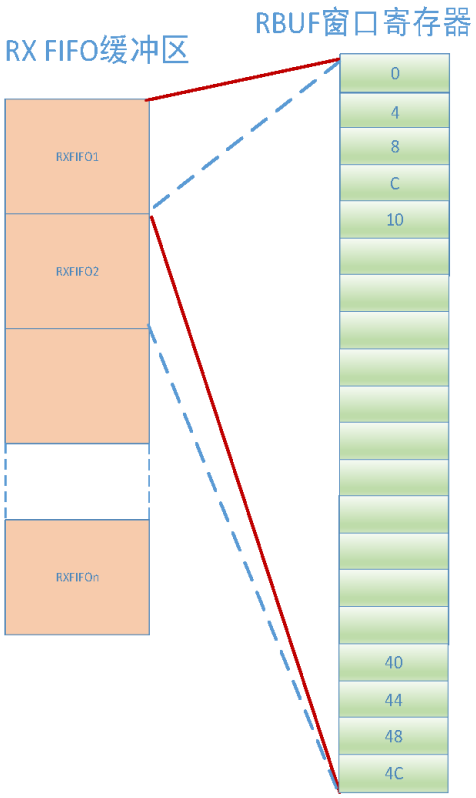
ACM32H5xx 系列芯片的 FDCAN 模块内置可存储 16 帧数据的 RXFIFO，用以保证在 CPU 无法及时响应时不会丢失接收数据。可通过 IR 寄存器中的相关标志位查看 RXFIFO 的状态。

发送缓存 TXFIFO 有高优先级 PTB（primary transmit buffer）和低优先级的 STB（secondary transmit buffer）。PTB 可以负责处理需要紧急发送的 CAN 帧，而 STB 缓存里的帧也可以配置成按 FIFO 方式或按 ID 优先级的方式进行发送，以有效应对实际需求。

3.3.1. 接收缓存 RXFIFO

RXFIFO 缓冲区共有 16 个槽，因此可以接收存贮最多 16 个 CAN-FD 帧。可以把这个 RXFIFO 缓冲区想象成一个 RBUF 数据结构类型的大小为 16 的数组。如下图左边所示。

图 3-2 接收缓存结构示意图



Address	Bit position								Function
	7	6	5	4	3	2	1	0	
RBUF	ID(7:0)								Identifier
RBUF+1	-						ID(10:8)		Identifier
RBUF+2	-								Identifier
RBUF+3	ESI	-							Identifier
RBUF+4	IDE=0	RTR	FDF	BRS	DLC(3:0)				Control
RBUF+5	KOER			TX	-				Status
RBUF+6	CYCLE_TIME(7:0)								TTCAN
RBUF+7	CYCLE_TIME(15:8)								TTCAN
RBUF+8	d1(7:0)								Data byte 1
RBUF+9	d2(7:0)								Data byte 2
.	...								.
RBUF+71	d64(7:0)								Data byte 64
RBUF+72	RTS(7:0)								CiA 603
.	...								.
RBUF+79	RTS(63:56)								CiA 603

RBUF 数据结构

中间的 RBUF 寄存器窗口（共 80 字节空间）为软件获取 CAN-FD 帧信息的寄存器接口，当前红色表示该窗口指向 RXFIFO1，当读取该 RXFIFO 中的数据后，通过将 FDCAN_CR 寄存器中的 RREL 位置 1，RBUF 寄存器窗口将指向下一个 RXFIFO2，如图中蓝色虚线所示，同时 RBUF 窗口寄存器里的数据也更新成 RXFIFO2 里的内容。

右边即为 RBUF 里的相关数据的定义。

3.3.2. 发送缓存 TXFIFO

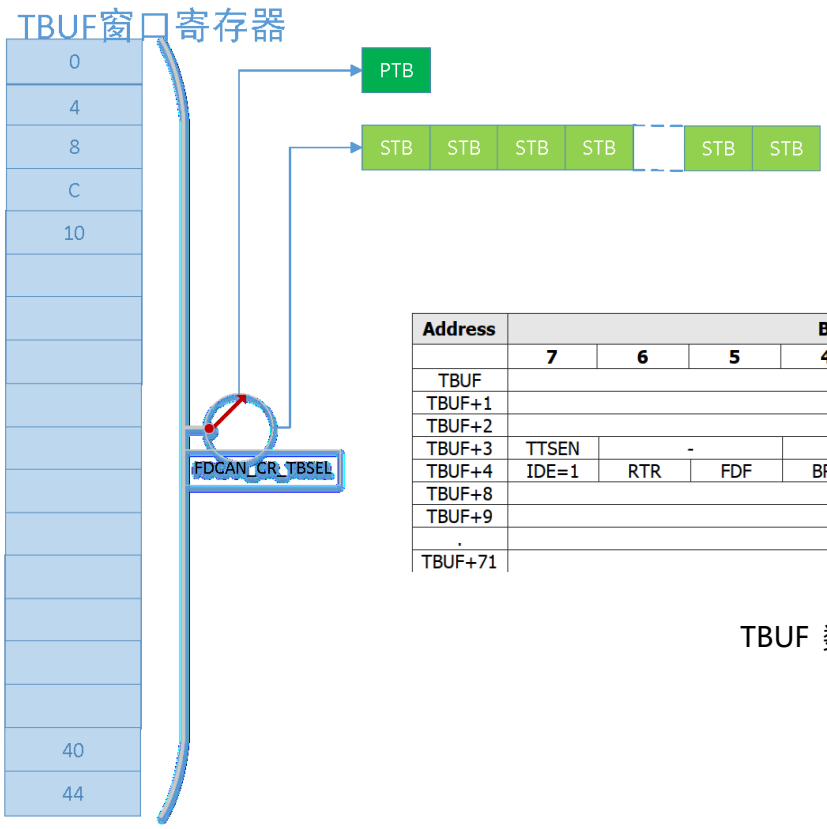
发送缓存 TXFIFO 有两种：

1 个 PTB，高优先级

16 个 STB，低优先级。16 个 STB 也可配置成 FIFO 方式（默认）或者以帧 ID 为优先级的方式进行发送

同样，也可以把 STB 看成是一个 TBUF 数据结构类型的大小为 16 的数组。

图 3-3 发送缓存结构示意图



Address	Bit position								Function
	7	6	5	4	3	2	1	0	
TBUF	ID(7:0)								Identifier
TBUF+1	ID(15:8)								Identifier
TBUF+2	ID(23:16)								Identifier
TBUF+3	TTSEN	-		ID(28:24)					Identifier
TBUF+4	IDE=1	RTR	FDF	BRS	DLC(3:0)				Control
TBUF+8	d1(7:0)								Data byte 1
TBUF+9	d2(7:0)								Data byte 2
.	...								.
TBUF+71	d64(7:0)								Data byte 64

TBUF 数据结构

如上图所示左边为 TBUF 窗口寄存器（共 72 个字节空间），寄存器里的每个字节的定义如右边所示。通过 FDCAN_CR 寄存器的 TBSEL 位，可以让 TBUF 选择 PTB 还是 STB。因为 STB 有 16 个，所以可以存放共 16 个要发送的 CAN-FD 帧

3.4. 测试模式

3.4.1. 监听模式（LOM）

监听模式用于监测 CAN 总线上的其他两个或多个 CAN 节点之间的数据收发，而不会去干预 CAN 总线行为。处于 LOM 模式的 CAN 节点不会回复 ACK，也不会发送主动错误帧。置位 FDCAN_CR.LOM 位将使能监听模式。

3.4.2. 回环模式（LBM）

回环模式有两种，一种是内部回环模式（LBMI），另一种是外部回环模式（LBME）。置位 FDCAN_CR.LBMI 或 FDCAN_CR.LBME 将使能 LBMI 模式或 LBME 模式。LBME 模式也可配合 FDCAN_CR.SACK 来使用。

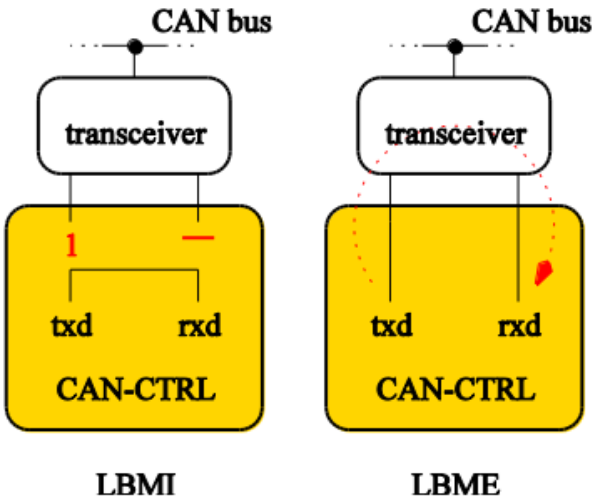
LBMI 模式下的节点通过模块内部将 CAN_TX 短接到 CAN_RX，同时自己会产生 ACK（self-ACK）。因此可以接收到自己发送的帧。

LBME 模式下的节点通过收发器的帮助在返回给自己。LBME 可以组合 SACK（self ACK）一起使用。

当 LBME=1，SACK=0 时，如果总线上有其他节点响应 ACK，则 LBME 的节点才可以接收到自己发送出去的帧，否则将产生 ACK-error。

当 LBME=1，SACK=1 时，LBME 的节点将自己产生 ACK 响应，因此可以无需总线上其他节点的参与

图 3-4 回环模式示意图


CAN bus
transceiver
1
txd
rxd
CAN-CTRL
LBMI
CAN bus
transceiver
txd
rxd
CAN-CTRL
LBME

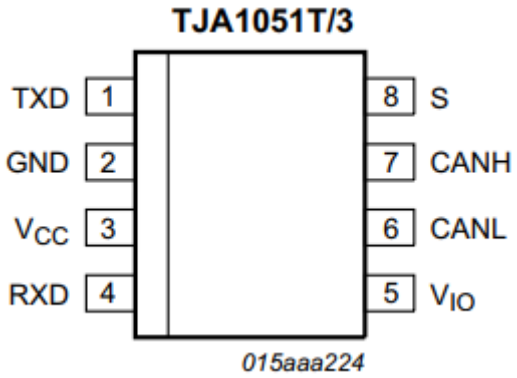
4. 应用使用说明

本章节内容将介绍 FDCAN 的使用方法。

4.1. 搭建硬件环境

在进行 FDCAN 的通信之前，需要先搭建好硬件电路环境。在一个 CAN 总线系统中，可以有多个 CAN 节点，但至少需要两个 CAN 节点才能实现通信。H5 的 FDCAN 模块是一个 CAN-FD 协议处理模块，需要通过支持 CAN-FD 的收发器挂载到 CAN 总线上才可以进行 CAN-FD 的通信。收发器处理物理层通信。

图 4-1 TJA1051T/3 CAN-FD 收发器



上图所示为 TJA1051T/3 CAN-FD 收发器，其中 TXD/RXD 管脚需要分别连接到 H5 FDCAN 模块所使用的 CAN_TX 和 CAN_RX 管脚。如果两个 CAN 节点的距离较远，需要通过屏蔽双绞线缆将收发器的 CANH 和 CANL 相连。

4.2. FDCAN 的初始化

因为 FDCAN 对时钟精度要求较高，在使用 FDCAN 进行通信时，建议系统时钟使用外部晶体 PLL 时钟源的方式，避免内部 RC 精度不够而导致的通信问题。在使用 FDCAN 模块进行通信时，我们需要先将其初始化好并工作起来。

4.2.1. 初始化配置

```
fdcan_handler.Instance = FDCAN1;
fdcan_handler.Init.Mode = FDCAN_MODE_NORMAL;
fdcan_handler.Init.FrameISOType = FDCAN_FRAME_ISO;
fdcan_handler.Init.RxBufOverFlowMode = FDCAN_RX_BUF_BLOCKING;
fdcan_handler.Init.NominalPrescaler     = 10;
fdcan_handler.Init.NominalSyncJumpWidth = 4;
fdcan_handler.Init.NominalTimeSeg1      = 24;
fdcan_handler.Init.NominalTimeSeg2      = 8;
fdcan_handler.Init.DataPrescaler        = 1;
fdcan_handler.Init.DataSyncJumpWidth    = 5;
fdcan_handler.Init.DataTimeSeg1         = 30;
fdcan_handler.Init.DataTimeSeg2         = 10;
HAL_FDCAN_Init(&fdcan_handler);
```

FDCAN 的初始化通过调用函数 HAL_FDCAN_Init 来完成。FDCAN 初始化包括以下的参数配置：

- FDCAN 实体及对应的管脚

  使用的 FDCAN 实体由 fdcan_handler.Instance 来定义，而管脚的选择则在 HAL_FDCAN_Init 函数里的回调函数 HAL_FDCAN_MspInit 里实现（详见 acm32h5xx_hal_msp.c），其中也会打开 FDCAN 模块的时钟

以及所使用的 GPIO 模块的时钟。

- 工作模式

  工作模块包含正常模式，监听模式及测试模式，由 fdcan_handler.Init.Mode 定义，详见 hal_fdcan.h 中 FDCAN_operating_mode

- 仲裁段波特率

  仲裁段波特率最高 1M，兼容 CAN2.0。由 fdcan_handler.Init.NominalXXX 字段来定义

- 数据段波特率

  数据段波特率一般根据使用的 CAN 收发器，最高 8M。由 fdcan_handler.Init.DataXXX 字段来定义

- FrameISOType 以及 RxBufOverFlowMode

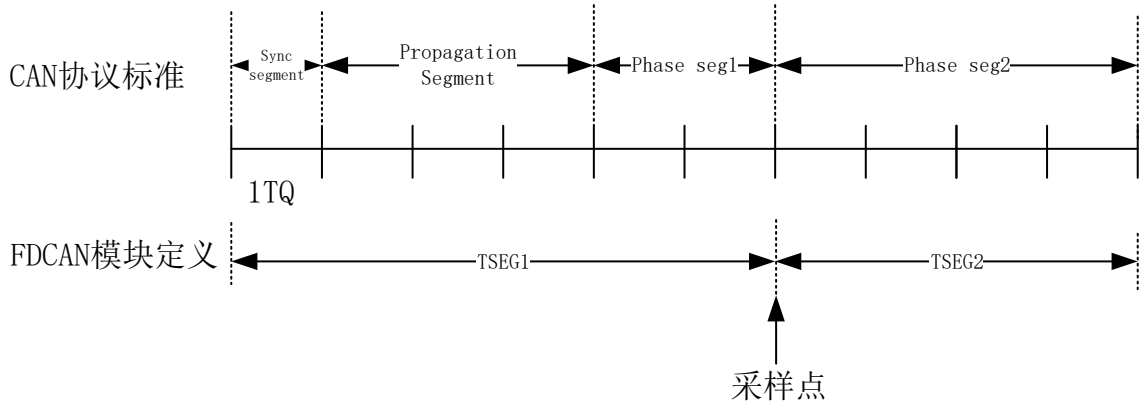
  由于 ISO 类型的帧结构和 Non-ISO 类型的帧结构不兼容，因此需要由用户指定选择某个类型的帧。由 fdcan_handler.Init.FrameISOType 来指定。fdcan_handler.Init.RxBufOverFlowMode 用于设置当接收缓存满了之后，后续接收到的帧是直接丢弃还是覆盖缓存中的帧。

这些初始化信息的配置，只有波特率的设置可能会让你觉得困扰。以下将详细介绍波特率的配置。

4.2.2. 波特率配置说明

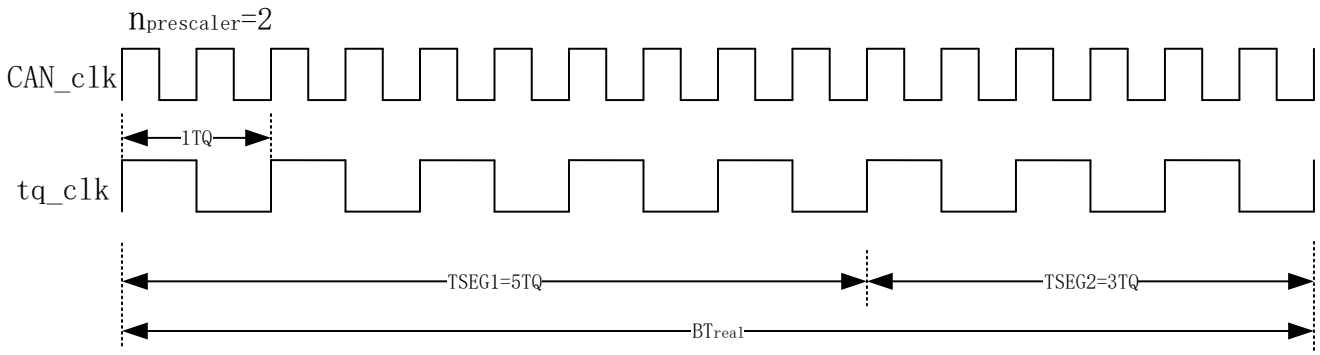
CAN 协议规定一个 bit 的时间 BT 由几段组成如上图所示，每段都包含 1 个或多个 TQ 组成。

图 4-2 CAN BIT 时间定义



1 个 TQ 的时间：$1TQ = n_{prescaler}/f_{HCLK}$

图 4-3 Bit 采样的时钟分频示意图



实际的 BT 时间为 CAN 总线波特率 BR 的倒数，即 $BT_{real} = 1/BR$。那么 n_{TQ} 和 $n_{prescaler}$ 的选择就要满足使得 BT_{idle} 尽量和 BT_{real} 接近最好是等于 BT_{real}。即：

$BT_{idle} \approx BT_{real} = n_{prescaler} * n_{TQ} / f_{clock} = TSEG1 + TSEG2$

仲裁段波特率和数据段波特率设置方法一致。现以设置数据段的波特率为 2M，且 HCLK 时钟频率为 80MHz，

进行举例说明：

- 为了尽量满足 BT_{idle} 和 BT_{real} 相等，选择 $n_{prescaler}$ 等于 5，n_{TQ} 等于 16，这样使得 $BT_{idle} = BT_{real} = 16$ TQ

  即 NominalPrescaler=5.

- BTreal = TSEG1 + TSEG2。TSEG1 需要略大于 TSEG2。因此 TSET1 = 10（TQ），TSEG2 = 6（TQ）是一个合适的选择。

- 设置 DataSyncJumpWidth，DataSyncJumpWidth 要满足不能大于 TSEG2，建议配置成 TSEG2 的一半。因此设置 DataSyncJumpWidth = 3（TQ）。

4.2.3. 关于采样点的设置

采样点的设置在 FDCAN 通信中起到至关重要的作用。由于 FDCAN 的数据段的速率切换是在 BRS 字段开始的，因此在一个 CAN 总线系统中，每个 FDCAN 节点的采样点位置都需要配置成一致。

采样点设置的准则是让 TSEG1 略大于 TSEG2。参考图 4-2 CAN BIT 时间定义里的采样点位置。

4.3. 过滤器的配置

4.3.1. 数据结构

通过 3.2 节过滤器的实现可以知道 H5 的 FDCAN 中有四种过滤器类型。因此代码中定义了相对应的四种过滤器的数据类型：

4.3.1.1. 四种 FDCAN 过滤器枚举类型

```
typedef enum
{
    FDCAN_FILTER_TYPE0 = 0,
    FDCAN_FILTER_TYPE1,
    FDCAN_FILTER_TYPE2,
    FDCAN_FILTER_TYPE3,
}FDCAN_Filter_enum;
```

4.3.1.2. 32 位过滤器的结构体位域定义

```
typedef union
{
    struct
    {
        uint32_t r0:1;
        uint32_t RTR:1;
        uint32_t IDE:1;
        uint32_t EXTID0_17:18;
        uint32_t id:11;
    }b;

    struct
    {
        uint32_t r0:1;
        uint32_t RTR:1;
        uint32_t IDE:1;
        uint32_t id:29;
```

```
    }ext;

    uint32_t w;
}FDCAN_Filter32_Map_u;
```

4.3.1.3. 16 位过滤器的结构体位域定义

```
typedef union
{
    struct
    {
        uint16_t EXTID15_17:3;
        uint16_t IDE:1;
        uint16_t RTR:1;
        uint16_t id:11;
    }b;
    uint16_t hw;
}FDCAN_Filter16_Map_u;
```

4.3.1.4. 四种过滤器类型结构体的联合体

```
typedef union
{
    struct
    {
        FDCAN_Filter32_Map_u code32;    //acode;
        FDCAN_Filter32_Map_u mask32;    //amask;
    }type0;

    struct
    {
        FDCAN_Filter32_Map_u code32;
        FDCAN_Filter32_Map_u code32_2;
    }type1;

    struct
    {
        FDCAN_Filter16_Map_u code16;
        FDCAN_Filter16_Map_u mask16;
        FDCAN_Filter16_Map_u code16_2;
        FDCAN_Filter16_Map_u mask16_2;
    }type2;

    struct
    {
        FDCAN_Filter16_Map_u code16;
        FDCAN_Filter16_Map_u code16_2;
        FDCAN_Filter16_Map_u code16_3;
        FDCAN_Filter16_Map_u code16_4;
    }type3;

}FDCAN_Filter_u;
```

- type0 对应一个 32 位带 mask 的过滤器类型
- type1 对应两个 32 位不带 mask 的过滤器类型
- type2 对应两个 16 位带 mask 的过滤器类型
- type3 对应四个 16 位不带 mask 的过滤器类型

4.3.1.5. FDCAN 过滤器结构体定义

```
typedef struct
{
    uint32_t           FilterIndex;    /* 0~15 */
    FDCAN_Filter_enum   FilterType;    /* 参考 FDCAN_Filter_enum */
    FDCAN_Filter_u      Filter;        /* 该 FilterType 类型下的过滤器 Code 和 Mask 的配置 */

}FDCAN_FilterTypeDef;
```

其中 FilterIndex 表示选择哪个过滤器

FilterType 表示使用哪个过滤器类型，取值参考枚举类型数据 FDCAN_Filter_enum。

Filter 表示该过滤器类型下的过滤器位域的结构体定义

4.3.2. 过滤器类型选择

对于四种过滤器类型，该如何选择对应的过滤器类型，下面是一个常用的选择方法：

- 当需要过滤的 ID 是 29 位，且只需要匹配这 29 位中某些 bit 位时，可以选择 type0 过滤器类型，即一个 32 位 code 和 32 位 mask 的过滤器；
- 当需要过滤的 ID 是 29 位，并且需要完全匹配该 ID 时，可以选择 type1 过滤器类型，即两个 32 位 code 不带 mask 的过滤器；
- 当需要过滤的 ID 是 11 位，且只需要匹配这 11 位中某些 bit 位时，可以选择 type2 过滤器类型，即两个 16 位 code 和 16 位 mask 的过滤器；
- 当需要过滤的 ID 是 11 位，并且需要完全匹配该 ID 时，可以选择 type3 过滤器类型，即四个 16 位 cdoe 不带 mask 的过滤器。

4.3.3. 配置流程

过滤器的配置函数为：

```
HAL_FDCAN_ConfigFilter(FDCAN_HandleTypeDef *hfdcan, FDCAN_FilterTypeDef *sFilterConfig)
```

该函数需要在 HAL_FDCAN_Init 函数之后进行调用。在调用 HAL_FDCAN_ConfigFilter 函数之前，需要设置过滤器参数信息 sFilterConfig。下面以配置一个 type0 类型的过滤器，只接收 0x18FExxxx 的扩展数据帧或扩展远程帧为例进行说明。

code32 里为要配置的帧 ID 相关的 code 字。

mask32 里为配置的屏蔽字，mask32 里的某个 bit 位为 0 表示必须匹配 code32 里对应的 bit 位，为 1 表示 code32 里对应的 bit 位可以不匹配。

需要注意的是，当 FilterType 选择了 FDCAN_FILTER_TYPE0 后，Filter 里的联合体类型也要是相应的 type0。

```
/* config FDCAN filters */
FDCAN_FilterTypeDef FilterConfig;                                   /* 定义 FilterConfig 变量 */

FilterConfig.FilterIndex = 0;                                       /* 选择过滤器组 0 */
```

```
FilterConfig.FilterType = FDCAN_FILTER_TYPE0;                    /* 选择过滤器类型 type0 */
FilterConfig.Filter.type0.code32.ext.IDE = FDCAN_EXTENDED_ID;    /* 过滤的 ID 为扩展 ID */
FilterConfig.Filter.type0.code32.ext.id = 0x18FE0000;            /* 过滤的 ID 号 */
FilterConfig.Filter.type0.code32.ext.RTR = 0;                    /* 数据帧 */

FilterConfig.Filter.type0.mask32.ext.IDE = 0;           /* mask 里的 IDE 为 0 表示只接收扩展帧 */
FilterConfig.Filter.type0.mask32.ext.id = 0x0000FFFF;   /*0x0000FFFF 表示必须高 16 位匹配 0x18FE
                                                          而低 16 位则不 care，表示可以接收 0x18FExxxx 的帧*/
FilterConfig.Filter.type0.mask32.ext.RTR = 1;           /* mask 里的 RTR 为 1，表示接收数据帧或远程帧 */
HAL_FDCAN_ConfigFilter(&fdcan_handler, &FilterConfig);
```

4.3.4. 注意事项

在这个 16 个过滤器组中，过滤器组 0 是默认开启的，并且默认是接收所有 CAN-FD 数据帧的。所以当你在配置过滤器时，选择了其他过滤器组，并且保留过滤器组 0 的默认值（Reset 复位后）配置的话，那么你会发现你所配置的过滤器根本不起作用。这是因为过滤器组 0 接收所有帧的原因。因此当一个 ID 被一个过滤器拒绝，但是却被另一个过滤器接收的话，那么仍然是可以接收到该 ID 的帧的。注意如果所有过滤器都不使能，即 FDCAN_ACFCR.AE = 0，那么将无法接收任何帧。

4.4. 发送数据帧

当 FDCAN 模块初始化好之后，可以通过调用函数

```
HAL_FDCAN_TransmitMessageBySTB(FDCAN_HandleTypeDef *hfdcan, FDCAN_TxHeaderTypeDef *pTxHeader, uint8_t *pTxData)
```

或

```
HAL_FDCAN_TransmitMessageByPTB(FDCAN_HandleTypeDef *hfdcan, FDCAN_TxHeaderTypeDef *pTxHeader, uint8_t *pTxData)
```

来进行一个 CAN-FD 帧的发送。在进行调用之前，需要先准备好相关的包头信息以及实际要发送的数据：

```
uint8_t TxData0[] = {0x10, 0x32, 0x54, 0x76, 0x98, 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66};
FDCAN_TxHeaderTypeDef TxHeader;
TxHeader.ID.b.ID = 0x111;                                   /* 帧 ID  */
TxHeader.ID.b.TTSEN = 0;                                    /* 是否使能时间戳，0 表示不使能  */
TxHeader.FrameInfo.b.BRS = 1;                               /* 是否启动数据段波特率加速，1 表示使能加速
                                                               如果 FDF=0，那么 BRS 将被强制置为 0 */
TxHeader.FrameInfo.b.DLC = __HAL_FDCAN_LEN2DLC( 12 );       /* DLC 编码的数据长度，详见 DLC 编码 */
TxHeader.FrameInfo.b.FDF = 1;                               /* CAN2.0 帧还是 CAN-FD 帧，1 表示 CAN-FD 帧 */
TxHeader.FrameInfo.b.IDE = 0;                               /* 是否是扩展帧，0 表示标准帧
TxHeader.FrameInfo.b.RTR = 0;                               /* 是否是远程帧，0 表示数据帧。如果 FDF=1，
                                                               那么 RTR 将被强制置为 0 */
HAL_FDCAN_TransmitMessageBySTB(&fdcan_handler, &TxHeader, TxData0);
```

4.5. 接收数据帧

CAN-FD 帧的接收用到以下数据结构：

```
typedef union
{
    struct{
        __IO uint32_t ID:29;            /* 接收到的帧 ID */
```

```
        __IO uint32_t RSV:2;
        __IO uint32_t ESI:1;           /* error state indicator */
    }b;
    __IO uint32_t w;
}FDCAN_RX_ID_u;
```

```
typedef union
{
    struct{
        __IO uint32_t DLC:4;           /* 接收到的帧长度，注意是 DLC 编码 */
        __IO uint32_t BRS:1;           /* 是否是 CAN-FD 数据段波特率加速， 1 表示加速 */
        __IO uint32_t FDF:1;           /* 是否是 CAN-FD 帧，1 表示 CAN-FD 帧 */
        __IO uint32_t RTR:1;           /* 是否是远程帧，1 表示远程帧 */
        __IO uint32_t IDE:1;           /* 是否是扩展帧，1 表示扩展帧 */
        __IO uint32_t RSV0:4;
        __IO uint32_t TX:1;            /* TX=1 表示是回环模式接收到自己发送的帧，为 0 表示其他节点发送的帧 */
        __IO uint32_t KOER:3;          /* 错误类型 */
        __IO uint32_t CycleTime:16;    /* 接收到帧时的时间戳 */
    }b;
    __IO uint32_t w;
}FDCAN_RX_FrameInfo_u;
```

数据帧的接收通过调用函数：

```
HAL_FDCAN_GetRxMessage(FDCAN_HandleTypeDef *hfdcan, FDCAN_RxHeaderTypeDef *pRxHeader, uint8_t *pRxData)
```

例如：当 FDCAN_IR 里的 RIF 标志置 1 时（需打开 RIE 中段使能），或 FDCAN_CR 里的 RSTAT 位域不为 0 时，调用 HAL_FDCAN_GetRxMessage 进行帧的获取。

```
FDCAN_RxHeaderTypeDef RxHeader;
uint8_t RxData[64];
if(HAL_FDCAN_GetRxBufFillState(&fdcan_handler) != FDCAN_RXBUF_EMPTY)
{ HAL_FDCAN_GetRxMessage(&fdcan_handler, &RxHeader, RxData);}
```

5. 版本历史

版本	日期	作者	描述
V0.1	2024-11-26	陆建钢	First release
V0.2	2024-12-11	陆建钢	审阅后更新

联系我们

公司：上海航芯电子科技股份有限公司

地址：上海市闵行区合川路 2570 号科技绿洲三期 2 号楼 702 室

邮编：200241

电话：+86-21-6125 9080

传真：+86-21-6125 9080-830

Email： service@aisinochip.com

Website： www.aisinochip.com