FAQ0106

常见问题解答

Flash带看门狗字节半字读写操作

Questions：带看门狗时，如何进行 Flash 字节半字的读写操作？

Answer：

在 FLASH 擦除操作 flash_sector_erase(sector_position * SECTOR_SIZE + FLASH_BASE);之前添加看门狗的喂狗操作 wdt_counter_reload ();。且喂狗时间开到最长 26 秒以防止在擦除 FLASH 和写入 FLASH 时看门狗超时产生系统复位。

喂狗时间试算：（其中，40kHz 为 HICK 时钟，0xff 为看门狗时钟最大分频值，0xfff 为最长喂狗时间）

$$t_{WDT} = \frac{1}{40kHz/0xff} \times 0xfff$$

在 BSP 库原有的读、写半字函数的基础上，增加了读、写字节函数。函数源码请见下文附录一。

```
void flash_read_byte(uint32_t read_addr, uint8_t *p_buffer, uint16_t num_read);
void flash_write_byte_nocheck(uint32_t write_addr, uint8_t *p_buffer, uint16_t num_write);
```

在 flash.h 中，#define WDT_EN 以使能 WDT 且喂狗，喂狗操作体现在 flash.c 的 flash_write_byte 和 flash_write 函数中，需要注意的是，在这两个函数的末尾，用户应当将看门狗重新配置为写 flash 前的设定值。函数源码请见下文附录一。

同时，添加了读、写 FLASH 函数操作通过参数的选择实现字节或半字操作功能，方便用户调用。

写 FLASH 函数操作通过参数的选择实现写字节或半字操作功能：

```
/**
  * @brief  写 flash 函数操作通过参数的选择实现写字节或半字操作功能
    * @param  read_addr: 第一个参数是数据将要写到 flash 的起始地址，
    * @param  _8pbuffer:  第二个参数为被写入 flash 的字节的数组（如果此次操作的数据是半字，则此值为 0），
    * @param  _16pbuffer: 第三个参数为被写入 flash 的半字的数组（如果此次操作的数据是字节，则此值为 0），
    * @param  num_read: 第四个参数为将要写入数据的个数，
    * @param  _16bit_is_1_8bit_is_0: 第五个为操作半字或者字节的选择（如果此值为 1，则是半字操作，字节则为 0）。
  */
void flash_write_halfword_or_byte(uint32_t write_addr,uint8_t* _8bitpbuffer, uint16_t*_16bitpbuffer,uint16_t
num_write,uint8_t _16bit_is_1_8bit_is_0)
{
    if(_16bit_is_1_8bit_is_0==0)
        flash_write_byte(write_addr,_8bitpbuffer,num_write);
    else
        flash_write(write_addr,_16bitpbuffer,num_write);
```

```
}
```

读 FLASH 函数操作通过参数的选择实现读字节或半字操作功能：

```
/**
  * @brief  读 flash 函数操作通过参数的选择实现读字节或半字操作功能
    * @param  read_addr: 第一个参数是数据将要读 flash 空间的地址，
    * @param  _8pbuffer:  第二个参数为被读入 flash 的字节的数组，
    * @param  _16pbuffer: 第三个参数为被读入 flash 的半字的数组，
    * @param  num_read:  第四个参数为将要读入数据的个数，
    * @param  _16bit_is_1_8bit_is_0:  第五个为半字或者字节的选择。
  */
void flash_read_halfword_or_byte(uint32_t read_addr,uint8_t *_8pbuffer, uint16_t *_16pbuffer,uint16_t num_read,uint8_t _16bit_is_1_8bit_is_0)
{
      if(_16bit_is_1_8bit_is_0 == 0)
            flash_read_byte( read_addr, _8pbuffer, num_read);
      else
            flash_read( read_addr, _16pbuffer, num_read);
}
```

附录一：将以下代码替换 bsp 库中\examples\flash\flash_write_read\src 文件夹 flash.c 代码：

```
#include "at32f403a_407_board.h"
#include "flash.h"
#if FLASH_SIZE<256
#define SECTOR_SIZE 1024  //字节
#else
#define SECTOR_SIZE 2048
#endif
uint16_t flash_buf[SECTOR_SIZE / 2];     //最多是 2K 字节
uint8_t flash_byte_buf[SECTOR_SIZE];   //最多是 2K 字节
/**
  * @brief   read data using halfword mode          从指定地址开始读出指定长度的数据
  * @param   read_addr: the address of reading          起始地址
  * @param   p_buffer: the buffer of reading data          数据指针
  * @param   num_read: the number of reading data        半字(16 位)数
  * @retval none
  */
void flash_read(uint32_t read_addr, uint16_t *p_buffer, uint16_t num_read)
{
  uint16_t i;
  for(i = 0; i < num_read; i++)
```

```
  {
    p_buffer[i] = *(uint16_t*)(read_addr); //读取 2 个字节
    read_addr += 2;    //偏移 2 个字节
  }
}
/**
  * @brief   read data using byte mode          从指定地址开始读出指定长度的数据
  * @param   read_addr: the address of reading          起始地址
  * @param   p_buffer: the buffer of reading data          数据指针
  * @param   num_read: the number of reading data          半字(16 位)数
  * @retval none
  */
void flash_read_byte(uint32_t read_addr, uint8_t *p_buffer, uint16_t num_read)
{
  uint16_t i;
  for(i = 0; i < num_read; i++)
  {
    p_buffer[i] = *(uint8_t*)(read_addr);   //读取 1 个字节
    read_addr++;        //偏移 1 个字节
  }
}
/**
  * @brief  读 flash 函数操作通过参数的选择实现读字节或半字操作功能
      * @param   read_addr: 第一个参数是数据将要读 flash 空间的地址，
      * @param   _8pbuffer:  第二个参数为被读入 flash 的字节的数组，
      * @param   _16pbuffer: 第三个参数为被读入 flash 的半字的数组，
      * @param   num_read:  第四个参数为将要读入数据的个数，
      * @param   _16bit_is_1_8bit_is_0:  第五个为半字或者字节的选择。
  */
void flash_read_halfword_or_byte(uint32_t read_addr,uint8_t *_8pbuffer, uint16_t *_16pbuffer,uint16_t num_read,uint8_t
_16bit_is_1_8bit_is_0)
{
      if(_16bit_is_1_8bit_is_0 == 0)
            flash_read_byte( read_addr, _8pbuffer, num_read);
      else
            flash_read( read_addr, _16pbuffer, num_read);
}
/**
  * @brief   write data using byte mode without checking   不检查的写入
  * @param   write_addr: the address of writing       起始地址
  * @param   p_buffer: the buffer of writing data       数据指针
  * @param   num_write: the number of writing data          字节(8 位)数
```

```
  * @retval none
  */
void flash_write_byte_nocheck(uint32_t write_addr, uint8_t *p_buffer, uint16_t num_write)
{
  uint16_t i;
  for(i = 0; i < num_write; i++)
  {
    flash_byte_program(write_addr, p_buffer[i]);
    write_addr++;//地址增加 1
  }
}
/**
  * @brief   write data using halfword mode without checking      不检查的写入
  * @param   write_addr: the address of writing        起始地址
  * @param   p_buffer: the buffer of writing data      数据指针
  * @param   num_write: the number of writing data          半字(16 位)数
  * @retval none
  */
void flash_write_nocheck(uint32_t write_addr, uint16_t *p_buffer, uint16_t num_write)
{
  uint16_t i;
  for(i = 0; i < num_write; i++)
  {
    flash_halfword_program(write_addr, p_buffer[i]);
    write_addr += 2;//地址增加 2
  }
}
/**
  * @brief   write data using halfword mode with checking  从指定地址开始写入指定长度的数据
  * @param   write_addr: the address of writing        起始地址(此地址必须为 2 的倍数!!)
  * @param   p_buffer: the buffer of writing data      数据指针
  * @param   num_write: the number of writing data  半字(16 位)数(就是要写入的 16 位数据的个数)
  * @retval none
  */
void flash_write(uint32_t write_addr, uint16_t *p_buffer, uint16_t num_write)
{
  uint32_t offset_addr;              //去掉 0X08000000 后的地址
  uint32_t sector_position;    //扇区地址
  uint16_t sector_offset;      //扇区内偏移地址(16 位字计算)
  uint16_t sector_remain;      //扇区内剩余地址(16 位字计算)
  uint16_t i;
```

```
  flash_unlock();          //解锁
  offset_addr = write_addr - FLASH_BASE;      //实际偏移地址
  sector_position = offset_addr / SECTOR_SIZE;      //扇区地址   0~512
  sector_offset = (offset_addr % SECTOR_SIZE) / 2; //在扇区内的偏移(2 个字节为基本单位)
  sector_remain = SECTOR_SIZE / 2 - sector_offset;//扇区剩余空间大小
  if(num_write <= sector_remain)
    sector_remain = num_write;     //不大于该扇区范围
  while(1)
  {
#ifdef WDT_EN
      /* disable register write protection */
      wdt_register_write_enable(TRUE);
      wdt_divider_set(WDT_CLK_DIV_256);                /* set the wdt divider value */
      wdt_reload_value_set(0xfff);/*  看门狗时间设置为最大  */
      wdt_counter_reload();/*  喂狗  */
#endif
    flash_read(sector_position * SECTOR_SIZE + FLASH_BASE, flash_buf, SECTOR_SIZE / 2); //读出整个扇区的内容
    for(i = 0; i < sector_remain; i++)        //校验数据
    {
      if(flash_buf[sector_offset + i] != 0xffff)
        break;//需要擦除
    }
    if(i < sector_remain)//需要擦除
    {
#ifdef WDT_EN
      wdt_counter_reload(); /*  喂狗  */
#endif
      flash_sector_erase(sector_position * SECTOR_SIZE + FLASH_BASE);//擦除这个扇区
      for(i = 0; i < sector_remain; i++)
      {
        flash_buf[i + sector_offset] = p_buffer[i];      //复制
      }
#ifdef WDT_EN
      wdt_counter_reload(); /*  喂狗  */
#endif
      flash_write_nocheck(sector_position * SECTOR_SIZE + FLASH_BASE, flash_buf, SECTOR_SIZE / 2);//写入整个
扇区
    }
    else
    {
#ifdef WDT_EN
      wdt_counter_reload(); /*  喂狗  */
```

```
#endif
        flash_write_nocheck(write_addr, p_buffer, sector_remain);//写已经擦除了的,直接写入扇区剩余区间
      }
      if(num_write == sector_remain)
        break;//写入结束了
      else//写入结束了
      {
        sector_position++;                                  //扇区地址增 1
        sector_offset = 0;                                        //偏移位置为 0
        p_buffer += sector_remain;                          //指针偏移
        write_addr += (sector_remain * 2);   //写地址偏移
        num_write -= sector_remain;                         //字节(16 位)数递减
        if(num_write > (SECTOR_SIZE / 2))
          sector_remain = SECTOR_SIZE / 2;   //下一个扇区还是写不完
        else
          sector_remain = num_write;                  //下一个扇区可以写完了
      }
    }
    flash_lock();             //上锁
#ifdef WDT_EN
        /* 用户喂狗值复位 */
        /*
        wdt_register_write_enable(TRUE);
        wdt_divider_set(WDT_CLK_DIV_4);
        wdt_reload_value_set(1000-1);
        wdt_counter_reload();
        */
#endif
}

/**
   * @brief   write data using byte mode with checking 从指定地址开始写入指定长度的数据
   * @param   write_addr: the address of writing       起始地址
   * @param   p_buffer: the buffer of writing data     数据指针
   * @param   num_write: the number of writing data 字节(8 位)数(就是要写入的 8 位数据的个数)
   * @retval none
   */
void flash_write_byte(uint32_t write_addr, uint8_t *p_buffer, uint16_t num_write)
{
    uint32_t offset_addr;          //去掉 0X08000000 后的地址
    uint32_t sector_position;     //扇区地址
    uint16_t sector_offset;       //扇区内偏移地址(16 位字计算)
```

```
  uint16_t sector_remain;       //扇区内剩余地址(16 位字计算)
  uint16_t i;

  flash_unlock();           //解锁
  offset_addr = write_addr - FLASH_BASE;      //实际偏移地址
  sector_position = offset_addr / SECTOR_SIZE;      //扇区地址   0~512
  sector_offset = (offset_addr % SECTOR_SIZE);      //在扇区内的偏移(2 个字节为基本单位)，即不足一页的占多少空间
  sector_remain = SECTOR_SIZE - sector_offset;      //扇区剩余空间大小
  if(num_write <= sector_remain)
    sector_remain = num_write;      //不大于该扇区范围
  while(1)
  {
#ifdef WDT_EN
      /* disable register write protection */
      wdt_register_write_enable(TRUE);
      wdt_divider_set(WDT_CLK_DIV_256);
      wdt_reload_value_set(0xfff);/* 看门狗时间设置为最大 */
      wdt_counter_reload();/* 喂狗 */
#endif
    flash_read_byte(sector_position * SECTOR_SIZE + FLASH_BASE, flash_byte_buf, SECTOR_SIZE);  //读出整个扇区的内容
    for(i = 0; i < sector_remain; i++)        //校验数据
    {
      if(flash_byte_buf[sector_offset + i] != 0xff)
        break;//需要擦除
    }
    if(i < sector_remain)//需要擦除
    {
#ifdef WDT_EN
      wdt_counter_reload(); /* 喂狗 */
#endif
      flash_sector_erase(sector_position * SECTOR_SIZE + FLASH_BASE);//擦除这个扇区
      for(i = 0; i < sector_remain; i++)
      {
        flash_byte_buf[i + sector_offset] = p_buffer[i];      //复制
      }
#ifdef WDT_EN
      wdt_counter_reload(); /* 喂狗 */
#endif
      flash_write_byte_nocheck(sector_position * SECTOR_SIZE + FLASH_BASE, flash_byte_buf, SECTOR_SIZE);//写入整个扇区
    }
```

```
      else
      {
#ifdef WDT_EN
        wdt_counter_reload(); /* 喂狗 */
#endif
          flash_write_byte_nocheck(write_addr, p_buffer, sector_remain);//写已经擦除了的,直接写入扇区剩余区间
      }
      if(num_write == sector_remain)
        break;//写入结束了
      else//写入结束了
      {
        sector_position++;                                  //扇区地址增 1
        sector_offset = 0;                                      //偏移位置为 0
        p_buffer += sector_remain;                          //指针偏移
        write_addr += (sector_remain);   //写地址偏移
        num_write -= sector_remain;                         //字节(16 位)数递减
        if(num_write > (SECTOR_SIZE))
          sector_remain = SECTOR_SIZE;        //下一个扇区还是写不完
        else
          sector_remain = num_write;                    //下一个扇区可以写完了
      }
    }
    flash_lock();             //上锁
#ifdef WDT_EN
        /* 用户喂狗值复位 */
        /*
        wdt_register_write_enable(TRUE);
        wdt_divider_set(WDT_CLK_DIV_4);
        wdt_reload_value_set(1000-1);
        wdt_counter_reload();
        */
#endif
}
/**
  * @brief  写 flash 函数操作通过参数的选择实现写字节或半字操作功能
        * @param  read_addr: 第一个参数是数据将要写到 flash 的起始地址，
        * @param  _8pbuffer:  第二个参数为被写入 flash 的字节的数组（如果此次操作的数据是半字，则此值为 0），
        * @param  _16pbuffer: 第三个参数为被写入 flash 的半字的数组（如果此次操作的数据是字节，则此值为 0），
        * @param  num_read: 第四个参数为将要写入数据的个数，
        * @param  _16bit_is_1_8bit_is_0:  第五个为操作半字或者字节的选择（如果此值为 1，则是半字操作，字节则为 0）。
  */
void     flash_write_halfword_or_byte(uint32_t     write_addr,uint8_t*     _8bitpbuffer,     uint16_t*_16bitpbuffer,uint16_t
```

```
num_write,uint8_t _16bit_is_1_8bit_is_0)
{
    if(_16bit_is_1_8bit_is_0==0)
        flash_write_byte(write_addr,_8bitpbuffer,num_write);
    else
        flash_write(write_addr,_16bitpbuffer,num_write);
}
```

附录二：将以下代码替换 bsp 库中\examples\flash\flash_write_read\inc 文件夹 flash.h 代码：

```
#include "at32f403a_407_board.h"
#define FLASH_SIZE 1024                    //所选 AT32F4xx 的 FLASH 容量大小(单位为 K)
#define WDT_EN                 //使能 WDT 且喂狗  (注释该句可以看到程序未执行完被 WDT 复位)
/** @defgroup FLASH_write_read_functions    */
void flash_read(uint32_t read_addr, uint16_t *p_buffer, uint16_t num_read);
void flash_read_byte(uint32_t read_addr, uint8_t *p_buffer, uint16_t num_read);
void flash_read_halfword_or_byte(uint32_t read_addr,uint8_t *_8pbuffer, uint16_t *_16pbuffer,uint16_t num_read,uint8_t
_16bit_is_1_8bit_is_0);
void flash_write_byte_nocheck(uint32_t write_addr, uint8_t *p_buffer, uint16_t num_write);
void flash_write_nocheck(uint32_t write_addr, uint16_t *p_buffer, uint16_t num_write);
void flash_write(uint32_t write_addr,uint16_t *p_Buffer, uint16_t num_write);
void flash_write_byte(uint32_t write_addr, uint8_t *p_buffer, uint16_t num_write);
void flash_write_halfword_or_byte(uint32_t write_addr,uint8_t* _8bitpbuffer, uint16_t*_16bitpbuffer,uint16_t
num_write,uint8_t _16bit_is_1_8bit_is_0);
```

类型： MCU 应用
适用型号： AT32F4 系列
主功能： Flash, WDT
次功能： 无

文档版本历史

日期	版本	变更
2022.3.24	2.0.0	最初版本

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。