网易 博客 发现 小组 风格 网易真人搭配社区iStyle 推荐 引用 加关注 使用此风格 写日志 个人中心 消息(0)

STM32 CAN通信 滤波器配置总结

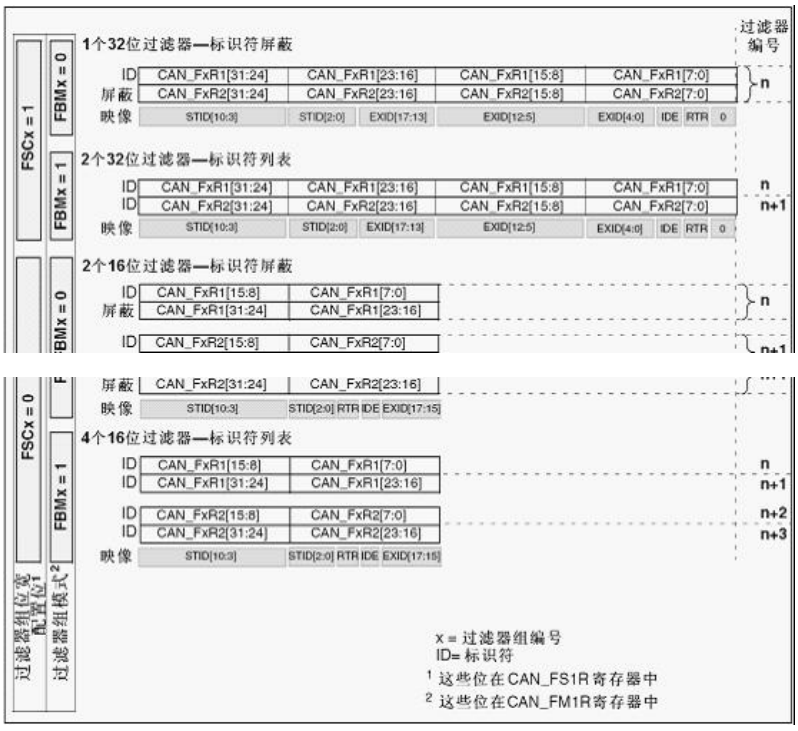
2012-09-20 16:28:34 | 分类： 默认分类 订阅 | 字号

首先声明stm32手册写的太蛋疼，让我看了好长时间没看懂，后来根据实践再回头看了，才看懂一些，在这里还要感激网友发表的博客，下面内容为转载：

一、在STM32互联型产品中，CAN1和CAN2分享28个过滤器组,其它STM32F103xx系列产品中有14个过滤器组,用以对接收到的帧进行过滤。

每组过滤器包括了2个可配置的32位寄存器：CAN_FxR0和CAN_FxR1。这些过滤器相当于关卡，每当收到一条报文时，CAN要先将收到的报文从这些过滤器上"过"一下，能通过的报文是有效报文，收进相关联FIFO（FIFO1或FIFO2），不能通过的是无效报文(不是发给"我"的报文)，直接丢弃。

（标准CAN的标志长度是11位。扩展格式CAN的标志长度是29。CAN2.0A协议规定CAN控制器必须有一个11位的标识符。CAN2.0B协议中规定CAN控制器的标示符长度可以是11位或29位。STM32同时支持CAN2.0A/CAN2.0B协议。）



每组过滤器组有两种工作模式：标识符列表模式和标识符屏蔽位模式。

标识符屏蔽位模式：可过滤出一组标识符。此时，这样CAN_FxR0中保存的就是标识符匹配值，CAN_FxR1中保存的是屏蔽码，即CAN_FxR1中如果某一位为1，则CAN_FxR0中相应的位必须与收到的帧的标志符中的相应位吻合才能通过过滤器；CAN_FxR1中为0的位表示CAN_FxR0中的相应位可不必与收到的帧进行匹配。

标识符列表模式：可过滤出一个标识。此时CAN_FxR0和CAN_FxR1中的都是要匹配的标识符，收到的帧的标识符必须与其中的一个吻合才能通过过滤。

注意：CAN_FilterIdHigh是指高16位CAN_FilterIdLow是低16位应该将需要得到的帧的和过滤器的设置值左对齐起。

所有的过滤器是并联的，即一个报文只要通过了一个过滤器，就是算是有效的。

按工作模式和宽度，一个过滤器组可以变成以下几中形式之一：

(1) 1个32位的屏蔽位模式的过滤器。

(2) 2个32位的列表模式的过滤器。

likang01

加博友 关注他

他的网易微博

最新日志

ARM编译中的RO、RW和..
C语言高效编程的几招.
硬件问题坑死人
STM32 CAN通信 滤波器...
单片机串口处理获得的经..
stm32f105/107互联性时..

随机阅读

男人接受不了什么样的女..
1979北京机场vs.2012爱..
中国特色的罚款经济学
佛家的用人故事
人脉管理的福音书
魏震：8月13-17日黄金外..

首页推荐

十八大上的台湾代表(图)
北京猿人或属于食人部落
江泽民18大前露面的用意
已婚男勾未婚女常用伎俩
公务员热迎来最后的疯狂
四张罕见的毛泽东旧照片

更多>>

(3) 2个16位的屏蔽位模式的过滤器。
(4) 4个16位的列表模式的过滤器。

每组过滤器组有两个32位的寄存器用于存储过滤用的"标准值"，分别是FxR1，FxR2。
在32位的屏蔽位模式下：
有1个过滤器。
FxR2用于指定需要关心哪些位，FxR1用于指定这些位的标准值。
在32位的列表模式下：
有两个过滤器。
FxR1指定过滤器0的标准值FxR2指定过滤器1的标准值。

收到报文的标识符只有跟FxR1与FxR1其中的一个完全相同时，才算通过。
在16位的屏蔽位模式下：
有2个过滤器。
FxR1配置过滤器0，其中，[31-16]位指定要关心的位，[15-0]位指定这些位的标准值。
FxR2配置过滤器1，其中，[31-16]位指定要关心的位，[15-0]位指定这些位的标准值。
在16位的列表模式下：
有4个过滤器。
FxR1的[15-0]位配置过滤器0，FxR1的[31-16]位配置过滤器1。
FxR2的[15-0]位配置过滤器2，FxR2的[31-16]位配置过滤器3。

STM32的CAN有两个FIFO，分别是FIFO0和FIFO1。为了便于区分，下面FIFO0写作FIFO_0，FIFO1写作FIFO_1。
每组过滤器组必须关联且只能关联一个FIFO。复位默认都关联到FIFO_0。
所谓“关联”是指假如收到的报文从某个过滤器通过了，那么该报文会被存到该过滤器相连的FIFO。
从另一方面来说，每个FIFO都关联了一串的过滤器组，两个FIFO刚好瓜分了所有的过滤器组。

每当收到一个报文，CAN就将这个报文先与FIFO_0关联的过滤器比较，如果被匹配，就将此报文放入FIFO_0中。
如果不匹配，再将报文与FIFO_1关联的过滤器比较，如果被匹配，该报文就放入FIFO_1中。
如果还是不匹配，此报文就被丢弃。

每个FIFO的所有过滤器都是并联的，只要通过了其中任何一个过滤器,该报文就有效。
如果一个报文既符合FIFO_0的规定，又符合FIFO_1的规定，显然，根据操作顺序，它只会放到FIFO_0中。

每个FIFO中只有激活了的过滤器才起作用，换句话说，如果一个FIFO有20个过滤器，但是只激活了5个，那么比较报文时，只拿这5个过滤器作比较。
一般要用到某个过滤器时，在初始化阶段就直接将它激活。
需要注意的是，每个FIFO必须至少激活一个过滤器，它才有可能收到报文。如果一个过滤器都没有激活，那么是所有报文都报废的。
一般的，如果不想用复杂的过滤功能，FIFO可以只激活一组过滤器组，且将它设置成32位的屏蔽位模式，两个标准值寄存器(FxR1，FxR2)都设置成0。这样所有报文均能通过。（STM32提供的例程里就是这么做的！）

STM32 CAN中，另一个较难理解的就是过滤器编号。
过滤器编号用于加速CPU对收到报文的处理。
收到一个有效报文时， CAN会将收到的报文 以及它所通过的过滤器编号， 一起存入接收邮箱中。CPU在处理时，可以根据过滤器编号，快速的知道该报文的用途，从而作出相应处理。
不用过滤器编号其实也是可以的， 这时候CPU就要分析所收报文的标识符， 从而知道报文的用途。
由于标识符所含的信息较多，处理起来就慢一点了。

STM32使用以下规则对过滤器编号：
(1) FIFO_0和FIFO_1的过滤器分别独立编号，均从0开始按顺序编号。
(2) 所有关联同一个FIFO的过滤器，不管有没有被激活，均统一进行编号。
(3) 编号从0开始，按过滤器组的编号从小到大，按顺序排列。
(4) 在同一过滤器组内，按寄存器从小到大编号。FxR1配置的过滤器编号小，FxR2配置的过滤器编号大。
(5) 同一个寄存器内，按位序从小到大编号。[15-0]位配置的过滤器编号小，[31-16]位配置的过滤器编号大。
(6) 过滤器编号是弹性的。 当更改了设置时，每个过滤器的编号都会改变。
但是在设置不变的情况下，各个过滤器的编号是相对稳定的。

这样，每个过滤器在自己在FIFO中都有编号。
在FIFO_0中，编号从0 -- (M-1)， 其中M为它的过滤器总数。
在FIFO_1中，编号从0 -- (N-1),，其中N为它的过滤器总数。

一个FIFO如果有很多的过滤器,，可能会有一条报文， 在几个过滤器上均能通过，这时候,，这条报文算是从哪儿过来的呢？
STM32在使用过滤器时，按以下顺序进行过滤：

(1) 位宽为32位的过滤器，优先级高于位宽为16位的过滤器。

(2) 对于位宽相同的过滤器，标识符列表模式的优先级高于屏蔽位模式。

(3) 位宽和模式都相同的过滤器，优先级由过滤器号决定，过滤器号小的优先级高。

按这样的顺序，报文能通过的第一个过滤器，就是该报文的过滤器编号，被存入接收邮箱中。

二、下面是我的代码：

```
/*时钟初始化*/
void RCC_Configuration(void)
{
    ErrorStatus HSEStartUpStatus;
 // RCC system reset(for debug purpose)
    RCC_DeInit();

    // Enable HSE
    RCC_HSEConfig(RCC_HSE_ON);

 //Enable HSI for Flash Operation
 RCC_HSICmd(ENABLE);

    // Wait till HSE is ready
    HSEStartUpStatus = RCC_WaitForHSEStartUp();

    if(HSEStartUpStatus == SUCCESS)
    {
        // HCLK = SYSCLK     AHB时钟为系统时钟  72MHz
        RCC_HCLKConfig(RCC_SYSCLK_Div1);

        // PCLK2 = HCLK     APB2时钟为系统时钟 72MHz
        RCC_PCLK2Config(RCC_HCLK_Div1);

        // PCLK1 = HCLK/2     APB1时钟为系统时钟 72MHz/2=36MHz
        RCC_PCLK1Config(RCC_HCLK_Div2);

        // Flash 2 wait state
        FLASH_SetLatency(FLASH_Latency_2);
        // Enable Prefetch Buffer
        FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);

 // PLLCLK = 8MHz * 9 = 72 MHz
        RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9);

        // Enable PLL
        RCC_PLLCmd(ENABLE);

        // Wait till PLL is ready
        while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET)
        {
        }

        // Select PLL as system clock source
        RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);

        // Wait till PLL is used as system clock source
        while(RCC_GetSYSCLKSource() != 0x08)
        {
        }
    }

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO |
                                          RCC_APB2Periph_GPIOA |
                                          RCC_APB2Periph_GPIOB |
                                          RCC_APB2Periph_GPIOC |
          RCC_APB2Periph_USART1 |
```

```
                                        RCC_APB2Periph_SPI1
                                        , ENABLE);


    RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG |
                                        RCC_APB1Periph_USART2 |
                                        RCC_APB1Periph_USART3 |
                                        RCC_APB1Periph_TIM3   |
          RCC_APB1Periph_TIM4   |
                                        RCC_APB1Periph_CAN1
//                                        RCC_APB1Periph_CAN2
                                        , ENABLE);
}
/*NVIC配置*/
void NVIC_Configuration(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;
    #ifdef  VECT_TAB_RAM
    // Set the Vector Table base location at 0x20000000
    NVIC_SetVectorTable(NVIC_VectTab_RAM, 0x0);
    #else  /* VECT_TAB_FLASH  */
    // Set the Vector Table base location at 0x08000000
    NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0);
    #endif
    // Configure one bit for preemption priority
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);

 NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
 NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
 NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 2;
 NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
 NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN1_RX0_IRQn;
 NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
 NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
 NVIC_Init(&NVIC_InitStructure);

}
/*管脚初始化*/
void CAN_PinInit(void)
{
 GPIO_InitTypeDef  GPIO_InitStructure;
     /* Configure CAN pin: RX */
 GPIO_InitStructure.GPIO_Pin = PIN_CAN_RX;
 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
 GPIO_Init(GPIO_CAN, &GPIO_InitStructure);

 /* Configure CAN pin: TX */
```

```
 GPIO_InitStructure.GPIO_Pin = PIN_CAN_TX;
 GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;//GPIO_Mode_Out_PP;
 GPIO_Init(GPIO_CAN, &GPIO_InitStructure);
}

/*CAN1配置函数*/

void CAN_Configuration(void)
{
 CAN_InitTypeDef        CAN_InitStructure;
    CAN_FilterInitTypeDef  CAN_FilterInitStructure;

    // CAN register init
    CAN_DeInit(CAN1);
    CAN_StructInit(&CAN_InitStructure);

    // CAN cell init
    CAN_InitStructure.CAN_TTCM=DISABLE;//禁止时间触发通信模式
    CAN_InitStructure.CAN_ABOM=DISABLE;
    CAN_InitStructure.CAN_AWUM=DISABLE;//睡眠模式通过清除sleep位来唤醒
    CAN_InitStructure.CAN_NART=ENABLE;//ENABLE;报文自动重传
    CAN_InitStructure.CAN_RFLM=DISABLE;//接收溢出时，FIFO未锁定
    CAN_InitStructure.CAN_TXFP=DISABLE;//发送的优先级由标示符的大小决定
    CAN_InitStructure.CAN_Mode=CAN_Mode_Normal;//正常模式下
    //设置can通讯波特率为50Kbps
    CAN_InitStructure.CAN_SJW=CAN_SJW_1tq;
    CAN_InitStructure.CAN_BS1=CAN_BS1_8tq;
    CAN_InitStructure.CAN_BS2=CAN_BS2_7tq;
    CAN_InitStructure.CAN_Prescaler=45;
    CAN_Init(CAN1,&CAN_InitStructure);

    // CAN filter init
    CAN_FilterInitStructure.CAN_FilterNumber=0;
    CAN_FilterInitStructure.CAN_FilterMode=CAN_FilterMode_IdMask;
    CAN_FilterInitStructure.CAN_FilterScale=CAN_FilterScale_32bit;//CAN_FilterScale_16bit; //32
bit

    CAN_FilterInitStructure.CAN_FilterIdHigh    = (((u32)slave_id<<21)&0xffff0000)>>16;
    CAN_FilterInitStructure.CAN_FilterIdLow    = (((u32)slave_id<<21)|CAN_ID_STD|CAN_RTR_DATA)&0
xffff;
    CAN_FilterInitStructure.CAN_FilterMaskIdHigh  = 0xFFFF;
    CAN_FilterInitStructure.CAN_FilterMaskIdLow    = 0xFFFF;

    CAN_FilterInitStructure.CAN_FilterFIFOAssignment=CAN_FIFO0;
    CAN_FilterInitStructure.CAN_FilterActivation=ENABLE;    //时能过滤器
    CAN_FilterInit(&CAN_FilterInitStructure);

    CAN_ITConfig(CAN1,CAN_IT_FMP0|CAN_IT_EPV, ENABLE);
}

/*CAN 发送函数*/

unsigned char CAN1_SendData(void)
{
    uint16 i;
 CanTxMsg TxMessage;
    unsigned char TransmitMailbox;

 TxMessage.StdId=0x11;       //标准标识符
 TxMessage.RTR=CAN_RTR_DATA; //数据帧
 TxMessage.IDE=CAN_ID_STD; //标准帧
 TxMessage.DLC=2;        //数据长度 2
 TxMessage.Data[0]=0xCA;      //发送的数据
 TxMessage.Data[1]=0xFE;
```

```
 TransmitMailbox=CAN_Transmit(CAN1,&TxMessage); //发送数据
    i = 0xFFF;
 do
{
    _NOP_(5);
 }
    while((CAN_TransmitStatus(CAN1,TransmitMailbox) != CANTXOK) && (--i));
    if(i <= 0x01)
        return 0;
    else
        return 1;
}
/*中断服务函数*/
void USB_LP_CAN1_RX0_IRQHandler(void)
{
 CanRxMsg RxMessage;
 CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);
}
```

三、滤波器配置详细如下：

1、对扩展数据帧进行过滤：(只接收扩展数据帧)

```
CAN_FilterInitStructure.CAN_FilterIdHigh    = (((u32)slave_id<<3)&0xFFFF0000)>>16;
CAN_FilterInitStructure.CAN_FilterIdLo=(((u32)slave_id<<3)|CAN_ID_EXT|CAN_RTR_DATA)&0xFFFF
;
CAN_FilterInitStructure.CAN_FilterMaskIdHigh  = 0xFFFF;
CAN_FilterInitStructure.CAN_FilterMaskIdLow   = 0xFFFF;
```

(注：标准帧数据帧、标准远程帧和扩展远程帧均被过滤)

2、对扩展远程帧过滤：(只接收扩展远程帧)

```
CAN_FilterInitStructure.CAN_FilterIdHigh    = (((u32)slave_id<<3)&0xFFFF0000)>>16;
CAN_FilterInitStructure.CAN_FilterIdLow   = (((u32)slave_id<<3)|CAN_ID_EXT|CAN_RTR_REMOTE)
&0xFFFF;
CAN_FilterInitStructure.CAN_FilterMaskIdHigh  = 0xFFFF;
CAN_FilterInitStructure.CAN_FilterMaskIdLow   = 0xFFFF;
```

3、对标准远程帧过滤：(只接收标准远程帧)

```
CAN_FilterInitStructure.CAN_FilterIdHigh    = (((u32)slave_id<<21)&0xffff0000)>>16;
CAN_FilterInitStructure.CAN_FilterIdLow   = (((u32)slave_id<<21)|CAN_ID_STD|CAN_RTR_REMOTE
)&0xffff;
CAN_FilterInitStructure.CAN_FilterMaskIdHigh  = 0xFFFF;
CAN_FilterInitStructure.CAN_FilterMaskIdLow   = 0xFFFF;
```

4、对标准数据帧过滤：(只接收标准数据帧)

```
CAN_FilterInitStructure.CAN_FilterIdHigh    = (((u32)slave_id<<21)&0xffff0000)>>16;
CAN_FilterInitStructure.CAN_FilterIdLow   = (((u32)slave_id<<21)|CAN_ID_STD|CAN_RTR_DATA)&
0xffff;
CAN_FilterInitStructure.CAN_FilterMaskIdHigh  = 0xFFFF;
CAN_FilterInitStructure.CAN_FilterMaskIdLow   = 0xFFFF;
```

5、对扩展帧进行过滤：(扩展帧不会被过滤掉)

```
CAN_FilterInitStructure.CAN_FilterIdHigh    = (((u32)slave_id<<3)&0xFFFF0000)>>16;
CAN_FilterInitStructure.CAN_FilterIdLow   = (((u32)slave_id<<3)|CAN_ID_EXT)&0xFFFF;
CAN_FilterInitStructure.CAN_FilterMaskIdHigh  = 0xFFFF;
CAN_FilterInitStructure.CAN_FilterMaskIdLow   = 0xFFFC;
```

6、对标准帧进行过滤：(标准帧不会被过滤掉)

```
CAN_FilterInitStructure.CAN_FilterIdHigh    = (((u32)slave_id<<21)&0xffff0000)>>16;
CAN_FilterInitStructure.CAN_FilterIdLow   = (((u32)slave_id<<21)|CAN_ID_STD)&0xffff;
CAN_FilterInitStructure.CAN_FilterMaskIdHigh  = 0xFFFF;
CAN_FilterInitStructure.CAN_FilterMaskIdLow   = 0xFFFC;
```

注：slave_id为要过滤的id号。

其中我们可以开启can错误中断，设置响应的标志位，在大循环里面不断的检测是否错误，一旦错误就重新配置can,这样有效地保证了CAN的正常通信。具体操作代码如下：

```
 /*CAN错误中断服务函数*/、
void CAN1_SCE_IRQHandler(void)
{
 CANWorkFlag &= ~CAN_RESET_COMPLETE;
}
/*CAN错误处理函数*/
/*****************************************************************************
 *函数名称：  CanErrorProcess
 *功能：      CAN故障，错误处理
 *参数说明：  无
 *****************************************************************************/
void CanErrorProcess(void)
{
 if ((CANWorkFlag & CAN_RESET_COMPLETE) == 0)
 {
  CAN1_Configuration();
//  CAN2_Configuration();
  CANWorkFlag |= CAN_RESET_COMPLETE;
 }

// if((CANWorkFlag & CAN2_RESET_COMPLETE) == 0)
// {
//  CAN1_Configuration();
//  CAN2_Configuration();
//  CANWorkFlag |= CAN2_RESET_COMPLETE;
// }
}
/*错误标志的定义*/
extern uint8 CANWorkFlag;
/*****************************************************************************
 *  CANWorkFlag 标志位掩码定义
 *****************************************************************************/
#define        CAN_INIT_COMPLETE             0x80    //CAN初始化完成标志
//#define        CAN_BUS_ERROR                 0x40    //CAN总线错误标志
#define        CAN_RESET_COMPLETE            0x40    //CAN控制器复位完成标志
#define        CAN2_INIT_COMPLETE            0x20    //CAN2初始化完成标志
//#define        CAN_BUS_ERROR                 0x40    //CAN总线错误标志
#define        CAN2_RESET_COMPLETE           0x10    //CAN2控制器复位完成标志
```

以上是我再调试时候添加的，挺有效的；

推荐 分享到：

阅读(16) | 评论(0) | 转载 (0) | 举报

单片机串口处理获得的经验（stm32）

硬件问题坑死人

最近读者

零洛 anfeidon

评论

零洛

发表

公司简介 - 联系方法 - 招聘信息 - 客户服务 - 隐私政策 - 博客风格 - 手机博客 - VIP博客 - 订阅此博客

网易公司版权所有 ©1997-2012