

基于 KE02 的 SD 卡升级 Demo

作者: Jianhui Tong

邮箱: Jianhui.tong@nxp.com

简介

本实验基于 FRDM-KE02Z40M, 完成了使用 SD 卡升级用户程序的功能。使用的 SD 卡文件格式为 FAT32 格式, 用户程序为二进制的 bin 文件。主要硬件使用了 FRDM-KE02Z40M 开发板和一个 SPI 通信的 SD 卡卡座模块, 软件部分的升级程序占用 FLASH 约 14KB, RAM 约 1KB。

第一部分 FatFS 的移植

1. 下载 FatFS 的移植 sample:

http://elm-chan.org/fsw/ff/00index_e.html

Resources

The FatFs module is a free software opened for education, research and development. You can use, modify and/or redistribute it for personal projects or commercial products without any restriction under your responsibility. For further information, refer to the application note.

- Read first: [FatFs module application note](#) [April 12, 2016]
- Download: [FatFs R0.12](#) | [Updates](#) | [Patches](#) [April 29, 2016]
- Download: [FatFs sample projects for various platforms](#) [April 12, 2016]
- Download: [Old Releases](#)

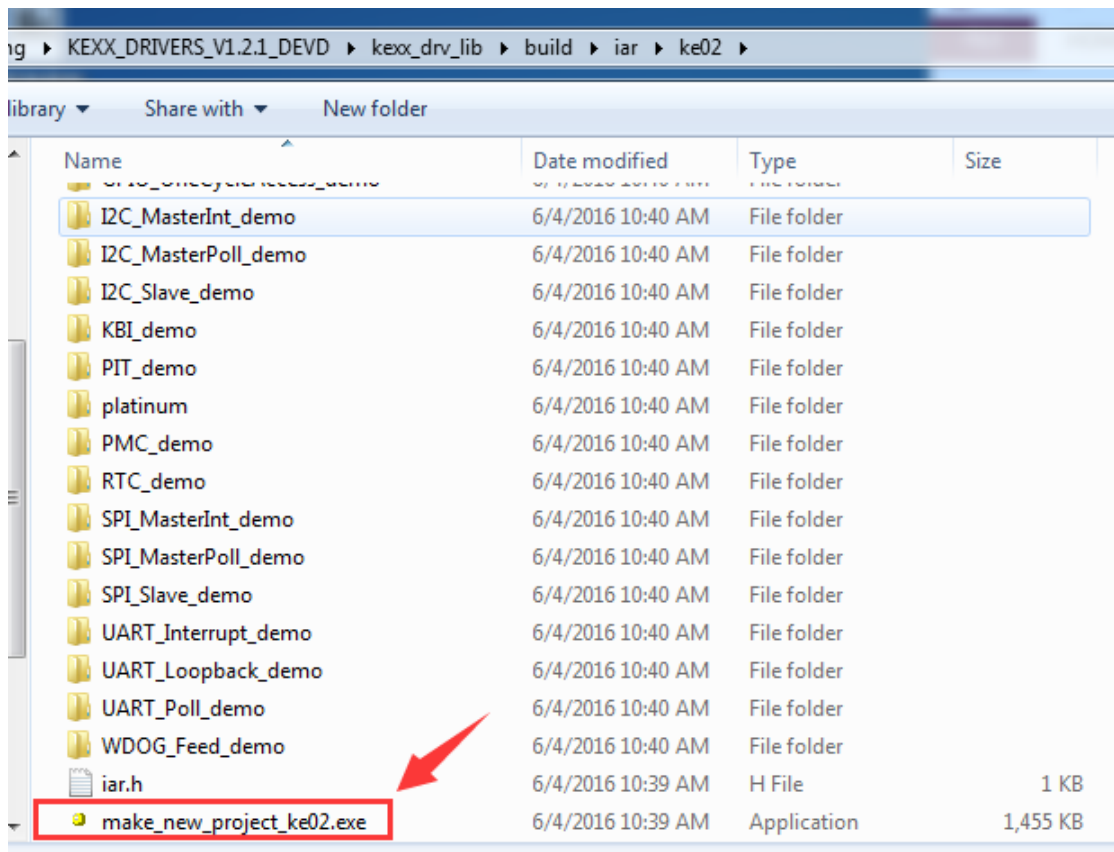
2. 下载 NXP 官方 KE 驱动库:

http://www.nxp.com/webapp/sps/download/license.jsp?colCode=KEXX_DRIVERS_V1.2.1_DEVD&location=null&fsrch=1&sr=6&pageNum=1&Parent_nodeId=&Parent_pageType

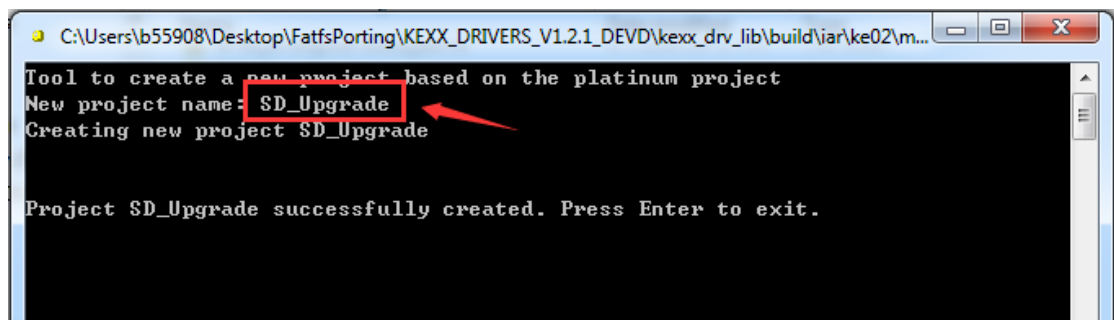
3. 下载完成后解压压缩文件

Name	Date modified	Type	Size
ffsample	5/23/2016 8:57 PM	File folder	
KEXX_DRIVERS_V1.2.1_DEVD	6/4/2016 10:39 AM	File folder	
ffsample.zip	5/12/2016 4:06 PM	Compressed (zipp...	6,721 KB
KEXX_DRIVERS_V1.2.1_DEVD.zip	5/23/2016 8:49 PM	Compressed (zipp...	24,151 KB

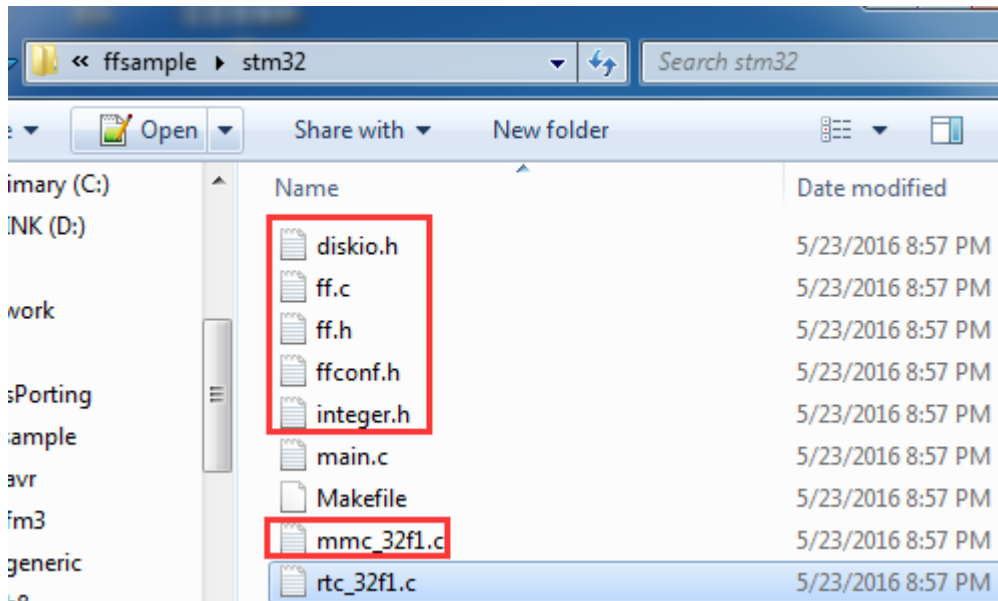
4. 使用 make_new_project_ke02.exe 工具新建工程



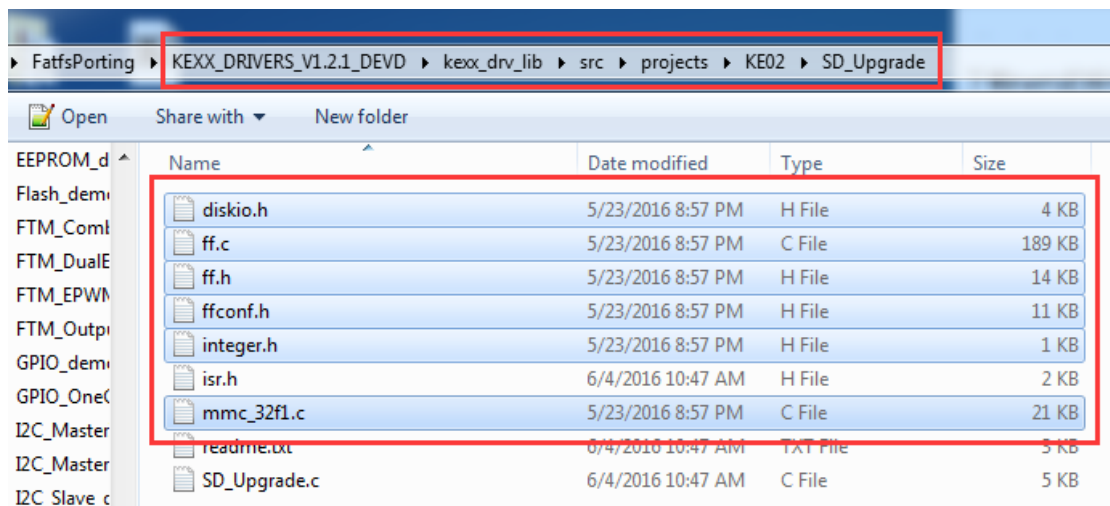
5. 输入工程名字，如“SD_Upgrade”，按回车确认，再按回车退出。



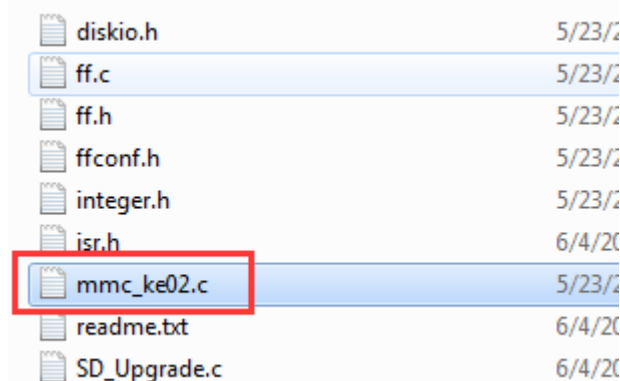
6. 回到 ffsample 文件夹，我们以 stm32 的移植范例为例，我们只需要拷贝其中的 mmc_32f1.c, integer.h, ffconf.h, ff.h, ff.c, diskio.h 六个文件。



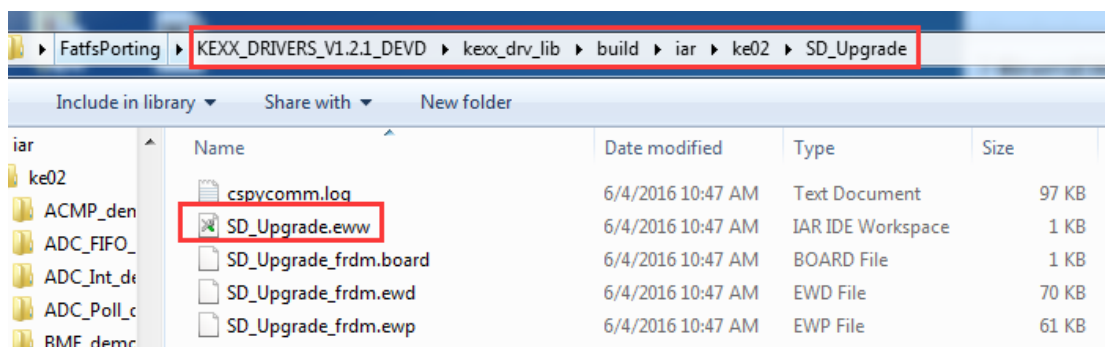
- 回到 SD_Upgrade 工程存放源文件的目录，并将复制的文件粘贴进来。



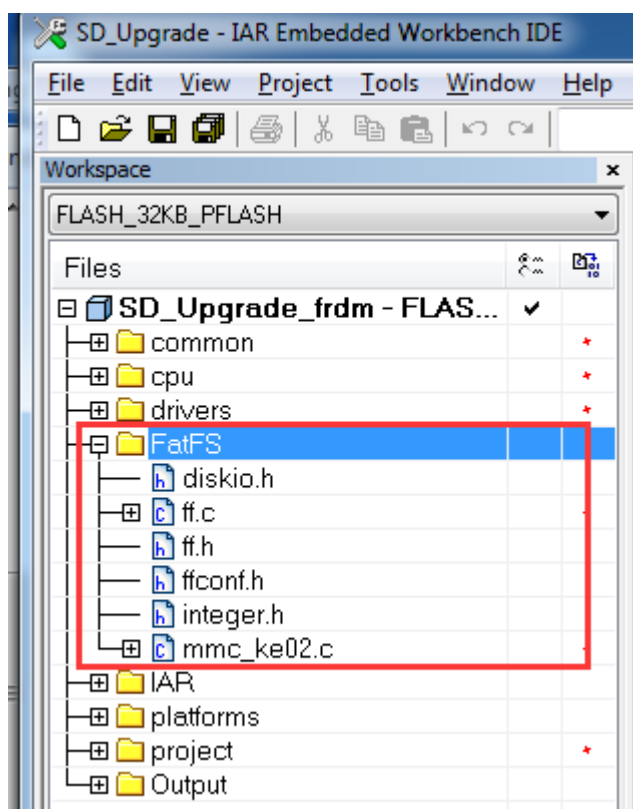
- 将 mmc_32f1.c 重命名为 mmc_ke02.c



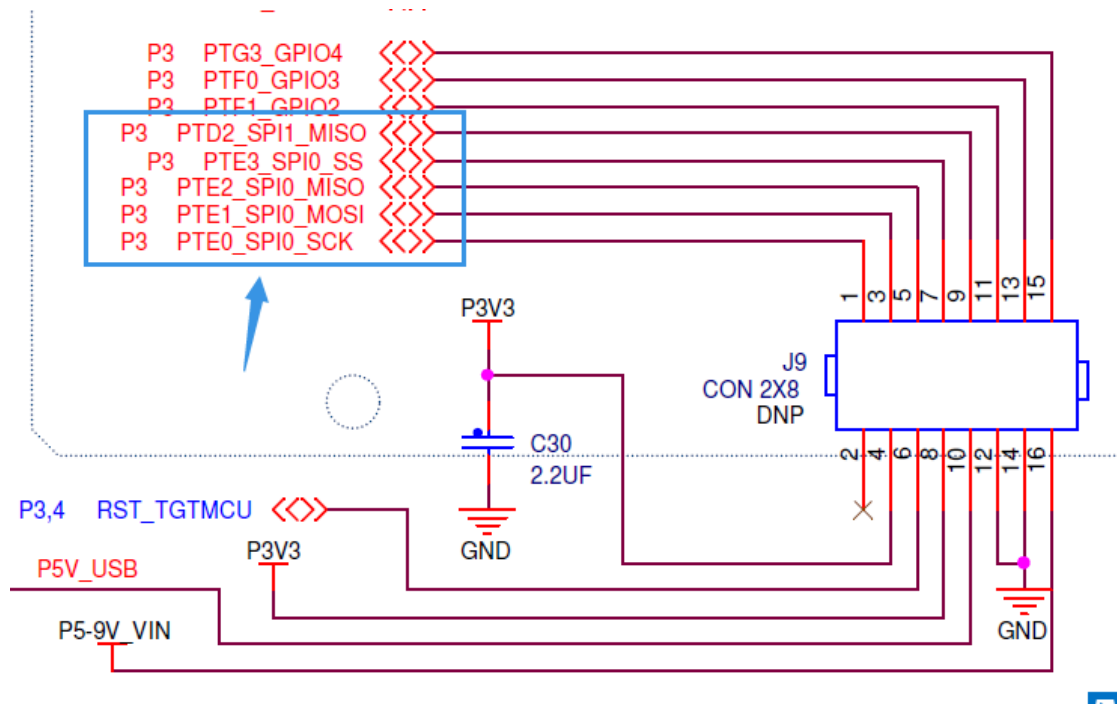
- 回到 SD_Upgrade 工程所在目录，并打开 IAR 工程文件。



10. 在 IAR 工程窗口中，新建 FatFS 文件夹，并将刚才的 6 个文件添加进来。



11. 本 Demo 使用 FRDM-KE02Z40M 开发板，使用 SPI0 与 SD 卡通信，PTD2 用于检测 SD 卡的 CD 信号(Card Detection)。参照开发板的原理图，实际需要配置的引脚如下图所示。



12. 打开 mmc_ke02.c 文件，开始修改。

```

// #define SPI_CH      1      /* SPI channel to use = 1: SPI1, 11: SPI1/remap, 2: SPI2 */

// #define FCLK_SLOW() { SPIx_CR1 = (SPIx_CR1 & ~0x38) | 0x28; } /* Set SCLK = PCLK / 64 */
// #define FCLK_FAST() { SPIx_CR1 = (SPIx_CR1 & ~0x38) | 0x00; } /* Set SCLK = PCLK / 2 */
// #define FCLK_SLOW() SPI_SetBaudRate(SPI0, BUS_CLK_HZ, 100000) // 设置慢速SPI速度为100K
// #define FCLK_FAST() SPI_SetBaudRate(SPI0, BUS_CLK_HZ, 1000000) // 设置快速SPI速度为1M

```

SPI_CH 用于选择 SPI 通道，在本 Demo 中使用固定的 SPI，因此可以去掉此定义。
FCLK_SLOW 和 FCLK_FAST 按照上图所示进行修改。

13. 将 CS_HIGH 和 CS_LOW 进行重新定义，其他未使用的部分可以注释掉。

```

// #if SPI_CH == 1      /* PA4:MMC_CS, PA5:MMC_SCLK, PA6:MMC_DO, PA7:MMC_DI, PC4:MMC_CD */
// #define CS_HIGH()    GPIOA_BSRR = _BV(4)
// #define CS_LOW()     GPIOA_BSRR = _BV(4+16)
// #define CS_HIGH()    GPIO_PinSet(GPIO_PTE3) // 设置CS管脚
// #define CS_LOW()     GPIO_PinClear(GPIO_PTE3)

// #define MMC_CD       !(GPIOC_IDR & _BV(4)) /* Card detect (yes:true, no:false, default:true) */
// #define MMC_WP       0 /* Write protected (yes:true, no:false, default:false) */
// #define SPIx_CR1      SPI1_CR1
// #define SPIx_SR       SPI1_SR
// #define SPIx_DR       SPI1_DR

```

14. SPIxENABLE() 在这里是以宏定义的方式出现，我们注释掉，并在文件后面使用函数进行替代。

```

// #define SPIxENABLE() {\
    __enable_peripheral(SPI1EN);\
    __enable_peripheral(IOPAEN);\
    __enable_peripheral(IOPCEN);\
    __gpio_conf_bit(GPIOA, 4, OUT_PP);
    __gpio_conf_bit(GPIOA, 5, ALT_PP);
    GPIOA_BSRR = _BV(6); __gpio_conf_bit(GPIOA, 6, IN_PUL); /* PA6: MMC_DO with pull-up */
    __gpio_conf_bit(GPIOA, 7, ALT_PP);
    GPIOC_BSRR = _BV(4); __gpio_conf_bit(GPIOC, 4, IN_PUL); /* PC4: MMC_CD with pull-up */
    SPIx_CR1 = _BV(9) | _BV(8) | _BV(6) | _BV(2);
}

```

在函数定义部分前，加上 SPIxENABLE 函数的定义。

```

/*-----*/
/* SPI controls (Platform dependent) */
/*-----*/

void SPIxENABLE(void)
{
    SPI_ConfigType sSPIConfig = {0};

    SIM_RemapSPI0Pin(); //SPI 管脚复用

    sSPIConfig.u32BitRate = 10000;
    sSPIConfig.u32BusClkHHz = BUS_CLK_HZ;
    sSPIConfig.sSettings.bModuleEn = 1;
    sSPIConfig.sSettings.bMasterMode = 1;
    sSPIConfig.sSettings.bClkPhase1 = 0;
    sSPIConfig.sSettings.bMasterAutoDriveSS = 0;
    sSPIConfig.sSettings.bClkPolarityLow = 1;
    SPI_Init(SPI0, &sSPIConfig);

    GPIO_Init(GPIOB, GPIO_PTE3_MASK, GPIO_PinOutput); //PTE3 SPI0 SS
}

```

15. 文件中还有一些使用 SPI 其他通道时相应的宏定义，都可以注释掉。

```

//#elif SPI_CH == 11 /* PA15:MMC_CS, PB3:MMC_SCLK, PB4:MMC_DO, PB5:MMC_DI, PB6:MMC_CD */
//#define CS_HIGH() GPIOA_BSRR = _BV(15)
//#define CS_LOW() GPIOA_BSRR = _BV(15+16)
//#define MMC_CD !(GPIOB_IDR & _BV(6)) /* Card detect (yes:true, no:false, default:true) */
//#define MMC_WP 0 /* Write protected (yes:true, no:false, default:false) */
//#define SPIx_CR1 SPI1_CR1
//#define SPIx_SR SPI1_SR
//#define SPIx_DR SPI1_DR
//#define SPIxENABLE() {\
//    AFIO_MAPR |= _BV(1);
//    __enable_peripheral(SPI1EN);\
//    __enable_peripheral(IOPAEN);\
//    __enable_peripheral(IOBEN);\
//    __gpio_conf_bit(GPIOA, 15, OUT_PP); /* PA15: MMC_CS */\
//    __gpio_conf_bit(GPIOB, 3, ALT_PP); /* PB3: MMC_SCLK */\
//    GPIOB_BSRR = _BV(4); __gpio_conf_bit(GPIOB, 4, IN_PUL); /* PB4: MMC_DO with pull-up */\
//    __gpio_conf_bit(GPIOB, 5, ALT_PP); /* PB5: MMC_DI */\
//    GPIOB_BSRR = _BV(6); __gpio_conf_bit(GPIOB, 6, IN_PUL); /* PB6: MMC_CD with pull-up */\
//    SPIx_CR1 = _BV(9)|_BV(8)|_BV(6)|_BV(2); /* Enable SPI1 */\
//}
//
//#elif SPI_CH == 2 /* PB12:MMC_CS, PB13:MMC_SCLK, PB14:MMC_DO, PB15:MMC_DI, PB8:MMC_CD */
//#define CS_HIGH() GPIOB_BSRR = _BV(12)
//#define CS_LOW() GPIOB_BSRR = _BV(12+16)
//#define MMC_CD !(GPIOB_IDR & _BV(8)) /* Card detect (yes:true, no:false, default:true) */
//#define MMC_WP 0 /* Write protected (yes:true, no:false, default:false) */
//#define SPIx_CR1 SPI2_CR1
//#define SPIx_SR SPI2_SR
//#define SPIx_DR SPI2_DR
//#define SPIxENABLE() {\
//    __enable_peripheral(SPI2EN);\
//    __enable_peripheral(IOBEN);\
//    __enable_peripheral(IOPDEN);\
//    __gpio_conf_bit(GPIOB, 12, OUT_PP); /* PB12: MMC_CS */\
//    __gpio_conf_bit(GPIOB, 13, ALT_PP); /* PB13: MMC_SCLK */\
//    GPIOB_BSRR = _BV(14); __gpio_conf_bit(GPIOB, 14, IN_PUL); /* PB14: MMC_DO with pull-up */\
//    __gpio_conf_bit(GPIOB, 15, ALT_PP); /* PB15: MMC_DI */\
//    GPIOB_BSRR = _BV(8); __gpio_conf_bit(GPIOB, 8, IN_PUL); /* PB8: MMC_CD with pull-up */\
//    SPIx_CR1 = _BV(9)|_BV(8)|_BV(6)|_BV(2); /* Enable SPI1 */\
//}
//
//#endif

```

16. 修改头文件部分，注释掉 STM32F100 头文件，添加 common,spi,sim,gpio 的头文件。

```

//#include "STM32F100.h"
#include "common.h"
#include "spi.h"
#include "sim.h"
#include "gpio.h"
#include "diskio.h"

```

17. 我们直接使用指令进行延时替代原有的定时器延时，因此需要将程序中延时部分进行修改。

```

#define MS_FACTOR 4000
void delay_ms(uint32_t cnt)
{
    uint32_t i;

    while(cnt--)
    {
        i = MS_FACTOR;
        while(i--);
    }
}

```

添加 MS_FACTOR 宏定义，和 delay_ms 延时函数。MS_FACTOR 可以根据实际 CPU 工作频率调整，使得每 1ms，delay_ms 函数中的 cnt 变量减 1。

对原来使用 Timer1 和 Timer2 变量进行延时的地方进行修改。

```

/* Initialize MMC interface */
static
void init_spi (void)
{
    SPIxENABLE();          /* Enable SPI function */
    CS_HIGH();              /* Set CS# high */

    //for (Timer1 = 10; Timer1; ) ; /* 10ms */
    delay_ms(10);
}

```

修改 init_spi 函数。

```

static
int rcvr_datablock ( /* 1:OK, 0:Error */
    BYTE *buff, /* Data buffer */
    UINT btr /* Data block length (byte) */
)
{
    BYTE token;

    UINT i = MS_FACTOR;

    Timer1 = 200;
    do { /* Wait for DataStart token in timeout of 200ms */
        token = xchg_spi(0xFF);
        /* This loop will take a time. Insert rot_rdq() here for multitask environent. */
        if(i>0)
        {
            i--;
        }
        else
        {
            i = MS_FACTOR;
            Timer1--;
        }
    } while ((token == 0xFF) && Timer1);

    // Timer1 = 200;
    // do { /* Wait for DataStart token in timeout of 200ms */
    //     token = xchg_spi(0xFF);
    //     /* This loop will take a time. Insert rot_rdq() here for multitask environent. */
    // } while ((token == 0xFF) && Timer1);

```

修改 rcvr_datablock 函数。

```

    UINT i = MS_FACTOR;

    if (drv) return STA_NOINIT; /* Supports only drive 0 */
    init_spi(); /* Initialize SPI */

    if (Stat & STA_NODISK) return Stat; /* Is card existing in the socket? */

    FCLK_SLOW();
    for (n = 10; n; n--) xchg_spi(0xFF); /* Send 80 dummy clocks */

    ty = 0;
    if (send_cmd(CMD0, 0) == 1) { /* Put the card SPI/Idle state */
        Timer1 = 1000; /* Initialization timeout = 1 sec */
        if (send_cmd(CMD8, 0x1AA) == 1) { /* SDv2? */
            for (n = 0; n < 4; n++) ocr[n] = xchg_spi(0xFF); /* Get 32 bit return value of R7 resp */
            if (ocr[2] == 0x01 && ocr[3] == 0xAA) { /* Is the card supports vcc of 2.7-3.6V? */
                while (Timer1 && send_cmd(ACMD41, 1UL << 30))
                {
                    if(i>0)
                    {
                        i--;
                    }
                    else
                    {
                        i = MS_FACTOR;
                        Timer1--;
                    }
                } /* Wait for end of initialization with ACMD41(HCS) */
                if (Timer1 && send_cmd(CMD58, 0) == 0) { /* Check CCS bit in the OCR */
                    for (n = 0; n < 4; n++) ocr[n] = xchg_spi(0xFF);
                    ty = (ocr[0] & 0x40) ? CT_SD2 | CT_BLOCK : CT_SD2; /* Card id SDv2 */
                }
            }
        }
    }

```



```

    } else {
        /* Not SDv2 card */
        if (send_cmd(ACMD41, 0) <= 1) {
            /* SDv1 or MMC? */
            ty = CT_SD1; cmd = ACMD41; /* SDv1 (ACMD41(0)) */
        } else {
            ty = CT_MMC; cmd = CMD1; /* MMCv3 (CMD1(0)) */
        }
        while (Timer1 && send_cmd(cmd, 0))
        {
            if(i>0)
            {
                i--;
            }
            else
            {
                i = MS_FACTOR;
                Timer1--;
            }
        }
        /* Wait for end of initialization */
        if (!Timer1 || send_cmd(CMD16, 512) != 0) /* Set block length: 512 */
            ty = 0;
    }
}

```

修改 disk_initialize 函数

```

int wait_ready (
    /* 1:Ready, 0:Timeout */
    UINT wt
    /* Timeout [ms] */
)
{
    BYTE d;

    UINT i = MS_FACTOR;

    Timer2 = wt;
    do {
        d = xchg_spi(0xFF);
        /* This loop takes a time. Insert rot_rdq() here for multitask environlment. */
        if(i>0)
        {
            i--;
        }
        else
        {
            i = MS_FACTOR;
            Timer2--;
        }
    } while (d != 0xFF && Timer2); /* Wait for card goes ready or timeout */
}

```

修改 wait_ready 函数

```

//void disk_timerproc (void)
//{
//    WORD n;
//    BYTE s;
//
//    n = Timer1;
//    if (n) Timer1 = --n;
//    n = Timer2;
//    if (n) Timer2 = --n;
//
//    s = Stat;
//    if (MMC_WP) /* Write protected */
//        s |= STA_PROTECT;
//    else /* Write enabled */
//        s &= ~STA_PROTECT;
//    if (MMC_CD) /* Card is in socket */
//        s &= ~STA_NODISK;
//    else /* Socket empty */
//        s |= (STA_NODISK | STA_NOINIT);
//    Stat = s;
//}

```

注释 disk_timeproc 函数。

18. 修改 SPI 底层相关函数。

```

/* Exchange a byte */
static
BYTE xchg_spi (
    BYTE dat          /* Data to send */
)
{
    // SPIx_DR = dat;
    // while (SPIx_SR & _BV(7)) ;
    // return (BYTE)SPIx_DR;
    while(!SPI_IsSPTEF(SPI0));
    SPI_WriteDataReg(SPI0, dat);
    while(!SPI_IsSPRF(SPI0));
    return (BYTE)SPI_ReadDataReg(SPI0);
}

```

修改 xchg_spi 函数。

```

/* Receive multiple byte */
static
void rcvr_spi_multi (
    BYTE *buff,          /* Pointer to data buffer */
    UINT btr             /* Number of bytes to receive (even number) */
)
{
    // WORD d;
    //
    //
    // SPIx_CR1 &= ~_BV(6);
    // SPIx_CR1 |= (_BV(6) | _BV(11)); /* Set SPI to 16-bit mode */
    //
    // SPIx_DR = 0xFFFF;
    // btr -= 2;
    // do {
    //     while (SPIx_SR & _BV(7)) ;
    //     d = SPIx_DR;
    //     SPIx_DR = 0xFFFF;
    //     buff[1] = d; buff[0] = d >> 8;
    //     buff += 2;
    // } while (btr -= 2);
    // while (SPIx_SR & _BV(7)) ;
    // d = SPIx_DR;
    // buff[1] = d; buff[0] = d >> 8;
    //
    // SPIx_CR1 &= ~(_BV(6) | _BV(11)); /* Set SPI to 8-bit mode */
    // SPIx_CR1 |= _BV(6);
    while(btr--)
    {
        *buff = xchg_spi(0xFF);
        buff++;
    }
}

```

修改 rcvr_spi_multi 函数。

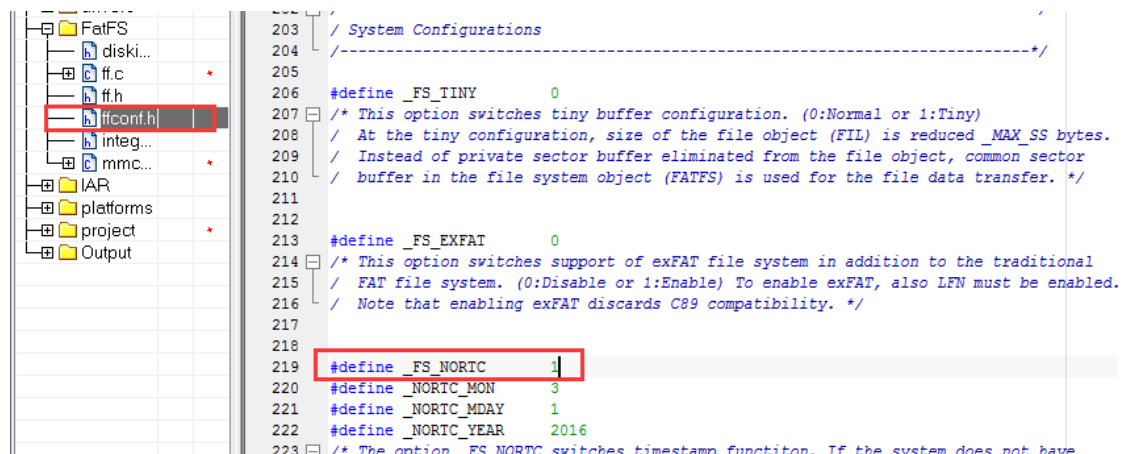
```

static
void xmit_spi_multi (
    const BYTE *buff,      /* Pointer to the data */
    UINT btx               /* Number of bytes to send (even number) */
)
{
    // WORD d;
    //
    // SPIx_CR1 &= ~_BV(6);
    // SPIx_CR1 |= (_BV(6) | _BV(11));      /* Set SPI to 16-bit mode */
    //
    // d = buff[0] << 8 | buff[1];
    // SPIx_DR = d;
    // buff += 2;
    // btx -= 2;
    // do {
    //     d = buff[0] << 8 | buff[1];
    //     while (SPIx_SR & _BV(7)) ;
    //     SPIx_DR;
    //     SPIx_DR = d;
    //     buff += 2;
    // } while (btx -= 2);
    // while (SPIx_SR & _BV(7)) ;
    // SPIx_DR;
    //
    // SPIx_CR1 &= ~(_BV(6) | _BV(11));      /* Set SPI to 8-bit mode */
    // SPIx_CR1 |= _BV(6);
    while(btx--)
    {
        xchg_spi(*buff);
        buff++;
    }
}

```

修改 xmit_spi_multi 函数。

19. 修改 ffconf.h 中的 NORTC 定义为 1



```

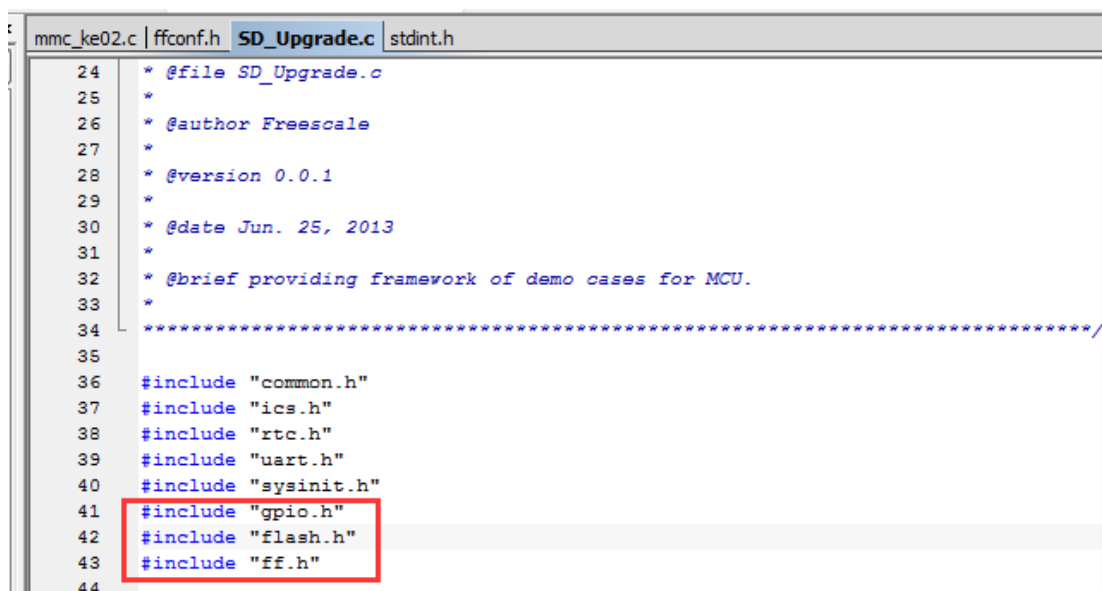
203 / System Configurations
204 /-----*/
205
206 #define _FS_TINY 0
207 /* This option switches tiny buffer configuration. (0:Normal or 1:Tiny)
208 / At the tiny configuration, size of the file object (FIL) is reduced _MAX_SS bytes.
209 / Instead of private sector buffer eliminated from the file object, common sector
210 / buffer in the file system object (FATFS) is used for the file data transfer. */
211
212
213 #define _FS_EXFAT 0
214 /* This option switches support of exFAT file system in addition to the traditional
215 / FAT file system. (0:Disable or 1:Enable) To enable exFAT, also LFN must be enabled.
216 / Note that enabling exFAT discards C89 compatibility. */
217
218
219 #define _FS_NORTC 1
220 #define _NORTC_MON 3
221 #define _NORTC_MDAY 1
222 #define _NORTC_YEAR 2016
223 /* The option _FS_NORTC switches timestamp function. If the system does not have

```

20. 此时编译就能通过了，FatFS 也就移植完成了，使用 FatFS 只用包含 ff.h 文件，就能调用其中的函数啦。

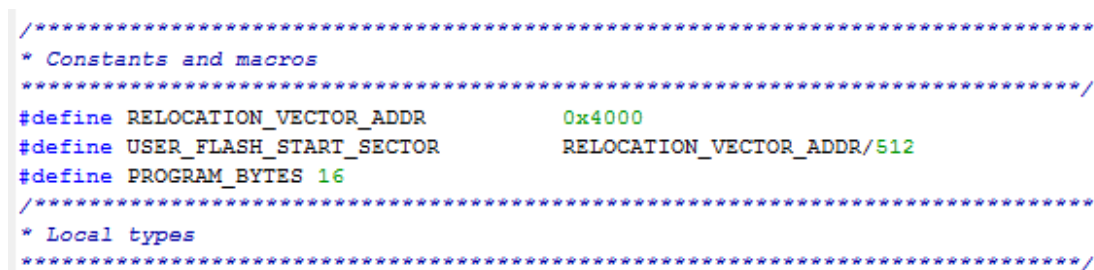
第二部分 SD 升级程序

1. 对主函数所在源文件进行修改。
2. 添加使用到的头文件 `gpio.h`, `flash.h`, `ff.h`。



```
mmc_ke02.c | ffconf.h | SD_Upgrade.c | stdint.h
24  * @file SD_Upgrade.c
25  *
26  * @author Freescale
27  *
28  * @version 0.0.1
29  *
30  * @date Jun. 25, 2013
31  *
32  * @brief providing framework of demo cases for MCU.
33  *
34  * *****/
35
36 #include "common.h"
37 #include "ics.h"
38 #include "rtc.h"
39 #include "uart.h"
40 #include "sysinit.h"
41 #include "gpio.h"
42 #include "flash.h"
43 #include "ff.h"
44
```

3. 添加三个宏定义:



```
/* *****/
/* Constants and macros
*****/
#define RELOCATION_VECTOR_ADDR      0x4000
#define USER_FLASH_START_SECTOR  RELOCATION_VECTOR_ADDR/512
#define PROGRAM_BYTES 16
/* *****/
/* Local types
*****/
```

`RELOCATION_VECTOR_ADDR`, 定义了用户程序向量表的起始地址, 这个必须超出 SD 卡升级 Demo 所占空间的尾地址, 此处定义为 `0x4000`, 即留出 16KB 给 SD 升级 Demo 程序, 实际上用不了这么多。下图为程序编译后, 实际占用大小情况。

```
13 544 bytes of readonly code memory
  48 bytes of readwrite code memory
  976 bytes of readonly data memory (+ 16 absolute)
  24 bytes of readwrite data memory
```

`USER_FLASH_START_SECTOR`, 定义了用户程序所在的起始 Sector, 由于使用了 KE02Z64VQH4 芯片, 可以在参考手册中查到, 每个 sector 为 512 Bytes, 如下图所示。

- 64 KB of flash memory composed of one 64 KB flash block divided into 128 sectors of 512 bytes

`PROGRAM_BYTES`, 定义了每次读取 bin 文件的大小, 同时也是烧写用户程序每次的大小。

此数值越大，则每次读取烧写时所需要的 buffer 也就越大。

4. 添加函数声明。

```
]/*****  
 * Local functions  
 *****/  
extern void delay_ms(uint32_t cnt);  
int main (void);  
int8_t is_application_ready_for_executing(uint32_t applicationAddress);  
void JumpToUserApplication(uint32_t userStartup);  
int8_t Update_User_Application(void);
```

5. 修改 main 函数

```
/* *****/  
int main (void)  
{  
    ICS_ConfigType sICSConfig;  
  
    /* Perform processor initialization */  
    sysinit();  
    /* switch clock mode from FEE to FEI */  
    sICSConfig.u32ClkFreq = 32; /* NOTE: use value 32 for 31.25KHz to 39.0625KHz of internal IRC */  
    ICS_SwitchMode(FEE,FEI, sICSConfig);  
  
    delay_ms(1000);  
    GPIO_Init(GPIOA, GPIO_PTD2_MASK, GPIO_PinInput); //PTD2 nCD Card Detection  
  
    //Detect no SD card  
    if(GPIO_Read(GPIOA)&GPIO_PTD2_MASK)  
    {  
        //User Application is Valid  
        if(is_application_ready_for_executing(RELOCATION_VECTOR_ADDR))  
        {  
            SCB->VTOR = RELOCATION_VECTOR_ADDR;  
            JumpToUserApplication(RELOCATION_VECTOR_ADDR);  
        }  
        else  
        {  
            while(1)  
            {}  
        }  
    }  
  
    if(Update_User_Application() !=0 )  
    {  
        //printf("Update Failed!\n");  
    }  
}
```

上电时会延时 1s，等待 CD 信号稳定，然后通过 PTD2 管脚检测 SD 卡座的 CD 信号来判断是否有 SD 卡插入。

如果没有 SD 卡插入，则首先通过 is_application_ready_for_executing 函数判断应用程序地址是否存在有效程序，有的话就直接跳转过去运行。否则进入 while(1)死循环。

如果有 SD 卡插入，则通过 Update_User_Application 函数进行用户程序升级。

6. Is_application_ready_for_executing 函数实现。这里只对用户程序首地址是否为全 0xFF 进行了简单的判断，实际使用中，可根据需要添加代码。

```

int8_t is_application_ready_for_executing(uint32_t applicationAddress)
{
    if ((* (uint32_t*) applicationAddress) != 0xFFFFFFFF)
    {
        return 1;
    }

    return 0;
}

```

7. JumpToUserApplication 跳转函数的实现

```

void JumpToUserApplication(uint32_t userStartup)
{
    /* set up stack pointer */
    asm("LDR    r1, [r0]");
    asm("mov    r13, r1");
    /* jump to application reset vector */
    asm("ADDS    r0, r0, #0x04 ");
    asm("LDR    r0, [r0]");
    asm("BX     r0");
}

```

8. Bin 文件烧写函数 Update_File_Process 实现。函数会首先根据 bin 文件大小来擦除占用的 Sectors。之后在 for 循环里，根据 PROGRAM_BYTES 指定的字节数，来读取文件，并烧写到 Flash 中。

```

int8_t Update_File_Process(FRESULT fr, FIL fil)
{
    uint8_t Erase_Sector_Cnt;
    uint32_t file_size;
    uint32_t i;
    uint8_t read_buf[PROGRAM_BYTES];
    UINT bw;

    FLASH_Init(BUS_CLK_HZ);
    file_size = f_size(&fil);
    Erase_Sector_Cnt=file_size/512+1;
    for(i=0;i<Erase_Sector_Cnt;i++)
    {
        FLASH_EraseSector((USER_FLASH_START_SECTOR+i)*FLASH_SECTOR_SIZE);
    }

    for(i=0;;)
    {
        fr = f_read(&fil, read_buf, PROGRAM_BYTES, &bw);

        if(fr)
        {
            //printf("\nError reading file\r\n");
            return -1;
        }
        else
        {
            if(bw == PROGRAM_BYTES)
            {
                FLASH_Program( USER_FLASH_START_SECTOR*FLASH_SECTOR_SIZE+i,read_buf,PROGRAM_BYTES );
                i+=PROGRAM_BYTES;
            }
            else
            {
                FLASH_Program( USER_FLASH_START_SECTOR*FLASH_SECTOR_SIZE+i,read_buf,bw );
                return 0;
            }
        }
    }
}

```

9. Update_User_Application 函数实现。此函数会打开 SD 卡中的 update.bin 文件，如果文件存在，则调用 Update_File_Process 函数进行文件烧写。并在烧写完成后，通过 JumpToUserApplication 函数跳转到用户程序运行。

```

int8_t Update_User_Application(void)
{
    FATFS fs;
    FRESULT fr;
    FIL      fil;

    fr= f_mount(&fs,"",0);
    if(fr)
    {
        //printf("\nError mounting file system\r\n");
        return -1;
    }

    fr = f_open(&fil, "update.bin", FA_OPEN_EXISTING|FA_READ);
    if(fr)
    {
        //printf("\nError opening text file\r\n");
    }
    else
    {
        if(Update_File_Process(fr,fil)!= 0)
        {
            //printf("Update File Process failed\r\n");
            return -1;
        }
    }

    fr = f_close(&fil);

    //printf("Jump to User App!\r\n");
    SCB->VTOR = RELOCATION_VECTOR_ADDR;
    JumpToUserApplication(RELOCATION_VECTOR_ADDR);

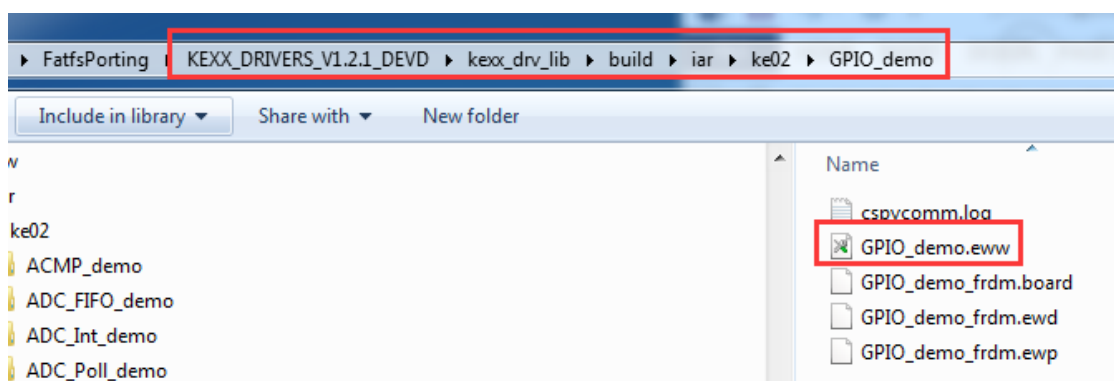
    return 0;
}

```

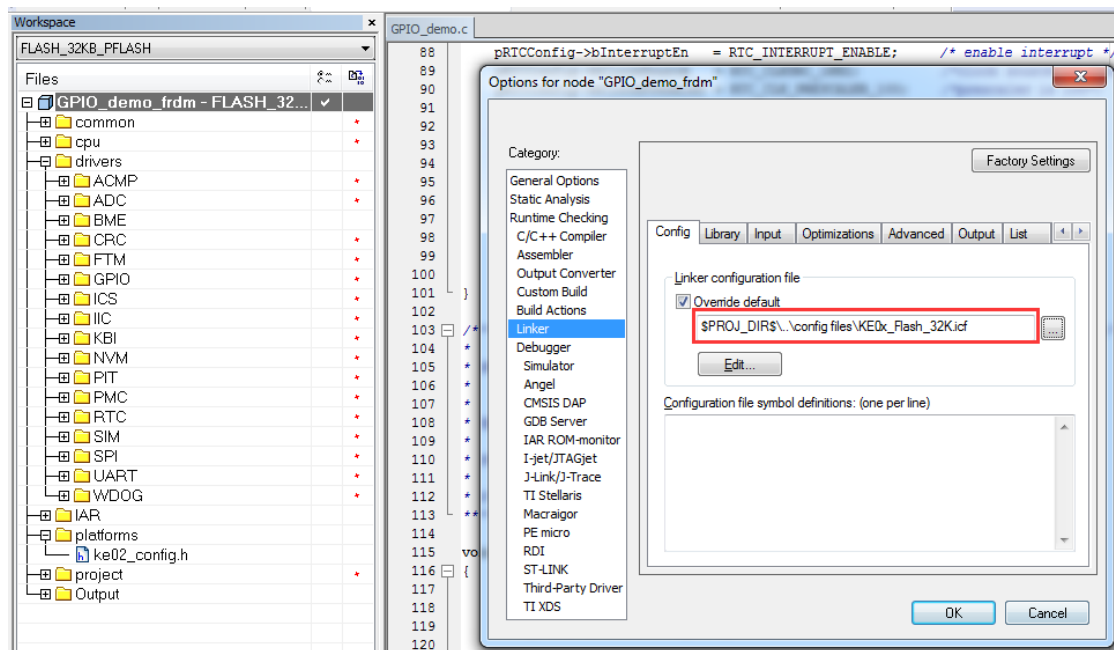
10. 修改完成后，编译整个工程，并将程序下载到单片机中。

第三部分 用户程序

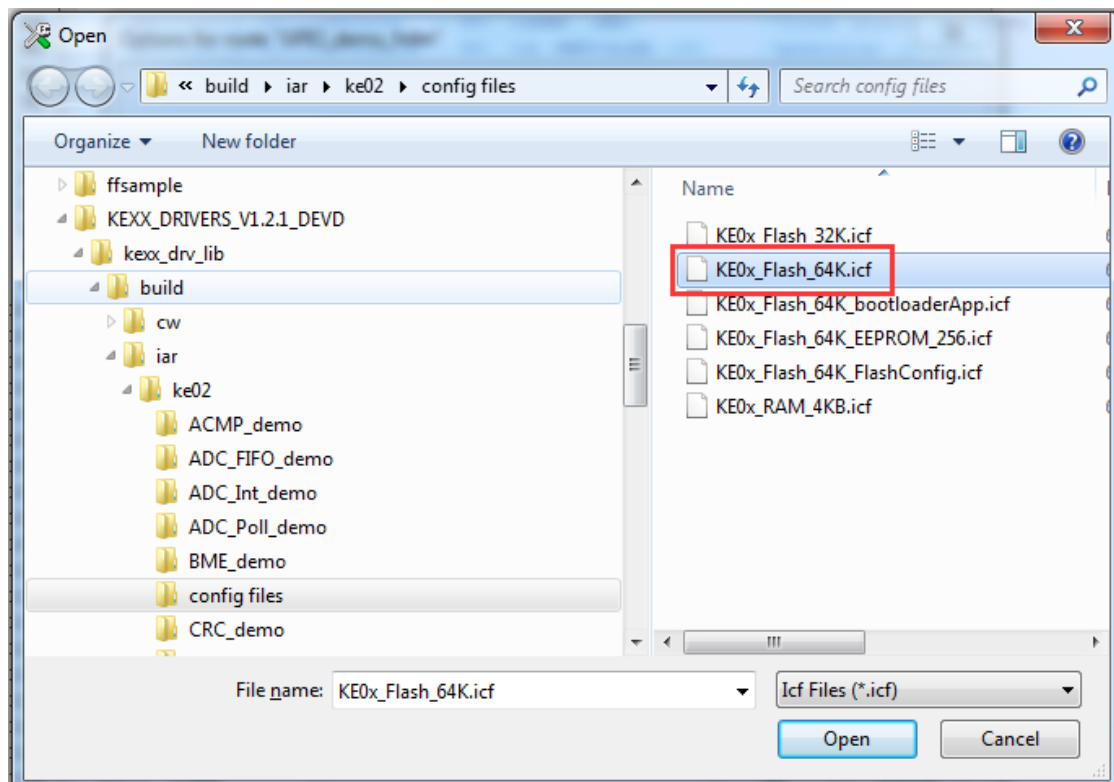
1. 用户程序我们直接使用 KE 驱动库里的 GPIO example 为例。



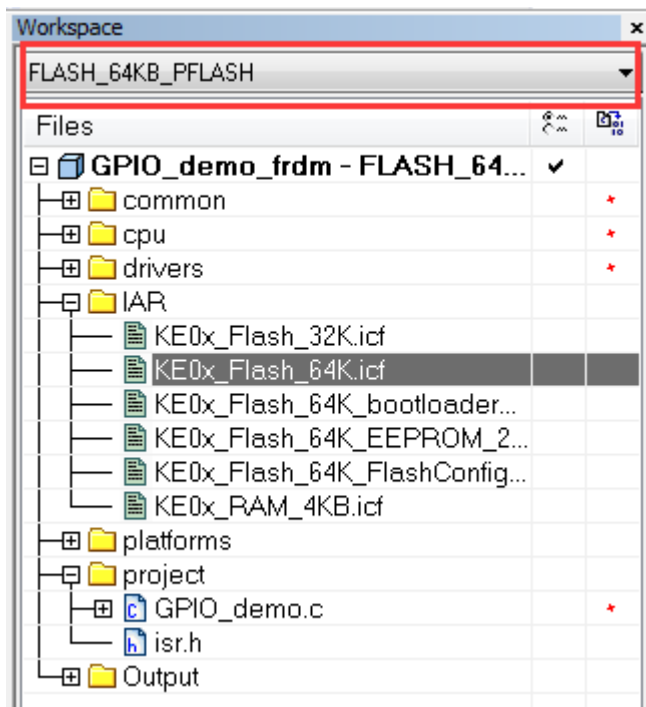
2. 打开 GPIO 工程，并在 Option 中找到链接文件，我们需要对链接文件进行修改。



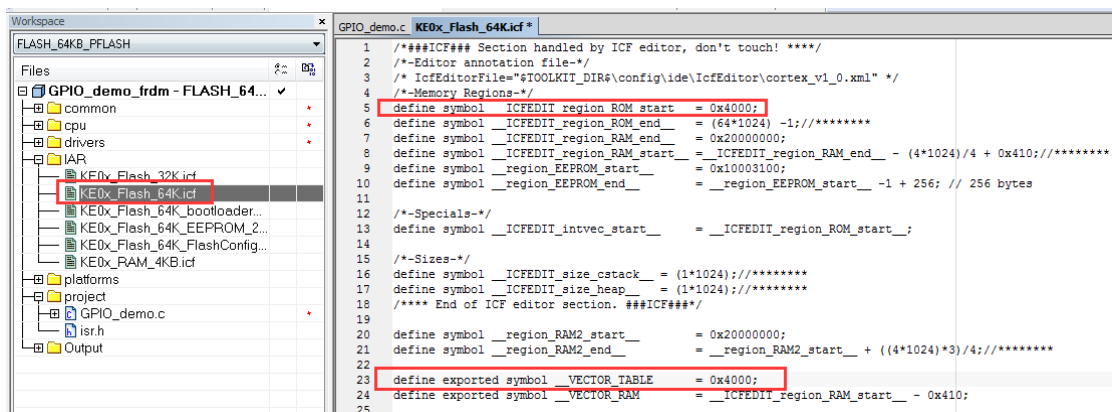
3. 选择 64K 的 icf 文件。



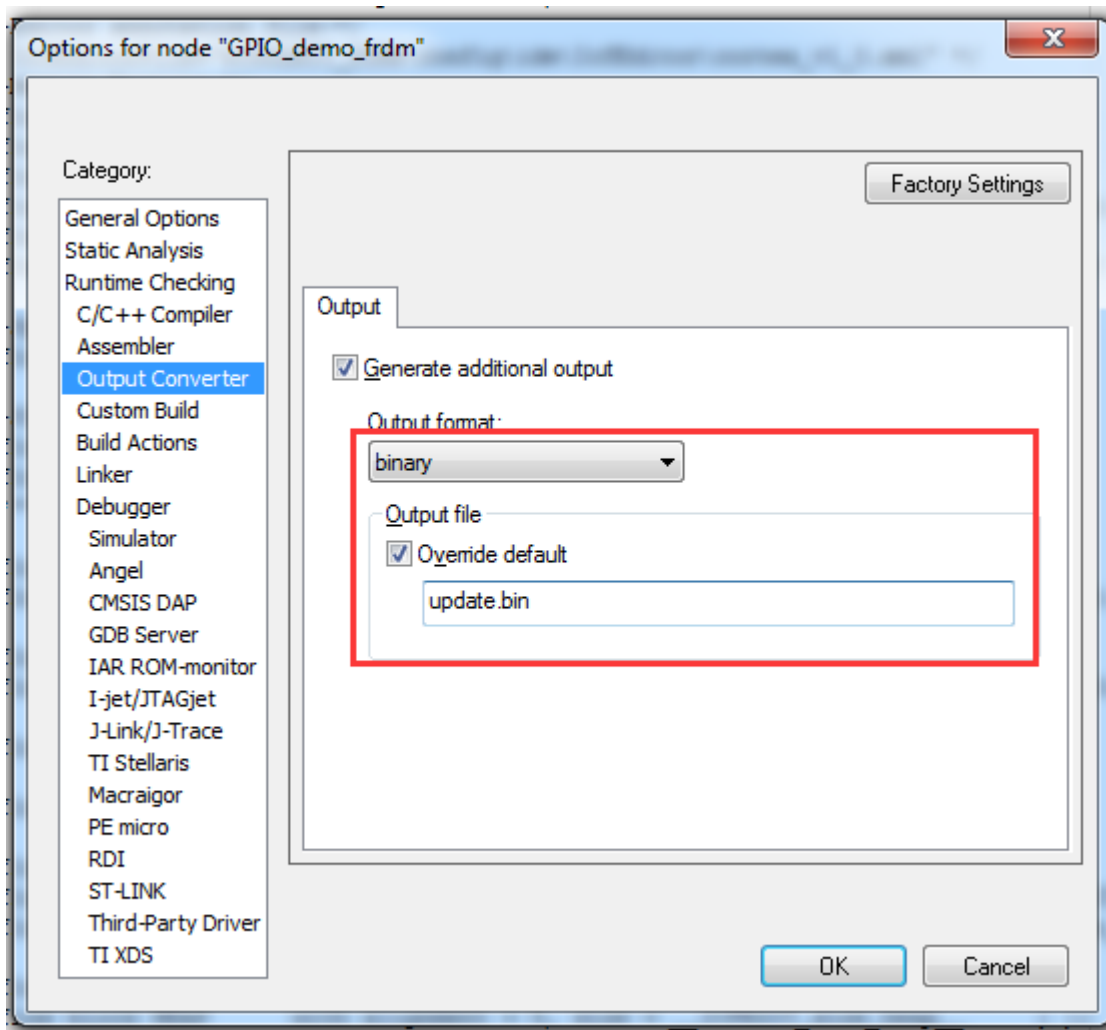
4. Workspace 中选择 64KB 的版本。



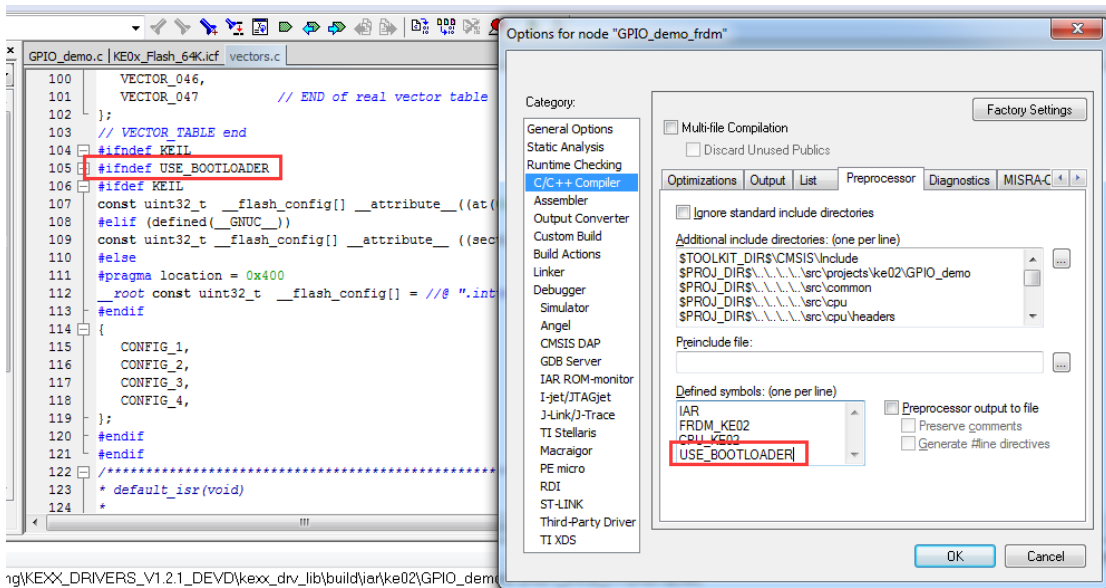
5. 选择 64K 的链接文件，并将 ROM 起始地址和向量表地址修改为 0x4000，和 SD 卡升级程序定义的宏 RELOCATION_VECTOR_ADDR 保持一致。



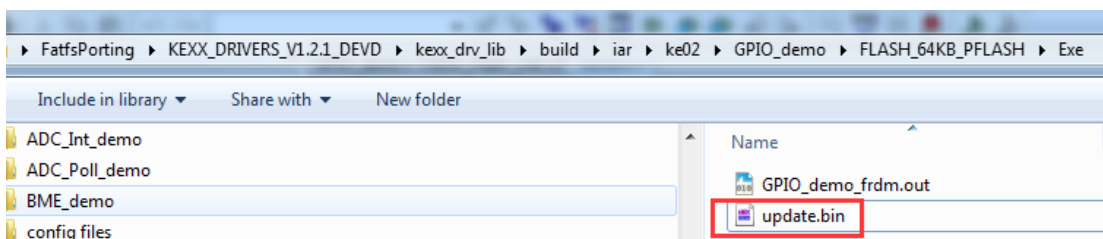
6. 在工程 option 中将输出格式改为 bin 文件，并命名为 update.bin



7. 在预处理中，添加 `USE_BOOTLOADER` 的定义，从而去掉 `flash_config` 指定地址的定义部分。

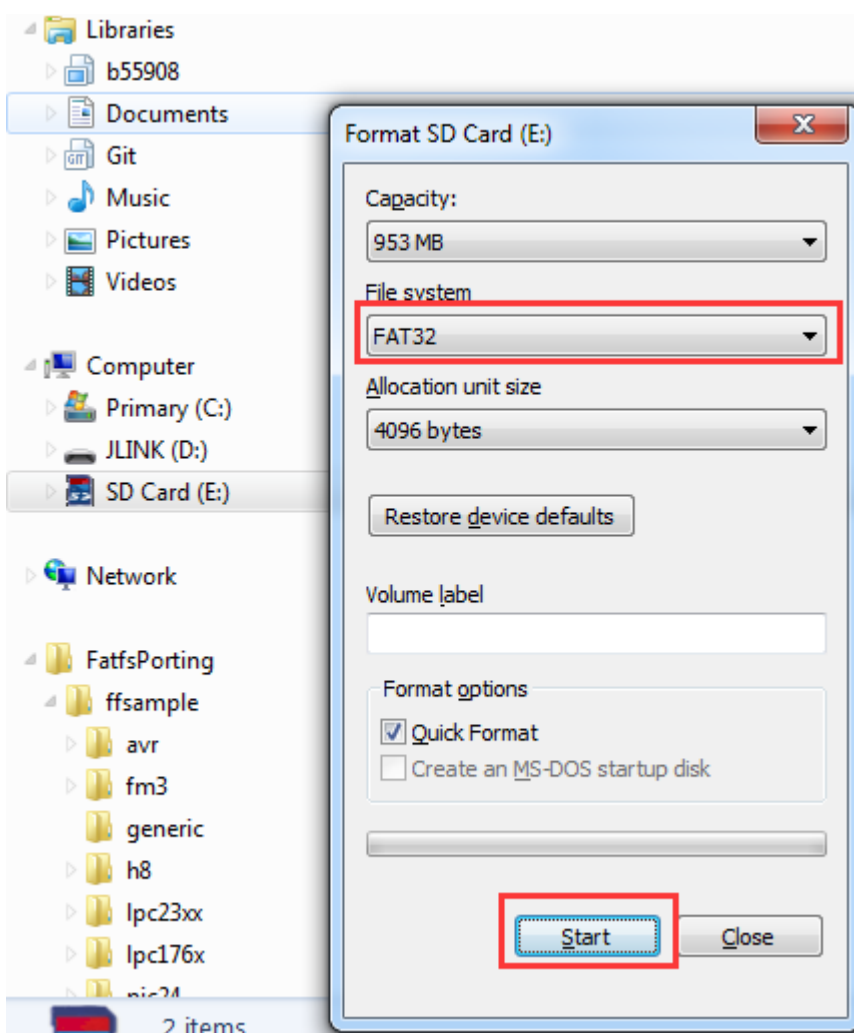


8. 重新编译工程，就能在如下目录中找到生成的用户程序的 `update.bin` 文件啦。

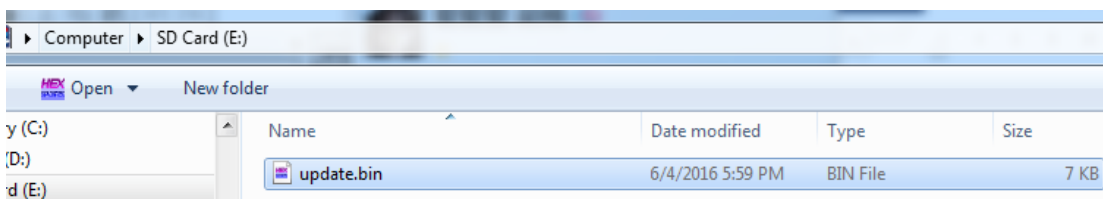


第四部分 SD 卡

1. 将 SD 卡插入电脑，首次使用需要将 SD 卡格式化为 FAT32 格式。



2. 完成后，将上一部分生成的 update.bin 文件，拷贝到 SD 卡中。



3. 接下来就可以使用 SD 卡，将 GPIO 的点灯程序下载到单片机里啦！