应用笔记

如何在 KF32A 系列中
实现 Flash driver 升级流程

前言

KF32A 系列提供两种代码运行方式，第一种方式，flash 区域运行，实现的方式是直接从 flash 读取指令执行；第二种方式，编译器指定 RAM 运行，实现的方式是执行数据还是烧录到 flash，但是启动时把 flash 指定 RAM 运行部分的数据 copy 到 RAM，之后直接从 RAM 区域读取指令执行。

RAM 运行相对于 flash 的优点在于运行速度更快，缺点在于，在 RAM 区域运行，就意味着需要占用一部分的 RAM 空间，并且 flash 占用的空间不会减少。

本应用笔记将以 KF32A156MQV 为例介绍如何把指定的函数放到 RAM 空间执行，并以长安 BCM 为例，介绍通过 RAM 运行实现 flash driver boot loader 的升级方式。

本应用笔记使用的 KF32 IDE 与 KF32A156 外设固件库及代码例程可以从 ChipON 官方网站 www.chipon-ic.com 下载。

目录

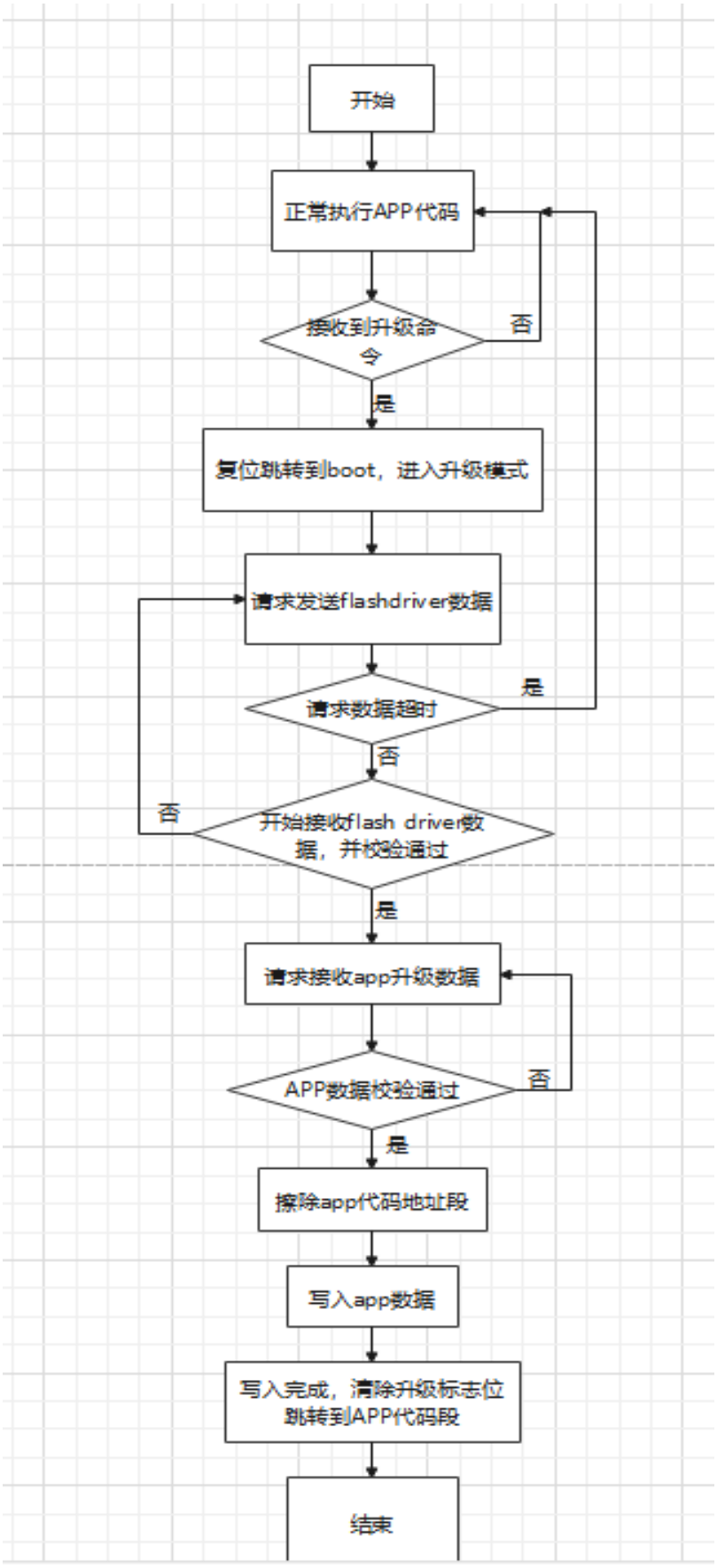
1 Flash driver boot loader 的目的和意义

KF32A156MQV 在正常运行时，尽管我们代码段里面储存有擦写 flash 的代码，也不会出现 flash 被擦除，程序被更改的情况。但是在特殊情况下，如静电干扰，软件 bug 导致代码跑飞等，这种情况下，如果 PC 指针执行到了 flash 的擦除部分，很有可能导致 flash 里的代码段被损坏，导致机器运行失败，无法再次正常使用。在这种情况下，为了避免代码段的擦写 flash 指令被误执行，就需要引进一种特殊的 boot loader 方式—flash driver boot loader。

2 Flash driver boot loader 实现流程

为了使代码段里面不存在擦写 flash 的指令，我们需要把擦除指令在代码里面去掉，但是升级的时候，我们又务必会用到擦写 flash 的指令，这个时候，我们只能在进 boot 后，如果需要执行升级命令，我们需要先接收到擦写 flash 的指令数据，这个数据可以存放在上一级处理文件的系统里，如 PC，，后台服务器，安卓、Linux 系统等，通过特定的通信方式以及安全的通信协议传输到我们 mcu 后，在把这部分数据放到 RAM 区域，需要执行时，我们利用函数指针执行这部分指令，来完成我们升级时需要的擦写指令。由于数据是存放到 RAM 区，升级完成后复位芯片就可以直接清除掉擦写 flash 的指令，后面即使程序出现 bug，也能确保不会执行修改 flash 的指令。

执行软件流程图：

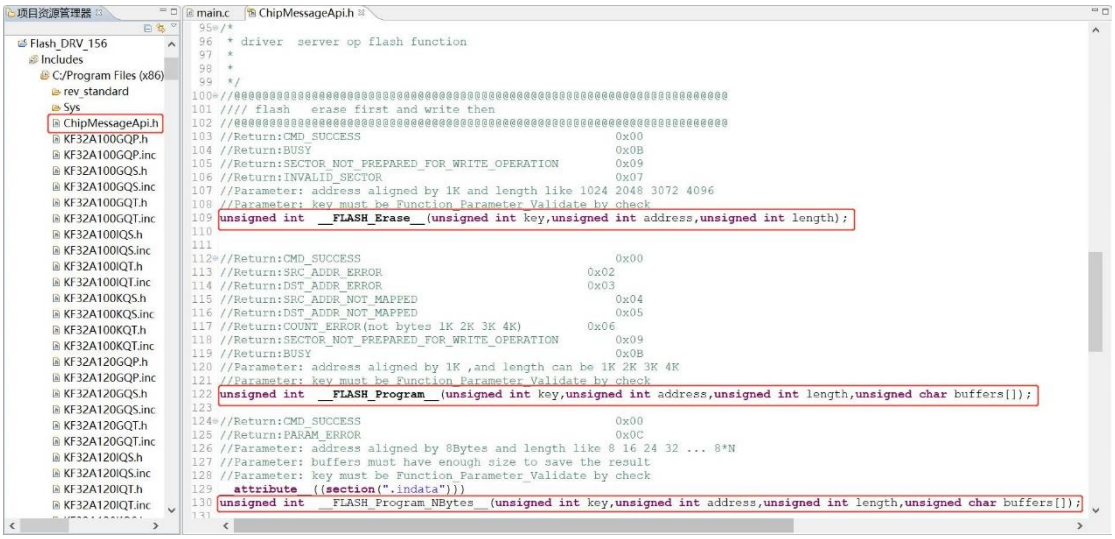

开始
正常执行APP代码
接收到升级命令
否
是
复位跳转到boot，进入升级模式
请求发送flashdriver数据
请求数据超时
是
否
否
开始接收flash driver数据，并校验通过
是
请求接收app升级数据
APP数据校验通过
否
是
擦除app代码地址段
写入app数据
写入完成，清除升级标志位跳转到APP代码段
结束

3 利用 IDE 获取编译后独立的 flash driver 数据（二进制）

为了使代码段里面没有擦写 flash 的数据，我们需要单独把擦写数据提取出来，通过外部工具发送到程序里面。提取方法如下：

第一步：找到 FLASH 擦写读目标函数

需要注意，提取的函数里面不能再调用其他函数，只能是独立语句，宏定义的函数也不可以。原因是调用的函数第二次编译后地址会有变化，会导致函数指针跳转到异常的地址，导致程序跑飞。与 KF32A150/1/2 不同的是，KF32A156 的操作 Flash 的函数是封装在函数库里的。在 ChipMessageApi.h 头文件里有对应函数接口的具体定义，打开方式可参考以下截图：



在 ChipMessageApi.h 文件里 FLASH 擦除只有一个函数接口定义，而写和读有多个。我们这里使用和提取擦、写、读函数和注释分别如下：

```
/*
Return:CMD_SUCCESS                                  0x00
Return:BUSY                                         0x0B
Return:SECTOR_NOT_PREPARED_FOR_WRITE_OPERATION      0x09
Return:INVALID_SECTOR                               0x07
Parameter: address aligned by 1K and length like 1024 2048 3072 4096
Parameter: key must be Function_Parameter_Validate by check  */
unsigned int __FLASH_Erase__(unsigned int key, unsigned int address,
unsigned int length);


/*
```

```
Return:CMD_SUCCESS                                    0x00
Return:PARAM_ERROR                                    0x0C
Parameter: address aligned by 8Bytes and length like 8 16 24 32 ...
8*N
Parameter: buffers must have enough size to save the result
Parameter: key must be Function_Parameter_Validate by check */
__attribute__((section(".indata")))
unsigned int  __FLASH_Program_NBytes__(unsigned int key, unsigned int
address, unsigned int length, unsigned char buffers[]);

/*
Return:CMD_SUCCESS                                    0x00
Parameter: buffers must have enough size to save the result*/
unsigned int  __FLASH_Read__(unsigned int address, unsigned int
length, unsigned char buffers[]);
```

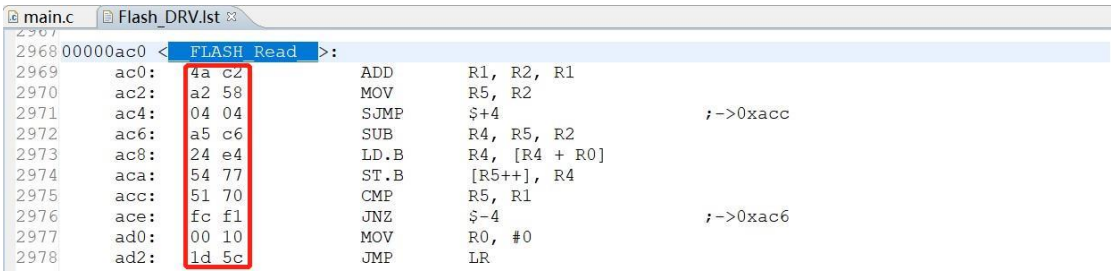
需要注意，上面的Flash擦除和写入函数接口的第一个形参定义的key指的是校验秘钥，其定义也在ChipMessageApi.h头文件里，具体如下：

```
// erase write function need key, no support modify
#define Function_Parameter_Validate   0x5A5A6688
```

第二步：编译生成 FLASH 擦写读函数数据（二进制）并提取

由于上面提到的 FLASH 擦写读函数都是库函数且定义也在库函数头文件里，所以编译之前并不需要修改.ld 文件。需要注意，在编译前必须在面函数里使用到 FLASH 擦读写函数，否则在编译时会被优化掉；

编译完成后，可在 lst 文件中找到 FLASH 擦读写对应的数据。然后需要把.lst 中对应的数据通过脚本或者手动提取出来。注意，只提取数据部分，汇编指令不需要，如下图红框部分：



main.c Flash_DRV.lst

```
00000a28 <  FLASH_Erase  >:
     a28:  2d 5d          PUSH    LR
     a2a:  e8 31          SUB     SP, #24
     a2c:  54 13          MOV     R5, #52
     a2e:  e5 85          ST.W    [SP], R5
     a30:  4a 10          MOV     R4, #10
     a32:  29 dd          LSR     R5, R1, R4
     a34:  01 25          ST.W    [SP + #1], R5
     a36:  13 45          LD      R5, [PC + #19]          ;->0xa80  :=0x3ff
     a38:  49 d5          ANL     R1, R1, R5
     a3a:  52 c2          ADD     R2, R2, R1
     a3c:  03 22          ST.W    [SP + #3], R2
     a3e:  03 0c          LD.W    R4, [SP + #3]
     a40:  64 d5          ANL     R4, R4, R5
     a42:  09 f0          JZ      $+9                     ;->0xa54
     a44:  03 0a          LD.W    R2, [SP + #3]
     a46:  03 0b          LD.W    R3, [SP + #3]
     a48:  41 10          MOV     R4, #1
     a4a:  a2 7a          LSL     R4, #10
     a4c:  22 c3          ADD     R4, R2, R4
     a4e:  6b d5          ANL     R5, R3, R5
     a50:  6c c7          SUB     R5, R4, R5
     a52:  03 25          ST.W    [SP + #3], R5
     a54:  03 0d          LD.W    R5, [SP + #3]
     a56:  6d cc          SUB     R5, R5, #1
     a58:  03 25          ST.W    [SP + #3], R5
     a5a:  01 0c          LD.W    R4, [SP + #1]
     a5c:  03 0d          LD.W    R5, [SP + #3]
     a5e:  aa 7c          LSR     R5, #10
     a60:  2d c3          ADD     R5, R5, R4
     a62:  02 25          ST.W    [SP + #2], R5
     a64:  57 10          MOV     R5, #7
     a66:  08 44          LD      R4, [PC + #8]           ;->0xa84  :=0x5a5a6688
     a68:  04 70          CMP     R0, R4
     a6a:  06 f1          JNZ     $+6                     ;->0xa76
     a6c:  0e 58          MOV     R0, SP
```

main.c Flash_DRV.lst

```
10003214 <  FLASH_Program_NBytes_RAM  >:
10003214:  0f 5f          PUSH    {R6-R9}
10003216:  c0 58          MOV     R6, R0
10003218:  22 45          LD      R5, [PC + #34]          ;->0x100032a0  :=0x40200100
1000321a:  23 44          LD      R4, [PC + #35]          ;->0x100032a4  :=0x50af0007
1000321c:  54 85          ST.W    [R5], R4
1000321e:  0c 10          MOV     R0, #12
10003220:  22 44          LD      R4, [PC + #34]          ;->0x100032a8  :=0x7ffff
10003222:  14 70          CMP     R1, R4
10003224:  3b f8          JHI     $+59                    ;->0x1000329a
10003226:  47 10          MOV     R4, #7
10003228:  39 d5          ANL     R7, R1, R4
1000322a:  38 f1          JNZ     $+56                    ;->0x1000329a
1000322c:  20 47          LD      R7, [PC + #32]          ;->0x100032ac  :=0x5a5a6688
1000322e:  67 70          CMP     R6, R7
10003230:  35 f1          JNZ     $+53                    ;->0x1000329a
10003232:  20 40          LD      R0, [PC + #32]          ;->0x100032b0  :=0x27654330
10003234:  a8 a2          ST.W    [R5 + #10], R0
10003236:  20 40          LD      R0, [PC + #32]          ;->0x100032b4  :=0x81040304
10003238:  a8 a2          ST.W    [R5 + #10], R0
1000323a:  20 40          LD      R0, [PC + #32]          ;->0x100032b8  :=0x7896dcde
1000323c:  e8 a2          ST.W    [R5 + #11], R0
1000323e:  20 40          LD      R0, [PC + #32]          ;->0x100032bc  :=0x92151407
10003240:  e8 a2          ST.W    [R5 + #11], R0
10003242:  02 d5          ANL     R0, R2, R4
10003244:  85 58          MOV     R4, R5
10003246:  00 38          CMP     R0, #0
10003248:  03 f1          JNZ     $+3                     ;->0x1000324e
1000324a:  27 38          CMP     R2, #7
1000324c:  05 f8          JHI     $+5                     ;->0x10003256
1000324e:  16 45          LD      R5, [PC + #22]          ;->0x100032a4  :=0x50af0007
10003250:  45 85          ST.W    [R4], R5
10003252:  0c 10          MOV     R0, #12
10003254:  23 04          SJMP    $+35                    ;->0x1000329a
10003256:  2b 4c          SET     [R5], #3
10003258:  85 98          LD.W    R0, [R5 + #2]
```

截图未截完，需要提取完整的函数下的数据!

第三步：调用前对工程的配置

为了更清晰的验证提取的数据是否有效，我们新建了一个工程来具体实现。

先要修改.ld 文件，指定擦写函数运行的 RAM 地址，如图：

main.c KF32A156MQV.ld

```
/*****************************************************************/
  .data :
  {
    . = (. + 3) & (-4);
  __data_start__ = .;

    KEEP (*(.ramvector*))       /* default server for vector writed in ram */

    . = 0x0000 ;
    KEEP (*(.userFdvdata*))     /* user defined RAM data start adress*/

  __lpdata_start__ = (. + 3) & (-4);
    *(.lpdata*)          /* Reset but Keep Space  */
    /* . = 0x4000 ;  */         /* Max length limit is 16K byte ,if need 16K all config ,delete com
  __lpdata_end__ =  (. + 3) & (-4);

    . = (. + 3) & (-4);
    *(.indata*)          /* server space for ram function */
    *(.inrdata*)
    *(.inrodata*)

    . = (. + 3) & (-4);
    *(.data*)            /* global variable with  initialization */
    . = (. + 3) & (-4);
  __data_end__ = .;

    . = (. + 3) & (-4);
  __bss_start__ = .;        /* global variable with no initialization */
    *(COMMON*)
    *(.bss*)
    . = (. + 3) & (-4);
  __bss_end__ = .;

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/     /* heap and stack space*/
  __allot_end__ = (. + 3) & (-4);
    Heap_Start      =     allot_end   ;
```

然后在 main 函数里把指定 FLASH_WritePage 的地址修饰用来修饰定义一个数组，数组的长度等于刚刚提取 FLASH_WritePage 数据的长度。如图所示：

```
/*定义数组用于存放FLASH擦除函数二进制数据，数组大小必须与实际二级制数据长度相等*/
uint8_t __attribute__((section(".userFdvdata")))
userFdvErasearray[100]={};

/*定义数组用于存放FLASH写入函数二进制数据，数组大小必须与实际二级制数据长度相等*/
uint8_t __attribute__((section(".userFdvdata")))
userFdvProgramarray[180]={};

/*定义数组用于存放FLASH读取函数二进制数据，数组大小必须与实际二级制数据长度相等*/
uint8_t __attribute__((section(".userFdvdata")))
userFdvReadarray[20]={};
```

接着从上位机把提取的数据发送给 mcu，让 mcu 依次把数据写到定义的数组里即可。

（注：这个时候如果我们没有上位机想做测试，可以直接把提取的数据赋值到数组里）。

第四步：调用生成的数据（二进制）实现 FLASH 擦读写功能

定义一个函数指针；指向第三步定义的数组首地址。后面在需要时直接调用对应的函数，就可以实现 flash 的擦写读功能了。

```
/**
    Return:CMD_SUCCESS                                  0x00
    Return:BUSY                                         0x0B
    Return:SECTOR_NOT_PREPARED_FOR_WRITE_OPERATION      0x09
    Return:INVALID_SECTOR                               0x07
    Parameter: address aligned by 1K and length like 1024 2048 3072 4096
    Parameter: key must be Function_Parameter_Validate by check
 */
typedef void (*pFdvEraseFunction) (unsigned int key,unsigned int address,unsigned int length);
pFdvEraseFunction Jump_FlashDv_erase;//定义FLASH擦除函数指针

/**
    Return:CMD_SUCCESS                                  0x00
    Return:PARAM_ERROR                                  0x0C
    Parameter: address aligned by 8Bytes and length like 8 16 24 32 ... 8*N
    Parameter: buffers must have enough size to save the result
    Parameter: key must be Function_Parameter_Validate by check
 */
typedef void (*pFdvProgramFunction) (unsigned int key,unsigned int address,unsigned int length,unsigned char buffers[]);
pFdvProgramFunction Jump_FlashDv_Program;//定义FLASH写入函数指针

/**
    Return:CMD_SUCCESS                                  0x00
    Parameter: buffers must have enough size to save the result
 */
typedef void (*pFdvReadFunction)(unsigned int address,unsigned int length,unsigned char buffers[]);
pFdvReadFunction Jump_FlashDv_Read;//定义FLASH读取函数指针

int main()
{

    //系统时钟120M,外设高频时钟16M
    SystemInit(120);//系统时钟初始化
    systick_delay_init(120);

    /*对FLASH数据数组附初值，验证功能使用*/
    for(volatile uint16_t m=0; m<256; m++)
    {
        flash_program_data[m]=m;
    }


    Jump_FlashDv_erase = (pFdvEraseFunction)userFdvErasearray;       //函数指针指向对应数组的首地址
    Jump_FlashDv_Program = (pFdvProgramFunction)userFdvProgramarray;//函数指针指向对应数组的首地址
    Jump_FlashDv_Read = (pFdvReadFunction)userFdvReadarray;          //函数指针指向对应数组的首地址
    delay_ms(1000);
    Jump_FlashDv_erase(Function_Parameter_Validate,ADDRESS_FLASH_DRIVER,1024);//调用FLASH擦除函数
    Jump_FlashDv_Program(Function_Parameter_Validate,ADDRESS_FLASH_DRIVER,1024,flash_program_data);//调用FLASH写入函数
    Jump_FlashDv_Read(ADDRESS_FLASH_DRIVER,1024,flash_read_data);//调用FLASH读取函数

    while(1)
    {

    }

}
```

切记，一定要先把数据放到 ram 完成后，才能执行 Jump_FLASH_WritePage 这个函数指针!

4 版本历史

文档版本历史

日期	版本	变更
2022 年 5 月 6 日	1	初始版本。