FAQ0106

Frequently Asked Questions

Byte and half-word Read and write Flash when WDT enabled

Questions:

How to read and write Flash in byte and half-word mode when watchdog is enabled?

Answer:

Add ***wdt_counter_reload ();*** before the ***flash_sector_erase(sector_position * SECTOR_SIZE + FLASH_BASE);*** and set watchdog time to 26 seconds to avoid the occurrence of watchdog timeout-triggered system reset during Flash erase or write operation.

Watchdog feeding time: (40 kHz HICK, 0xff represents the maximum clock frequency division, and 0xfff represents the maximum watchdog feeding time)

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

Add read/write byte function on top of the previous BSP read/write half-word function. See appendix 1 for function source code.

```
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);
```

In ***flash.h***, use "***#define WDT_EN***" to enable WDT and feed it. Feeding operation is implemented in the "***flash_write_byte***" and "***flash_write***" functions in ***flash.c***. Note that at the end of the two functions, users should load the watchdog with the value programmed before Flash write operation. See *appendix 1* for source code.

Read/write Flash function is made easy for users to select byte or half-word operation mode

Write Flash function: select byte or half-word mode by selecting the following parameters

```
/**
  * @brief  Write Flash function selects write byte or half-word mode through the following parameters
    * @param  read_addr:  (parameter 1) Flash start address where data are to be written
    * @param  _8pbuffer:  (parameter 2) Array of bytes that are written to Flash (this value is 0 for half-word data)
    * @param  _16pbuffer: (parameter 3) Array of half words that are written to Flash (this value is 0 for byte data)
    * @param  num_read:  (parameter 4) the count of data to be written
    * @param  _16bit_is_1_8bit_is_0: (parameter 5) half words or byte selection (if this value is 1, it is a half-word operation. If this value is 0, it means byte operation mode)
  */
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);
}
```

Read Flash function: select byte or half-word operation mode through the following parameters:

```
/**
  * @brief  Read Flash function selects read byte or half words through the following parameters
    * @param  read_addr:  (parameter 1) the address of Flash to be read
    * @param  _8pbuffer:  (parameter 2) Array of bytes that are read into Flash
    * @param  _16pbuffer: (parameter 3) Array of half words that are read into Flash
    * @param  num_read:  (parameter 4) the count of data to be read
    * @param  _16bit_is_1_8bit_is_0:  (parameter 5) half-word or byte selection
  */
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);
}
```

Appendix 1: replace ***flash.c code*** located in the BSP library\examples\flash\flash_write_read\src with the following code

```
#include "at32f403a_407_board.h"
#include "flash.h"
#if FLASH_SIZE<256
#define SECTOR_SIZE 1024  //byte
#else
#define SECTOR_SIZE 2048
#endif
uint16_t flash_buf[SECTOR_SIZE / 2];     //up to 2KB
uint8_t flash_byte_buf[SECTOR_SIZE];    //up to 2KB
/**
  * @brief   read data using halfword mode          read data at a given address
  * @param   read_addr: the address of reading          start address
  * @param   p_buffer: the buffer of reading data          data pointer
  * @param   num_read: the number of reading data        the number of half words (16 bits)
  * @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);  //read 2KB
    read_addr += 2;     //offset 2KB
  }
}
```

```
/**
  * @brief   read data using byte mode          read data at at a given address
  * @param   read_addr: the address of reading          start address
  * @param   p_buffer: the buffer of reading data          data pointer
  * @param   num_read: the number of reading data          the number of half words (16 bits)
  * @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);    //read 1KB
    read_addr++;         //offset 1KB
  }
}
/**
  * @brief  Read Flash function select byte or half-word mode through the following parameters
      * @param   read_addr:  (parameter 1) the address of Flash to be read
      * @param   _8pbuffer:   (parameter 2) Array of bytes that are to be read into Flash
      * @param   _16pbuffer: (parameter 3) Array of half words that are to be read into Flash
      * @param   num_read:  (parameter 4) the number of data to be read
      * @param   _16bit_is_1_8bit_is_0:  (parameter 5) select half words or byte
  */
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
  * @param   write_addr: the address of writing        start address
  * @param   p_buffer: the buffer of writing data        data pointer
  * @param   num_write: the number of writing data          the number of bytes (8 bits)
  * @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++;//address plus 1
    }
  }
  /**
    * @brief    write data using halfword mode without checking
    * @param    write_addr: the address of writing        start address
    * @param    p_buffer: the buffer of writing data      data pointer
    * @param    num_write: the number of writing data  the number of half words (16 bits)
    * @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;//address plus 2
    }
  }
  /**
    * @brief    write data using halfword mode with checking     write data of a certain length to a given address
    * @param    write_addr: the address of writing        start address (it must be a multiple of 2)
    * @param    p_buffer: the buffer of writing data      data pointer
    * @param    num_write: the number of writing data  the number of half words (the number of 16-bit data to be written)
    * @retval none
    */
  void flash_write(uint32_t write_addr, uint16_t *p_buffer, uint16_t num_write)
  {
    uint32_t offset_addr;                 // this is the address after removing 0X08000000
    uint32_t sector_position;     //sector address
    uint16_t sector_offset;       //sector offset address (counted in 16-bit word)
    uint16_t sector_remain;       // remaining address in a sector (counted in 16-bit word)
    uint16_t i;

    flash_unlock();          //unlock
    offset_addr = write_addr - FLASH_BASE;      //actual offset address
    sector_position = offset_addr / SECTOR_SIZE;       //sector address    0~512
    sector_offset = (offset_addr % SECTOR_SIZE) / 2; //offset within a sector (in two bytes unit)
    sector_remain = SECTOR_SIZE / 2 - sector_offset;//the remaining size in a sector
    if(num_write <= sector_remain)
      sector_remain = num_write;     // no more than this sector range
    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);/* set the maximum watchdog time */
      wdt_counter_reload();/* feed the watchdog */
#endif
    flash_read(sector_position * SECTOR_SIZE + FLASH_BASE, flash_buf, SECTOR_SIZE / 2); //read the entire sector
    for(i = 0; i < sector_remain; i++)        //data check
    {
      if(flash_buf[sector_offset + i] != 0xffff)
        break;//erase
    }
    if(i < sector_remain)//erase
    {
#ifdef WDT_EN
      wdt_counter_reload(); /* feed the watchdog */
#endif
      flash_sector_erase(sector_position * SECTOR_SIZE + FLASH_BASE);//erase this sector
      for(i = 0; i < sector_remain; i++)
      {
        flash_buf[i + sector_offset] = p_buffer[i];      //copy
      }
#ifdef WDT_EN
      wdt_counter_reload(); /* feed the watchdog */
#endif
      flash_write_nocheck(sector_position * SECTOR_SIZE + FLASH_BASE, flash_buf, SECTOR_SIZE / 2);//write the
entire sector
    }
    else
    {
#ifdef WDT_EN
      wdt_counter_reload(); /* feed watchdog */
#endif
      flash_write_nocheck(write_addr, p_buffer, sector_remain);// write the remaining sector address
    }
    if(num_write == sector_remain)
      break;//end of write
    else//end of write
    {
      sector_position++;                                //sector address plus 1
      sector_offset = 0;                                      //offset 0
      p_buffer += sector_remain;                        //pointer offset
```

```
        write_addr += (sector_remain * 2);   //write address offset
        num_write -= sector_remain;                    // the number of byte ((16 bits)) is decremental
        if(num_write > (SECTOR_SIZE / 2))
          sector_remain = SECTOR_SIZE / 2;   // the next sector is still not finished
        else
          sector_remain = num_write;                   //the next sector is finished
      }
    }
    flash_lock();             //lock
#ifdef WDT_EN
        /* user watchdog value reset */
        /*
        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   write data of a certain length at a given address
  * @param   write_addr: the address of writing        start address
  * @param   p_buffer: the buffer of writing data      data pointer
  * @param   num_write: the number of writing data  the number of bytes (means the number of 8-bit data to be written)
  * @retval none
  */
void flash_write_byte(uint32_t write_addr, uint8_t *p_buffer, uint16_t num_write)
{
  uint32_t offset_addr;          //the address after removing0X08000000
  uint32_t sector_position;     //sector address
  uint16_t sector_offset;       //sector offset address (counted in 16-bit words)
  uint16_t sector_remain;       // remaining address in a sector (counted in 16-bit words)
  uint16_t i;

  flash_unlock();          //unlock
  offset_addr = write_addr - FLASH_BASE;      //actual offset address
  sector_position = offset_addr / SECTOR_SIZE;      //sector address    0~512
  sector_offset = (offset_addr % SECTOR_SIZE);      // offset within a sector (in 2KB terms), means the space occupied
      by less than a sector
  sector_remain = SECTOR_SIZE - sector_offset;     // remaining sector size
  if(num_write <= sector_remain)
    sector_remain = num_write;     //no more than this sector size
  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);/* set the maximum watchdog time */
      wdt_counter_reload();/* feed the watchdog */
#endif
    flash_read_byte(sector_position * SECTOR_SIZE + FLASH_BASE, flash_byte_buf, SECTOR_SIZE);  //read the entire sector
    for(i = 0; i < sector_remain; i++)       //data check
    {
      if(flash_byte_buf[sector_offset + i] != 0xff)
        break;//need erase operation
    }
    if(i < sector_remain)// need erase operation
    {
#ifdef WDT_EN
      wdt_counter_reload(); /* feed the watchdog */
#endif
      flash_sector_erase(sector_position * SECTOR_SIZE + FLASH_BASE);// erase the entire sector
      for(i = 0; i < sector_remain; i++)
      {
        flash_byte_buf[i + sector_offset] = p_buffer[i];     //copy
      }
#ifdef WDT_EN
      wdt_counter_reload(); /* feed the watchdog */
#endif
      flash_write_byte_nocheck(sector_position * SECTOR_SIZE + FLASH_BASE, flash_byte_buf, SECTOR_SIZE);//write the entire sector
    }
    else
    {
#ifdef WDT_EN
      wdt_counter_reload(); /* feed watchdog */
#endif
      flash_write_byte_nocheck(write_addr, p_buffer, sector_remain);//write the remaining sector range
    }
    if(num_write == sector_remain)
      break;//end of write
    else//end of write
    {
      sector_position++;                                  //sector address plus 1
      sector_offset = 0;                                  //offset address 0
```

```
        p_buffer += sector_remain;                          //pointer offset
        write_addr += (sector_remain);   //write address offset
        num_write -= sector_remain;                         //the number of byte (16 bits) is decremental
        if(num_write > (SECTOR_SIZE))
          sector_remain = SECTOR_SIZE;        //the next sector is still finished
        else
          sector_remain = num_write;                    //the next sector is finished
      }
    }
    flash_lock();             //lock
#ifdef WDT_EN
      /* ser watchdog feed value reset */
      /*
      wdt_register_write_enable(TRUE);
      wdt_divider_set(WDT_CLK_DIV_4);
      wdt_reload_value_set(1000-1);
      wdt_counter_reload();
      */
#endif
}
/**
   * @brief  write Flash function selects byte or half-word mode through the following parameters
      * @param   read_addr:  (parameter 1) Flash start address to be written
      * @param   _8pbuffer: (parameter 2) Array of bytes that are to be written into Flash (this value is 0 if it is a half-word mode)
      * @param   _16pbuffer: (parameter 3) Array of half words that are to be written into Flash (this value is 0 if it is a byte operation mode)
      * @param   num_read:  (parameter 4) the number of data to be written
      * @param   _16bit_is_1_8bit_is_0:  (parameter 5) select byte or half-word mode (if this value is 1, it means a half-word operation; if this value is 0, it is a byte operation)
   */
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);
}
```

Appendix 2: replace ***flash.h*** code located in the BSP library\examples\flash\flash_write_read\inc with the following code

```
#include "at32f403a_407_board.h"

#define FLASH_SIZE 1024                    //AT32F4xx FLASH size (unit: K)

#define WDT_EN                 //enable WDT and feed it (if this macro definition is commented, you can see that program is reset by WDT even if it has not yet finished)

/** @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);
```

Type: MCU application
Applicable products: AT32F4xx series
Main function: Flash, WDT
Other function: None

Document revision history

Date	Revision	Changes
2022.3.24	2.0.0	Initial release

IMPORTANT NOTICE – PLEASE READ CAREFULLY

Purchasers are solely responsible for the selection and use of ARTERY's products and services, and ARTERY assumes no liability whatsoever relating to the choice, selection or use of the ARTERY products and services described herein

No license, express or implied, to any intellectual property rights is granted under this document. If any part of this document deals with any third party products or services, it shall not be deemed a license granted by ARTERY for the use of such third party products or services, or any intellectual property contained therein, or considered as a warranty regarding the use in any manner of such third party products or services or any intellectual property contained therein.

Unless otherwise specified in ARTERY's terms and conditions of sale, ARTERY provides no warranties, express or implied, regarding the use and/or sale of ARTERY products, including but not limited to any implied warranties of merchantability, fitness for a particular purpose (and their equivalents under the laws of any jurisdiction), or infringement on any patent, copyright or other intellectual property right.

Purchasers hereby agree that ARTERY's products are not designed or authorized for use in: (A) any application with special requirements of safety such as life support and active implantable device, or system with functional safety requirements; (B) any aircraft application; (C) any aerospace application or environment; (D) any weapon application, and/or (E) or other uses where the failure of the device or product could result in personal injury, death, property damage. Purchasers' unauthorized use of them in the aforementioned applications, even if with a written notice, is solely at purchasers' risk, and Purchasers are solely responsible for meeting all legal and regulatory requirements in such use.

Resale of ARTERY products with provisions different from the statements and/or technical characteristics stated in this document shall immediately void any warranty grant by ARTERY for ARTERY's products or services described herein and shall not create or expand any liability of ARTERY in any manner whatsoever.