FEE使用介绍

1. FEE数据结构

fee里面最大的单位是Cluster GROUP（一般使用就一组Cluster GROUP就可以），每个GROUP会包含多个Cluster，Cluster里面又包含了多个Block。Block可以认为是存储数据的最小单元。按照需求用户可以自己定义cluster和block的数量、大小。其中一个Cluster的数据组成由下图所示：

4 bytes	4 bytes	4 bytes	4 bytes	Description
ClusterID	ClusterSize	Startaddress	Checksum	Cluster Header
unused		ClusterValidFlag		Cluster Status
BlockID	BlockLength	BlockDataAddress	~(BlockID) ~(BlockLength) ~(BlockDataAddress)	Block Header 1
BlockInvalidFlag		BlockValidFlag		Block Status 1
BlockID	BlockLength	BlockDataAddress	~(BlockID) ~(BlockLength) ~(BlockDataAddress)	Block Header 2
BlockInvalidFlag		BlockValidFlag		Block Status 2
BlockID	BlockLength	BlockDataAddress	~(BlockID) ~(BlockLength) ~(BlockDataAddress)	Block Header 3
BlockInvalidFlag		BlockValidFlag		Block Status 3
……				
……				
……				
BlockID	BlockLength	BlockDataAddress	~(BlockID) ~(BlockLength) ~(BlockDataAddress)	Block Header n
BlockInvalidFlag		BlockValidFlag		Block Status n
BlockHeaderBottom				16 byte
BlockHeaderBottom				16 byte
BLOCK DATA n				Block Data n
BLOCK DATA n				Block Data n
BLOCK DATA n				Block Data n
BLOCK DATA n				Block Data n
BLOCK DATA n-1				Block Data n-1
BLOCK DATA n-1				Block Data n-1
……				
……				
BLOCK DATA 1				Block Data 1
BLOCK DATA 1				Block Data 1

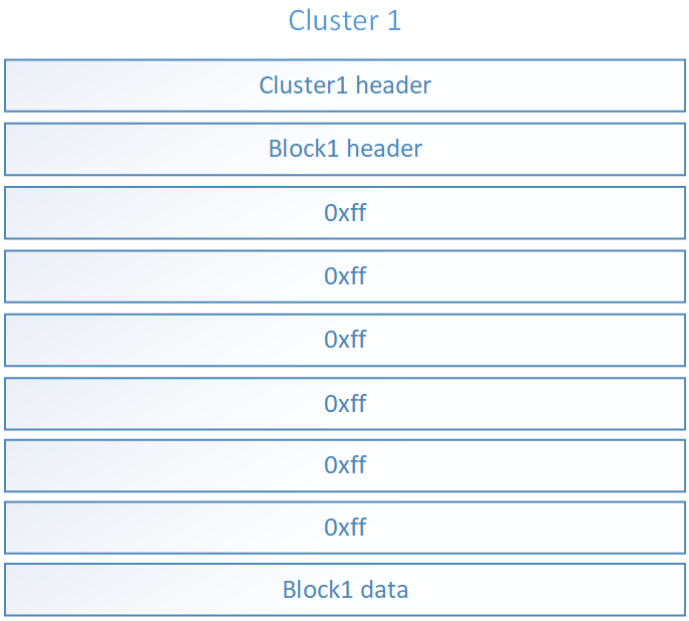
- Blockinvalidflag为uint64: 0x55, 0x55, 0x55, 0x55, 0xAA, 0xAA, 0xAA, 0xAA,
  Blockvalidflag 和 Clustervalidflag为uint64: 0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,

数据的存储原理

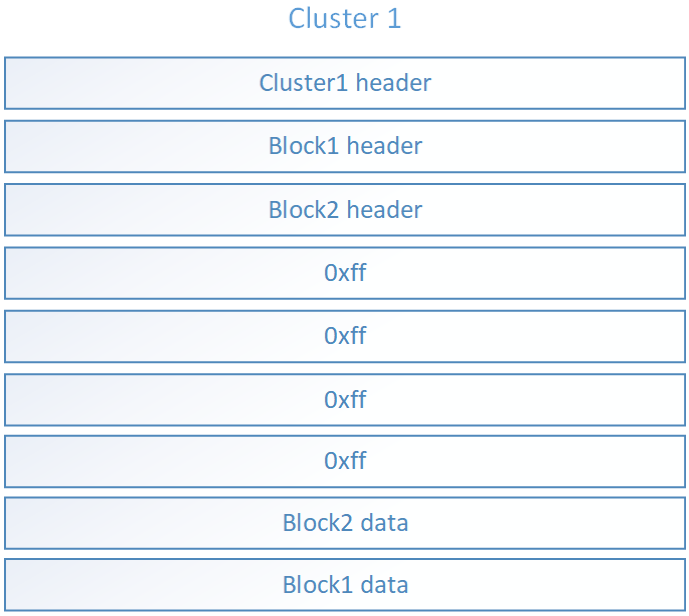
我想先描述一下FEE是怎么存储数据的，让读者有个大概的印象，再去细讲Cluster、Block都存了什么数据。怎么去解读上图的模型。

如下图所示，这是一个全新的cluster，一个全新的芯片在第一次上电运行Fee初始化后，FEE会建立第一个ID为1的Cluster，此时Cluster中并没有写入blcok的数据。

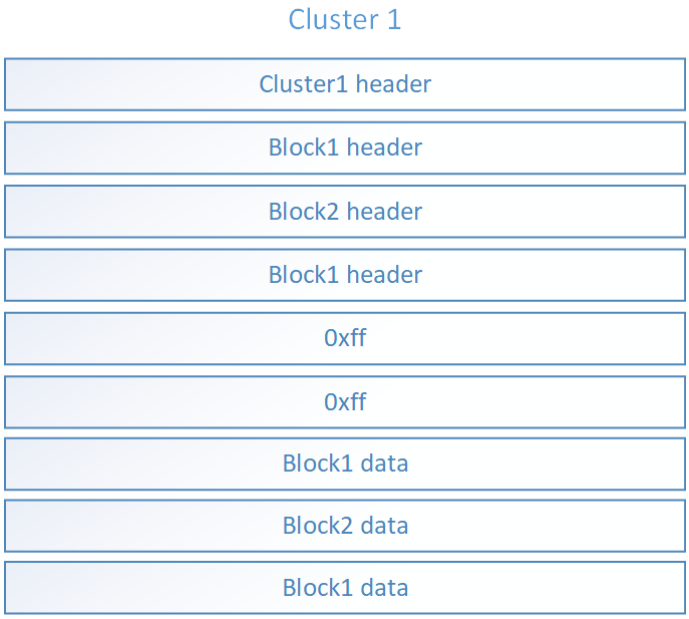
当我们需要存储一个Block1的数据，调用Fee_Write函数，存储完成后得到的cluster模型如下图，Block包含两个部分，Block header和Block Data。且Block header的位置顺延在Cluster header后，Block1 Data的位置是在Cluster的末尾，地址递减地去写。



接下来我们再去存储一个Block2地数据，调用Fee_Write，存储完成后得到的cluster模型如下图。观察一下Block2 header和Data的位置。

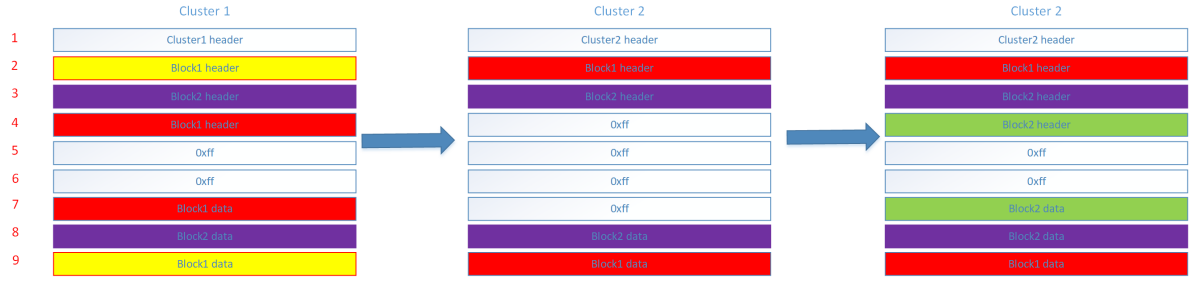


此时Block1存储的数据有变化，需要更新Block1里面存储的数据，调用Fee_Write函数，存储完成后得到的cluster模型如下图。此时Cluster中包含了两个block header和Data。此时如果调用Fee_Read函数去读取Block1的数据，则会将最新储存的Block1数据返回给用户。**其原理是FEE算法读取Block数据的时候，地址最大的同序号block即为最新的Block数据。**



Block2的数据又要更新了，此时Block2的数据仍然按照前面的顺序，写入block2的新的header和data，但是如果此时cluster的剩余空间不足怎么办？此时又要执行一个换页的操作（swap操作）。

如下图所示，如果我需要新写入一个数据而当前cluster剩余的空间不足够存储新的数据，第一步：FEE算法会将当前cluster1中block的**最新**数据，写入到下一个cluster2中。这个操作我们称为swap。第二步：换页完成后，再将需要存储的最新的Block2的数据存储到新的Cluster中。需要注意的时，此时新的Cluster2中会有两个Block2。一个是Swap后Cluster1中存储的最新的Block2数据，另一个是完成Swap后，在Cluster2中存储的数据。如图所示，图中绿色Block2是最新的数据。



总结：

Cluster中每次存储Block的数据，不会擦掉历史数据，会存储到Cluster中剩余的空间，如果剩余的空间不足，会触发swap换页操作，将当前的cluster中每个block的最新数据转移到新的cluster。因此如果在block的大小固定的情况下，Cluster越大，可以存储blcok数据的次数会越多。

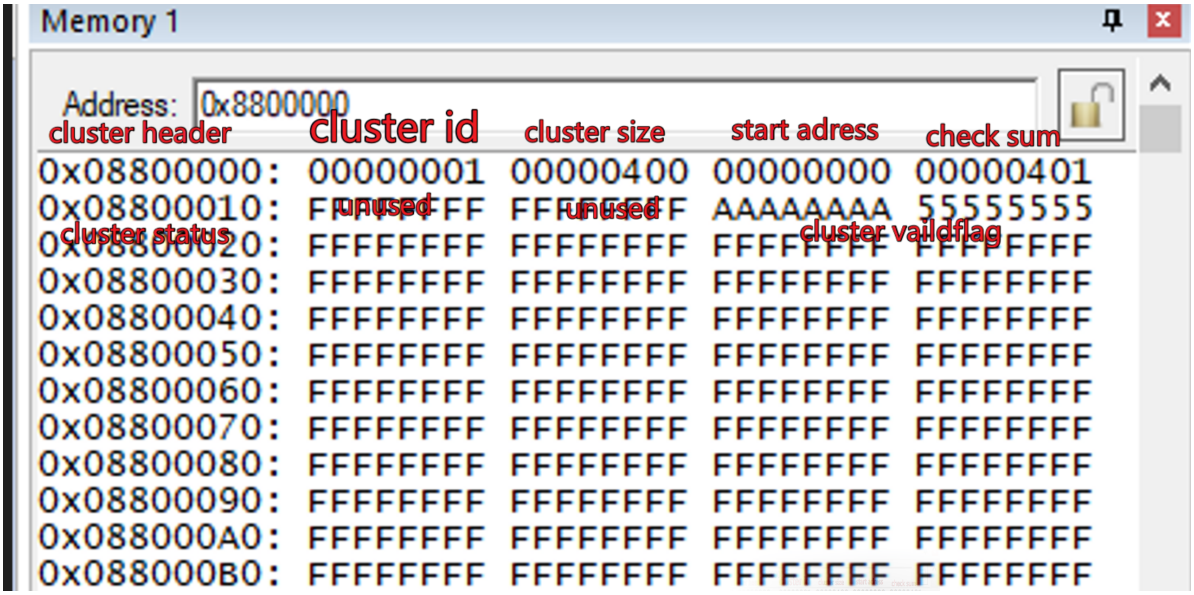
一起看看Cluster和Block都有什么吧！

Cluster数据解析

Cluster中包含Cluster header和block的数据。Cluster header中包含了当前cluster的使用状态和信息。Cluster ID代表当前cluster的序号，Cluster Size是当前Cluster的容量大小，以字节为单位。StartAdress表示当前cluster的起始地址对于基地址的偏移（Base:0x08000000）。

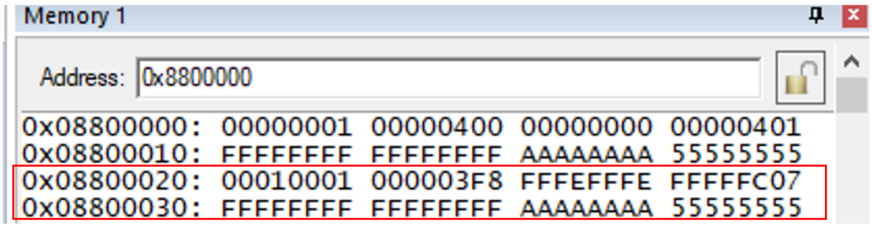
如下图所示，我们可以根据内存的信息得出这是一个cluster id为0x01,起始地址为0x08800000+0，大小为0x400字节的cluster。

Cluster的大小决定了最大可以存储的的数据量，Cluster的数量决定了整个Dflash擦写数据的次数。



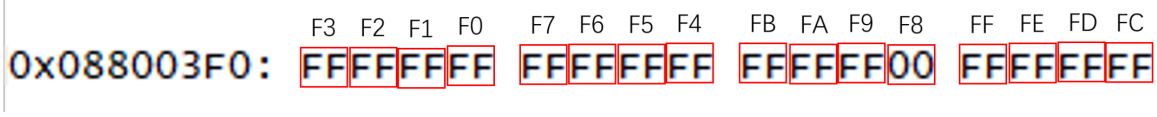
Block数据解析

Block组成由Block header和data组成，下图为一个Block的结构，其中Block ID为当前Block的序号，BlockLength为这个Block中存储的data的大小，最小为8个字节。BlockDataAdress为当前Block的Data的起始地址对于基地址的偏移（Base:0x08000000）。



BlockID	BlockLength	BlockDataAddress	~(BlockID)	~(BlockLength)	~(BlockDataAddress)	Block Header 1
BlockInvalidFlag			BlockValidFlag			Block Status 1

从上图的信息来看，我们可以分析出这段内存存储的是Block1的数据，其大小为1个字节（最小为8个字节，即使配置block大小小于8个字节，实际也会占用8个字节。）Block1对应的Data的地址为0x08800000+0x03F8 = 0x088003F8；如下图所示，存储的Data为00。



2. FEE重要函数、参数介绍

Cluster与Block配置

Fee_Cfg,c 文件中 Fee_aaGstCluster[FEE_NUMBER_OF_CLUSTER] 用于定义Cluster的大小和数量，以GD32A503VD举例，其Dflash的大小为64K，那么配置的Cluster大小*Cluster数量不能超过64KB。确保每个Cluster的大小要保持一致。且Cluster的大小要小于所有block所占空间的总和。

```
CONST(Fee_ClusterType, AUTOMATIC) Fee_aaGstCluster[FEE_NUMBER_OF_CLUSTER] =
{
    {.uiFeeClusterStartAddr = 0,//Cluster的偏移地址
    .uiFeeClusterLength = 4096},//Cluster的大小
```

```
    {.uiFeeClusterStartAddr = 4096,
    .uiFeeClusterLength = 4096},

    {.uiFeeClusterStartAddr = 4096*2,
    .uiFeeClusterLength = 4096},

    {.uiFeeClusterStartAddr = 4096*3,
    .uiFeeClusterLength = 4096},

    。。。。。。
}
```

Fee_Cfg,c 文件中Fee_aaGstBlock[FEE_CRT_CFG_NR_OF_BLOCKS]用于定义Block的大小和数量，因为每个block还要由对应的block header，所以需要合理安排block和Cluster的大小。

```
CONST(Fee_ConfigType, FEE_CONST) Fee_aaGstBlock[FEE_CRT_CFG_NR_OF_BLOCKS]=
{
    {.uiFeeBlockNumber = 1,//Block ID
    .uiFeeBlockSize = 8, //Block Data大小
    .bFeeImmediateData = FALSE,
    },

    {.uiFeeBlockNumber = 2,
    .uiFeeBlockSize = 16,
    .bFeeImmediateData = FALSE,
    },


    {.uiFeeBlockNumber = 3,
    .uiFeeBlockSize = 32,
    .bFeeImmediateData = FALSE,
    },
    。。。。。。
}
```

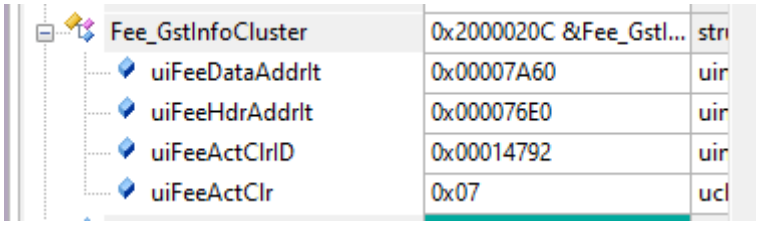
Debug过程找到最新Cluster和Block的位置

MCU上电初始化后，FEE会扫描整个Dflash。扫描完成后会确认最新的CLuster和每个Blcok的地址。用户可以通过Fee_GstInfoCluster来查看当前的最新Cluster信息。

```
typedef struct
{
    VAR(Fls_AddressType, FEE_VAR) uiFeeDataAddrIt;          /* Address of
current Fee data block in flash*/
    VAR(Fls_AddressType, FEE_VAR) uiFeeHdrAddrIt;             /* Address of
current Fee block header in flash */
    VAR(uint32, FEE_VAR) uiFeeActClrID;          /* ID of active cluster*/

    VAR(uint8, FEE_VAR) uiFeeActClr;             /* Index of active cluster */

}Fee_ClusterInfoType;
```



通过debug我们可以看到当前的Cluster为第7个，那么根据cluster的大小我们可以计算出当前cluster的起始地址在0x08800700(一个cluster大小为0x1000)。uiFeeActClrID是每次产生swap这个ID都会加1，也就是说当前已经产生过0x7A60次swap了。

Fee_aaGstInfoBlock变量则记录了所有的最新block数据的地址。

```
typedef struct
{
    VAR(Fls_AddressType, FEE_VAR) uiFeeDataAddr;                    /* Address of
Fee block data in flash */
    VAR(Fls_AddressType, FEE_VAR) uiFeeInvalidAddr;                      /*
Address of Fee block invalidation field in flash */
    VAR(Fee_BlockStateType, FEE_VAR) enBlockStatus ;          /* Current status
of Fee block */
}Fee_BlockInfoType;
```

Fls_Init()和Fee_Init() FEE初始化函数

这两个函数需要在MCU上电后初始化FEE模块，其作用是扫描Dflash，找到最新的cluster以及block地址,对ECC错误进行处理等功能。未初始化FEE模块直接使用FEE_Write或者Fee_Read会导致函数返回错误值。初始化使用例程如下：

```
    //FEE Init
    MemIf_StatusType status = MEMIF_IDLE;
    /* Initialize Fls driver */
    Fls_Init(&Fls_GstConfig);
    /* Initialize Fee driver */
    Fee_Init(NULL_PTR);
    /*Perform init Fee driver*/
    do
    {
        Fls_MainFunction();
        Fee_MainFunction();
        status = Fee_GetStatus();
        Result = Fee_GetJobResult();
    } while (status != MEMIF_IDLE);
```

Fee_Read()函数

读取block内的数据。该函数为同步函数，直接调用完就能获取数据。该函数的返回值只代表了该函数有没有执行到FLS_Driver层。如果想知道读取是否成功，要调用Fee_GetJobResult()函数，当返回MEMIF_JOB_OK代表读取成功。有一种情况需要说明的是，当调用Fee_Read()函数读取的flash地址存在ECC错误时，此时读取的结果会是MEMIF_JOB_FAILED，此时建议将FEE_Read读取到的数据是不可信的。因此使用时需要加上一个针对读取结果的判断。那ECC的问题如何解决，后面会单独讲解。

Fee_Write()函数

将数据写入指定的block中。该函数为异步函数，因此当函数调用完之后，数据并不会写入flash中。举例如下图，在10ms轮询函数中调用fee_write函数，然后当代码执行到Fls_MainFunction();Fee_MainFunction();这两个函数时，才会执行真正的写，并且具体执行多少个周期要看具体情况，数据的大小，是否发生换页，当前Flash_state是否为IDLE等等。因此调用Fee_Write()返回的结果实际并不能代表该数据是否写入成功，真正能确认的函数是Fee_GetJobResult()函数。

Fee_GetJobResult()函数

Fee_GetJobResult()函数是获取上一次的操作结果。问题是我们什么时候调用，因为不知道什么时候操作结束。并且如果在一个10ms周期中多次调用FEE_Write函数的话，我们没办法准确知道这个函数如果返回失败的话到底是哪个写操作失败。实际上这些问题是在NVM层去进行操作的。并不是没有办法解决，只是在NVM上会有相关的管理算法。

关于NVM层，其在AutoSar中处于应用层的位置，下图为NVM的简单解释，这里不再详细赘述。

NvM（NVRAM Manager，非易失性存储器管理器)是AUTOSAR存储服务层的核心模块，它将ECU的持久数据存储设施组织在称为“NvM块”的单独可管理单元中。NvM组件独立于它们的实际存储位置管理这些块(比如闪存设备中的串行EEPROM或EEPROM模拟)，并提供读取、写入、恢复、使NvM块无效和擦除的服务，将Nv Data存储到对应的NvRAM Block。其中Nv Data指的是存储在非易失性存储器中的数据，这些数据中被分为两类一类是不能有延时需要立即写入存储器中的数据，这一类数据被称为crash data；另一类为普通的数据，其写入存储器的可以存在一定的延时。而NvRAM Block是指整个管理和存储非易失数据的结构，它的基本单位称为Nv Block7Nv Block驻留在非易失存储中，它是读、写等操作的基本单位。若Nv Block 中存储的 NvData 为 Crash Data,则该 Nv Block 称为 Immediate Block。另外,AUTOSAR 标准中，对每个Nv Block都设置了一个优先级，优先级的范围为O到255，ImmediateBlock的优先级为0，0为最高优先级。每一个读写任务即是把数据从一个NvBlock读取或写入的过程，故该Nv Block的优先级即是对应读写任务的优先级。

AUTOSAR标准中规定对Nv Block的读、写过程采用异步的方式实现，即用户调用特定的函数发出读、写请求，将读、写请求任务存储到队列中(AUTOSAR标准规定，同一时间，队列中只能存储一个针对同一个Nv Block的任务)。通过另一个周期性调用的函数从队列中抓取Nv Block的读、写请求任务，完成对应的读、写操作。AUTOSAR标准中从读写队列中读取Nv Block读写请求任务的顺序根据用户配置参数NvmJobPrioritization(布尔型用户配置参数)是否为Ture，被分为基于Nv Block优先级的顺序处理和先来先服务的顺序处理两种方式。AUTOSAR NvM 378中规定在基于Nv Block优先级任务处理的方式下，NvM模块需要使用两个队列，一个为立即队列，它仅存放crash data的写任务,而另一个为普通队列，它存储除Crash Data写任务以外的其他任务。立即队列中的任务可以抢占普通队列中任务的执行(即取消普通队列中正在进行的任务，执行立即任务队列中的写任务)， 普通任务队列不能抢占其他任务，但也需要根据读写任务Nv Block优先级的顺序执行。

Fee_MainFunction()和Fls_MainFunction()

我们可以理解为FEE的初始化、读、写等函数都是下发了一个指令，而实际执行这些指令的是由Fee_MainFunction()和Fls_MainFunction()这两个函数去执行的。对于Fee_Write()这种异步函数，其写入一个数据不是执行完一次Fee_MainFunction()和Fls_MainFunction()就可以完成的，他们会将这个指令分段执行，譬如我要先确认写入的位置剩余空间是否足够，然后先写block header、再写Data，再计算校验完成后写入block vaildflag。如果需要swap的话我可能需要更多的步骤。因此往往调用一个Fee_Write后需要执行多次Fee_MainFunction()和Fls_MainFunction()函数。而对于Fee_Read这种同步函数，起始只需要执行一次Fee_MainFunction()和Fls_MainFunction()后即可，**但是读和写的时候当前正在执行别的读、写等任务，那指令将无法下发，必须等到下一次FEE状态为IDLE的时候才能下发。所以在实际调用的时候需要多加注意，事实上，NVM层就是做这个事情的，他会监控FEE当前状态，对于新下发的读写等指令可以进行缓存，等到总线空闲时按照优先级去执行这些指令。**具体的使用可以参考我们提供的demo，或者用户有更好的思路也可以去尝试。

3. FEE使用

Fee_Init()

该函数在MCU初始化时调用，为异步函数，其主要功能是扫描当前Dflash，找出最新的cluster和block数据。如前面所说，Fee_Init和Fls_Init函数也是一个指令，具体的实施操作还需要Fls_MainFunction()和Fee_MainFunction()函数。

```
MemIf_StatusType status = MEMIF_IDLE;
 /* Initialize Fls driver */
 Fls_Init(&Fls_GstConfig);
 /* Initialize Fee driver */
 Fee_Init(NULL_PTR);
 /*Perform init Fee driver*/
 do
 {
     Fls_MainFunction();
     Fee_MainFunction();
     status = Fee_GetStatus();
     Result = Fee_GetJobResult();
 } while (status != MEMIF_IDLE);
```

Fee_Write()

该函数需要在FEE模块初始化完成后调用，如果此时FEE处于BUSY状态，此时函数指令是未下发的。函数的使用可以有两种方式，

第一种使用方式会对MCU的运行产生阻塞，调用Fee_Write(1, DataBlock1)后在while循环中一直调用 Fls_MainFunction()，Fee_MainFunction()。直到写入成功，FEE状态由BUSY变为IDLE。此时再调用Fee_GetJobResult()获取此次写入的结果。

```
/*Write data to block 0*/
Ret = Fee_Write(1, DataBlock1); //1
/*Perform write data to Block 0*/
do
{
    Fls_MainFunction();
    Fee_MainFunction();
    status = Fee_GetStatus();
    Result = Fee_GetJobResult();
} while (status != MEMIF_IDLE);
```

第二种写入方式可以避免阻塞，但是面对一个周期同时存在多次写入的场景会更复杂一点，其主要的问题是如果一个周期内同时调用了多个Fee_Write（）函数，该怎么去将任务缓存并且将Fee_GetJobResult()函数返回的结果与对应的写任务对应起来。

下图的示例，启动了Systick定时器，每1ms计时一次，每10ms将task_10ms_flag置1，在10ms周期调度中使用Fee_Write(1, DataBlock1)，然后函数继续运行，当代码执行到Fee_MainFunction()和Fls_MainFunction()会将当前写的任务分段执行。用户可以调用status = Fee_GetStatus()查询FEE状态是否为IDLE来确认是否写入完成，执行Result = Fee_GetJobResult()来查询当前写操作是否成功。

```
while(1)
{
    if (task_5ms_flag == 1)
    {
        task_5ms_flag = 0;
    }
    else if (task_10ms_flag == 1)
    {
        task_10ms_flag = 0;
        Fee_Write(1, DataBlock1);
```

```
    }
    else
    {
        // do nothing
    }
    Fee_MainFunction();
    Fls_MainFunction();
}
```

Fee_Read()

该函数为同步函数，执行Fee_Read下发读取指令后，需要运行Fls_MainFunction()和Fee_MainFunction()才能执行实际的读操作，因此用户想执行完Fee_Read后立刻读到数据，需要在下面使用如下的用例。可以通过Fee_GetJobResult()函数来判断读取是否成功，如果读取的地址有ECC错误或者通不过校验，Fee_GetJobResult会返回FAILED。

```
Fee_Read(1, 0, ReadDataBlock1, 8);
do
{
    Fls_MainFunction();
    Fee_MainFunction();
    status = Fee_GetStatus();
    Result = Fee_GetJobResult();
} while (status != MEMIF_IDLE);
```

4. 异常情况下FEE数据恢复原理

写入数据过程中断电/复位，数据可恢复到上一次写入成功的数据

先读取Block6的数据，再将读取到的计数值打印并且加1，重新写入block6中，如果写入成功也会打印写成功。

```
void DataWriteTest()
{
    uint8_t status = 0;
    uint8_t Result = 0;

    if(Fee_GetStatus() == MEMIF_IDLE)
    {
        Fee_Read(6,0,DataBlock6,256);
        do
        {
            Fls_MainFunction();
            Fee_MainFunction();
            status = Fee_GetStatus();
            Result = Fee_GetJobResult();
        } while (status != MEMIF_IDLE);

        DataCount = (((uint16_t)DataBlock6[1] << 8 ) |
((uint16_t)DataBlock6[0]));
        SEGGER_RTT_printf(0,"Read Count = %d\n",DataCount);
    }


    if(Fee_GetStatus() == MEMIF_IDLE)
```

```
    {
        DataCount++;
        DataBlock6[0] = (uint8_t)(DataCount&0xff);
        DataBlock6[1] = (uint8_t)(DataCount>>8);

        Fee_Write(6, DataBlock6);
        SEGGER_RTT_printf(0,"Write BLOCK6 COUNT %d\n",DataCount);

    }
}


while(1)
{
    if (task_5ms_flag == 1)
    {
        task_5ms_flag = 0;
    }
    else if (task_10ms_flag == 1)
    {
        task_10ms_flag = 0;
        DataWriteTest();
    }
    else
    {
         // do nothing
    }
    Fee_MainFunction();
    Fls_MainFunction();
    LastResult = Result;
    Result = Fee_GetJobResult();
    if(Result == MEMIF_JOB_OK && LastResult != MEMIF_JOB_OK)
    {
        SEGGER_RTT_printf(0,"Write success!\n");
    }
}
```

MCU上电后一直按下复位键或者断电上电，查看打印出来的log。

正常运行的情况

```
Read Count = 16229
Write BLOCK1 COUNT 16230
Write success!
Read Count = 16230
Write BLOCK1 COUNT 16231
Write success!
Read Count = 16231
Write BLOCK1 COUNT 16232
Write success!
****************************************
****************************************
****************************************
****************************************
****************************************
Read Count = 16232
Write BLOCK1 COUNT 16233
Write success!
Read Count = 16233
Write BLOCK1 COUNT 16234
Write success!
Read Count = 16234
Write BLOCK1 COUNT 16235
Write success!
Read Count = 16235
```

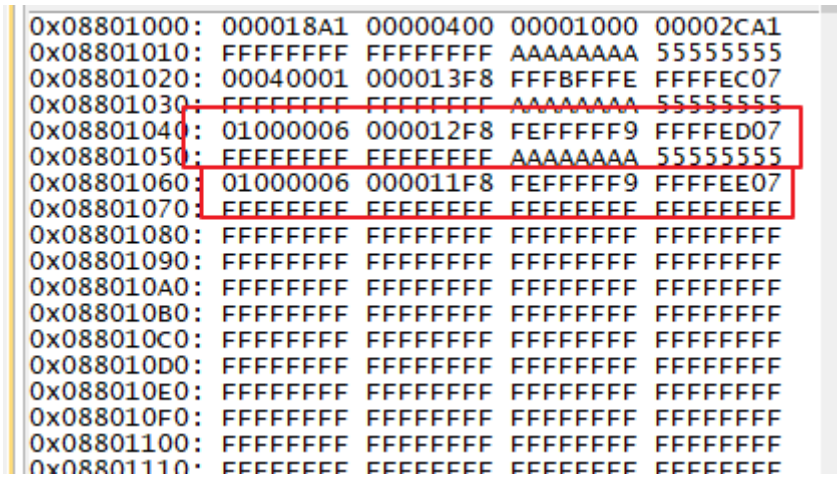
数据写入失败，断电后恢复上一次数据

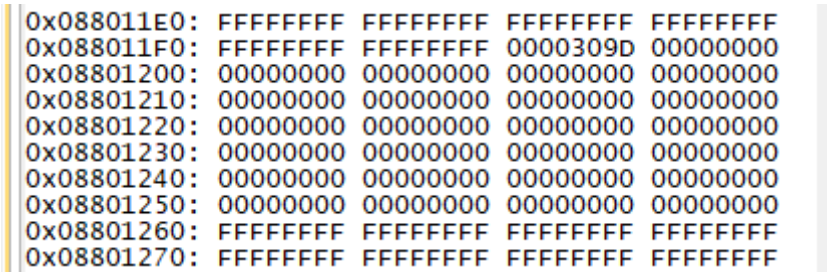
```
Write BLOCK1 COUNT 16713
Write success!
Read Count = 16713
Write BLOCK1 COUNT 16714
Write success!
Read Count = 16714
Write BLOCK1 COUNT 16715
Write success!
Read Count = 16715
Write BLOCK1 COUNT 16716
****************************************
****************************************
****************************************
****************************************
****************************************
Read Count = 16715
Write BLOCK1 COUNT 16716
Write success!
Read Count = 16716
Write BLOCK1 COUNT 16717
Write success!
Read Count = 16717
Write BLOCK1 COUNT 16718
Write success!
```

```
Write BLOCK1 COUNT 17388
Write success!
Read Count = 17388
Write BLOCK1 COUNT 17389
****************************************
****************************************
****************************************
****************************************
****************************************
Read Count = 17388
Write BLOCK1 COUNT 17389
Write success!
Read Count = 17389
Write BLOCK1 COUNT 17390
```

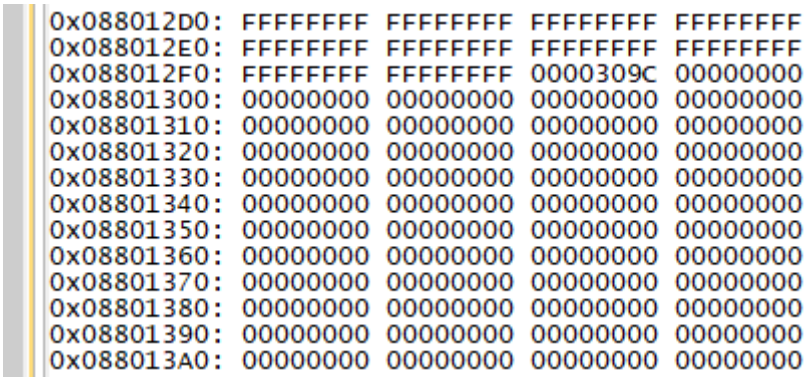
我们还可以通过debug确认是否写入失败&数据成功恢复：下图一可以看到最新的数据应该是在0x11F8的偏移值，但是缺少了0xAAAAAAAA55555555的标志位，说明该段数据没有被完全写入，而上一个有效数据的地址在0x12F8的偏移值。我们一起看一下这两个地址的COUNT值分别是多少。

0x12F8的值是0x309D

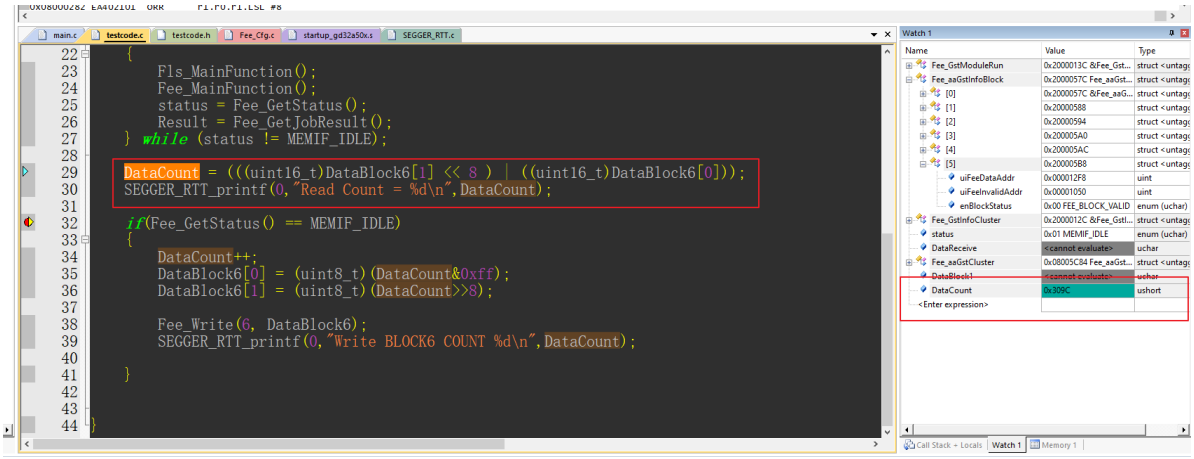




0x11F8的值是0x309C



通过debug可以看到调用FEE_READ读取到的值是0x309C，也就表明了如果写入失败，可以追溯到上一次的有效数据。



ECC问题处理

ECC相关知识介绍

1. ECC错误的产生：DFLASH中的数据单位是64bit的数据+8bit的校验码，如果在falsh进行擦写时产生了掉电，复位，电压不稳等情况，都可能导致flash数据混乱，从而产生ECC错误。这里的ECC错误主要是2bit及2bit以上的错误，单bit我们的A503是可以自动纠错的。
2. ECC错误的识别：如果出现了ECC双bit及以上的错误，FMC_ECCCS寄存器中ECCDET位会置1，但是前提是MCU访问了出现ECC错误的地址。如果0x08800000这个地址出现了ECC错误但是MCU没有访问过这个地址，ECCDET标志位是不会置1的。如果你尝试去读取出现ECC错误的这个地址，返回的值会变成全F。
3. ECC错误清除：将出错地址所在的Flash页擦除，即可清除ECC错误。

产生ECC错误的现象

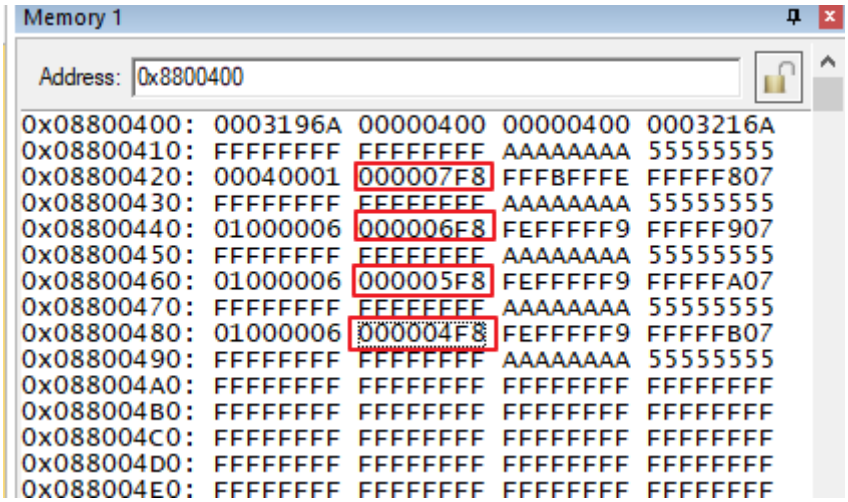
测试方法：一直进行FEE_WRITE和FEE_READ操作，然后在Debug状态下运行，一直复位，将断点打在下图位置Fls_MainFunction()->Fls_LLD_job_Read()

```
596             }
597         }
598
599         LenFlag1 = Fls_HLD_flag_get(FMC_FLAG_ECCCOR);ECC单位错误标志位
600         LenFlag2 = Fls_HLD_flag_get(FMC_FLAG_ECCDET);ECC双位错误标志位
601         /* no ECC error occured */
602         if((SET != LenFlag1) && (SET != LenFlag2)) {
603             return MEMIF_JOB_PENDING;
604         } else {
605             //FLS_ENTER_CRITICAL_SECTION(RAM_DATA_PROTECTION);
606 产生ECC错位，返回FAILED  Fls_GstModuleRun.siJobResult = MEMIF_JOB_FAILED;
607             //FLS_EXIT_CRITICAL_SECTION(RAM_DATA_PROTECTION);
608             Fls_HLD_CommandComplete();
609             Fls_HLD_flag_clear();
610
611             return MEMIF_JOB_FAILED;
612         }
613     }
614 }
615
```

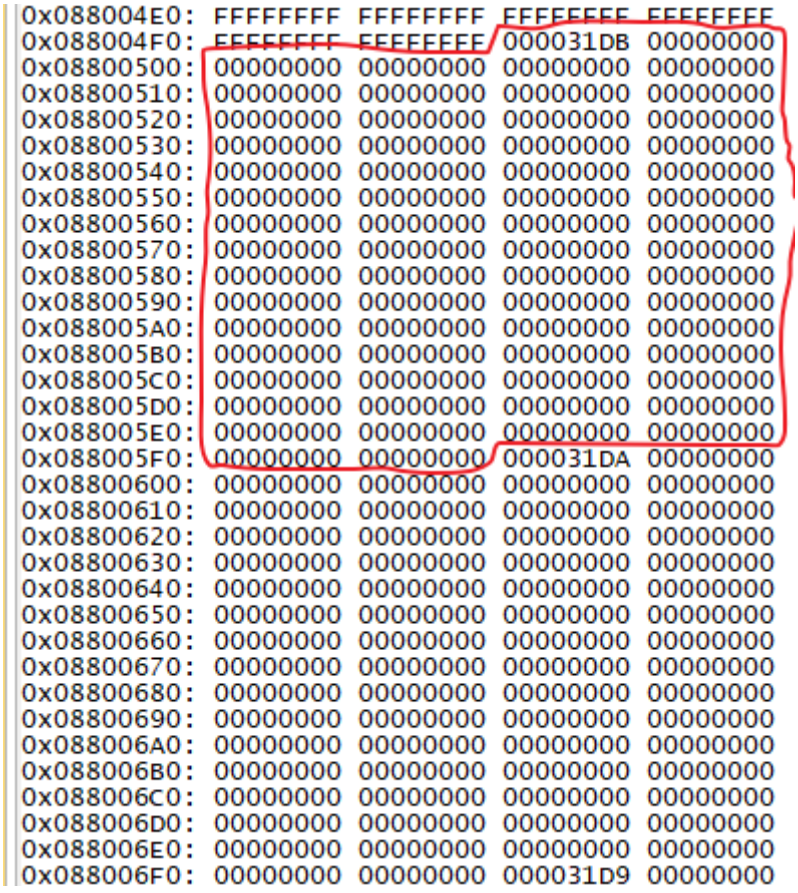
进到断点后说明此时发生了ECC错误，我们再来定位寻找ECC错误的地址。我们操作的事Block6，可以看一下当前BLOCK6的最新地址；

Name	Value	Type
Fee_GstModuleRun	0x2000013C &Fee_Gst...	struct <untagg
Fee_aaGstInfoBlock	0x2000057C Fee_aaGst...	struct <untagg
[0]	0x2000057C &Fee_aaG...	struct <untagg
[1]	0x20000588	struct <untagg
[2]	0x20000594	struct <untagg
[3]	0x200005A0	struct <untagg
[4]	0x200005AC	struct <untagg
[5]	0x200005B8	struct <untagg
uiFeeDataAddr	0x000004F8	uint
uiFeeInvalidAddr	0x00000490	uint
enBlockStatus	0x00 FEE_BLOCK_VALID	enum (uchar)

查看基地址0x08800000+偏移基地址0x490的地址，这是最新的操作地址，那发生ECC错误的地址在该地址的前面。我们直接查看0x08800400这个地址开始，因为这个地址是cluster的头。



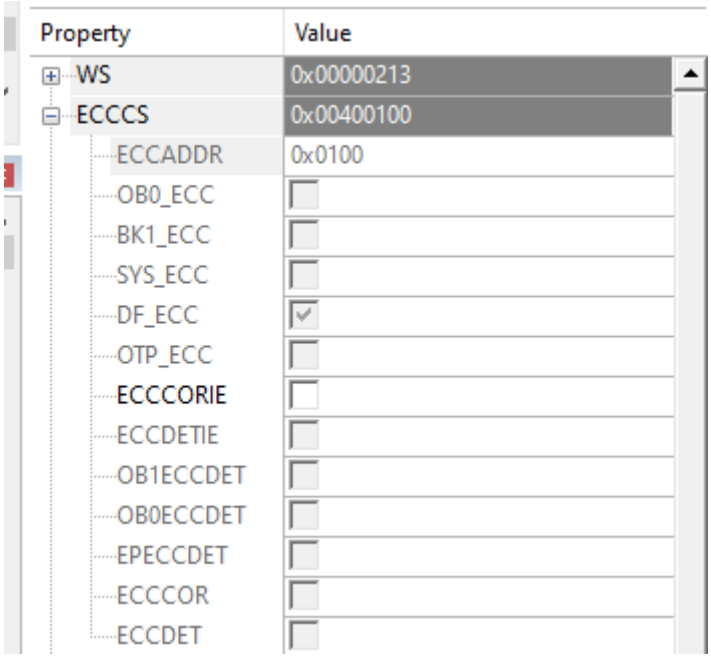
查看这几个历史的数据，图1是正常一个block的数据；图二的数据异常，这个地方发生了ECC错误。



```
0x088007F0: 00000000 00000000 00000566 FFFFFFFF
0x08800800: FFFFFFFF FFFFFFFF 7967FFBE 6FFFFFBD
0x08800810: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x08800820: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x08800830: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x08800840: EF5DFFFF FFFF7EFF FFFFFFFF FFFFFFFF
0x08800850: FFFFFFFF FFFFFFFF EFEFFFFF FFEFF7FF
0x08800860: DF7FEFEF F7F7F9FF FFFFFFFF FFFFFFFF
0x08800870: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x08800880: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x08800890: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x088008A0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x088008B0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x088008C0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x088008D0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x088008E0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x088008F0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x08800900: DFBFD7FF FFFFFFFF FFFFFFFF FFFFFFFF
0x08800910: F7E7FEDF FFE7FEFF BD7FEF3F 9BFFDFFF
0x08800920: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x08800930: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x08800940: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x08800950: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
```

我们也可以根据寄存器找出出现ECC的错误地址，如果所示DF_ECC置1，说明ECC错误发生在DFLASH中，根据ECCADDR的偏移地址可以计算出产生ECC错误的地址为0x08800000（DFLASH基地址）+ 0x100 * 8 = 0x0880 0800，与上图flash中的ECC错误一致。

14:0	ECCADDR[14:0]	检测到 ECC 错误的双字的偏移地址。 错误地址 = 基地址 + ECCADDR[14:0] * 8，基地址可以是 bank0，bank1，数据闪存，EEPROM SRAM，system 区，选项字节 0，选项字节 1 以及 OTP 的起始地址。 详细信息参考 *2.3.1*。



ECC出现在不同位置的处理方式

如图所示，以上是一个简单的dlfash存储数据的模型，此时需要在Cluster1中写入一个新的block1数据，但是剩余的空间不够用，因此触发了swap，将cluster1中最新的block1和block2的数据转移到cluster2中，然后再重新写入最新的Block1的数据，此时block1的最新数据即为★所在的位置。

我们大概介绍一下完成上面的描述的这个任务都经历了哪些操作。

1. FEE_Init（如果是刚上电/复位的话）上电后会scan当前的dflash的数据，他会扫描整个dlfash的所有的cluster和block的header。然后找到cluster最新的数据并且校验，校验会校验cluster的header和cluster的有效标志位是否完整。然后根据cluster id最大的认为是最新的cluster；确认cluster后继续扫描cluster中的block，验证block header是否完整。然后将每个不同的block id根据地址顺序找出最新的block数据。记录各自block的地址。
2. 写入新的block1数据，此时cluster1的剩余空间不足，需要进行swap。
3. 首先将cluster的空间进行擦除，将其全部擦除为0xff。
4. 写入cluster head部分，Cluter的 ID、大小、校验和等，此时不会写入cluster vaild flag。
5. 写入block head，然后写入block data，校验无误之后写入block vaild flag。
6. 等到全部blcok数据写入并校验完成，将cluster的vaild flag写入，代表此时swap操作成功
7. 最后将触发swap操作的blcok1的数据写入cluster2中，至此整个任务完成。

ECC的错误往往都是在擦或者写flash的时候由于电压不稳、断电、复位等原因导致的。我们可以根据上面的操作来分析哪个环节可能导致ECC错误并应该如何处理。

在第3步操作，擦除cluster2的时候如果产生ECC，此时上电后scan得到最新的数据是处于cluster1中，上一步的fee_write的block1数据并未写入，此时cluster1中的数据都是有效的，读取这些数据可以正常读取。如果此时重新写入block1的数据，仍会先将cluster2的空间先擦除一遍，此时ECC问题也清除了。

第4步、第5步、第6步操作执行的过程，如果产生了ECC，此时上电后scan，发现cluster2的vaild flag并未写入，此时cluter2并不是一个有效的cluster，因此程序仍会认为cluster1为最新的cluster。因此也不会存在问题。

第7步操作执行的过程中，如果产生了ECC，此时上电后scan，发现cluster2为最新的cluster，但是在扫描block的过程中会出现两种情况：

1. 如果ECC的错误产生在block head，此时上电后scan会读取block head，因此在Init的时候就能检测到ECC的故障，那么此时blcok1存储的数据会按照上一次最新的block1数来读取。并且由于检测到了ECC，在Init后如果执行了FEE_Write操作，FEE模块会自动执行swap操作，将所有的有效数据转换到cluster3中。那cluster2的ECC错误会在后面swap到cluster2地址的时候提前擦除，清除ECC故障。
2. 如果ECC的错误产生在block data阶段，此时由于断电会导致block vaild flag并未写入。因此此时★标志的block并不会认为是有效数据。