

MSP430FRXX FRAM 操作注意点

TI 新出的 FRAM 430 在数据写速度以及寿命上相较于原先的 Flash 型的 MCU（包括但不限于 430）都有了质的飞跃。一般而言 Flash 的擦写寿命是 10 万次，超过这个次数有可能导致数据存储失效，而铁电的性质决定了它的擦写寿命几乎为“无限次”，达到 10^{15} 万次。这个特性在很多数据更新频繁的产品上具有非常高的应用契合度。当然，我们这里要讲的是如何保证数据在 FRAM 上不仅能够快速更新，还要保证更新后的数据是稳定不易被“篡改”的，那 MPU 在这里的作用不可忽略。

TI 已经新推出了多种系列的 FRAM 430 芯片，有些芯片是带 MPU 模块的，比如 FR6972、FR6989 等。有些则是不带 MPU 的，比如 FR4133 以及 FR2633。事实上 FR2xx 以及 FR4xx 迄今为止都是不带 MPU 模块。

针对 FRAM 的特性，我们从以下几个方面进行阐述分析：

1) FRAM 的易操作性

MSP430 的 FR 系列使用的是 FRAM 内存，它相比 FLASH 型的 MSP430 有诸多优点，擦写次数接近无限次，而 FLASH 擦写寿命只有 10 万次，在 Data Log \position track 等应用中将体现 FLASH 出无法比拟的优越性。这里我们需要重点讲述下 FRAM 在编程使用时的注意点。

首先，FRAM 最大的优点就是“写”操作比 FLASH 简单很多。FLASH 型 430 如果需要更新 FLASH 数据需要有一个 PUMP 将电压升压到 FLASH 编程电压。如果电源电压小于 2.2V (例如 MSP430F2xx、MSP430G2xx) 对 FLASH 的操作将会失败，而 FRAM 就像操作 RAM 一样，无需额外升压。

我们通常看到一个 FLASH 操作函数是这样的：

```
Void Flash_Erase(u16_t EraseAdr)
{
    u16_t Flash_ptrA;
    Flash_ptrA = (u16_t *) EraseAdr;           // Point to beginning of seg
    FCTL2 = FWKEY + FSSEL0 + FN1;             // 配置时钟，使得 FLASH 时钟在 257Khz~476KHz
    FCTL3 = FWKEY;                             //FLASH 解锁
        FCTL1 = FWKEY + ERASE;                 // Set Erase bit
        *Flash_ptrA = 0x00;                     // Dummy write to erase Flash seg
        while(FCTL3 & BUSY);                    //等待擦除操作完成，变空闲
    FCTL3 = FWKEY + LOCK;                       // 重新上锁
}
```

```
Void Flash_Write(u16_t WriteAdr, u16_t* DataBuf, u16_t WriteByts)
{
    u16_t i, Flash_ptrA;
    Flash_ptrA = (u16_t *) WriteAdr;           // Point to beginning of seg
    FCTL2 = FWKEY + FSSEL0 + FN1;             // 配置时钟，使得 FLASH 时钟在 257Khz~476KHz
```

```
FCTL3 = FWKEY;                //FLASH 解锁
FCTL1 = FWKEY + WRT;          // Set Write bit
for(i=0;i< WriteByts;i++)
{
    *Flash_ptrA = *DataBuf++;    // Dummy write to erase Flash seg
    while( !(FCTL3 & WAIT) );
}
    while(FCTL3 & BUSY);        //等待擦除操作完成，变空闲
FCTL3 = FWKEY + LOCK;         // Clear LOCK & LOCK bits
}
```

从这里我们可以看到需要的操作有：

- 1) FLASH 时钟设定
- 2) FLASH 的解锁
- 3) 选择 FLASH 操作，是擦除还是写
- 4) 启动擦除或者写操作
- 5) 等待操作完成
- 6) FLASH 重新加锁，并退出。

其实，后台的操作更复杂，MCU 需要等待 FLASH 编程电压达到标称值方可进行。

而 FRAM 呢？

以 MSP430FR4133 为例：

0xF000 这个地址在 MSP430FR4133 内属于程序存储区，它是 FRAM。如果要将这个内存单元“擦除”，只

要

```
void Fram_Erase(void )
{
    u16_t* ErasePtr = (u16_t *) (0xF000);
    *ErasePtr = 0xFFFF;
}
```

实际上在 FRAM 内可以认为没有“擦除”操作，你只要将你希望的值赋值给它，内存单元当即被“覆盖”。

是不是非常高效、简单？

但是，反过来想想，如此“简易”方式即可将程序存储区的数据改写，是不是太过于“草率”了？这个请看后续 FRAM 的安全性。

其次，FLASH 只能从 1 “写” 0， 要将数据从“0”恢复到“1”时，只能选择“擦除”操作，擦除操作才是将“0”改成“1”的操作。此外，擦除的最小单位是一个 Segment，这就导致我们即使误写了一个 bit 就需要擦除整个 segment。从 MSP430 的角度来讲的话，只要一个 bit 出错（主 FLASH 内），即需要擦除 512 个字

节。这种情况不仅导致不必要的擦除操作（相比 FRAM、RAM 而言）导致 FLASH 寿命降低，另外，在数据频繁存储的应用中它还将明显的增加功耗。

Parameter	FRAM (FR4133) ⁽¹⁾	Flash (F2274) ⁽¹⁾
Program time for byte or word (maximum)	120 ns	116 μ s (approximately)
Erase time for segment (maximum)	Not applicable (pre-erase not required)	18 ms
Supply current during program (maximum)	No extra current during write (included in active power specification)	5 mA
Supply current during erase (maximum)	Not applicable (pre-erase not required)	7 mA
Nonvolatile memory maximum read frequency	8 MHz	16 MHz

2) FRAM 的安全性

我们之前有举例说明更改一个 FRAM 区的数据有多么的简单、高效。但是，如果这种“改写”并非设计者的本身意图，而是干扰呢？——这将是致命和不可饶恕的。当然，TI 也考虑到了这点。

实际上 MSP430FR4133 一上电即执行 Fram_Erase 函数是改变不了 0xF000 的值的。

```
void Fram_Erase(void)
{
    u16_t* ErasePtr = (u16_t *) (0xF000);
    *ErasePtr = 0xFFFF;
}
```

因为 MSP430FR413x 上电后默认 FRAM 是处于“写保护”状态，如果没有解锁，对 FRAM 的操作是被忽略的，它不会触发任何中断或者引发复位，但是它也改变不了内存单元的值。

Table 6-28. Memory Organization

	ACCESS	MSP430FR4133	MSP430FR4132	MSP430FR4131
Memory (FRAM) Main: interrupt vectors and signatures Main: code memory	Read/Write (Optional Write Protect) ⁽¹⁾	15 KB FFFFh-FF80h FFFFh-C400h	8 KB FFFFh-FF80h FFFFh-E000h	4 KB FFFFh-FF80h FFFFh-F000h
RAM	Read/Write	2 KB 27FFh-2000h	1 KB 23FFh-2000h	512 B 21FFh-2000h
Information Memory (FRAM)	Read/Write (Optional Write Protect) ⁽²⁾	512B 19FFh-1800h	512B 19FFh-1800h	512B 19FFh-1800h
Bootstrap loader (BSL) Memory (ROM)	Read only	1 KB 13FFh-1000h	1 KB 13FFh-1000h	1 KB 13FFh-1000h
Peripherals	Read/Write	4 KB 0FFFh-0000h	4 KB 0FFFh-0000h	4 KB 0FFFh-0000h

那么如何“刻意”的更改 FRAM 内的值呢？

如果这个内存单元在程序存储区，以 FR4133 为例，如果需要改写的内存单元在 0xC400~0xFFFF（常说的执行代码存储区）之间则需要对 SYSCFG0 寄存器的 PFWP 位进行清零操作；如果需要改写的内存单元在 0x1800~19FF 内（常说的信息段存储区），则需要对 SYSCFG0 寄存器的 DFWP 位进行清零操作。记得操作完毕之后，要重新置位相应的保护位，以免数据被“无意”更改。

1.16.2.1 MSP430FR413x SYSCFG0 Register (offset = 00h) [reset = 0003h]

System Configuration Register 0

Figure 1-31. SYSCFG0 Register

15	14	13	12	11	10	9	8
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
r0	r0	r0	r0	r0	r0	DFWP rw-1	PFWP rw-1

Table 1-28. SYSCFG0 Register Description

Bit	Field	Type	Reset	Description
15-2	Reserved	R	0h	Reserved. Always read as 0.
1	DFWP	RW	1h	Data FRAM write protection 0b = Data FRAM write enable 1b = Data FRAM write protected (not writable)
0	PFWP	RW	1h	Program FRAM write protection 0b = Program FRAM write enable 1b = Program FRAM write protected (not writable)

3) MSP430FR4xx 是否可以進行 FRAM 和 RAM 的任意分配？

MSP430FRxx 有一个优势就是用户可以灵活分配程序存储区和变量存储区的大小。那么 MSP430FR2xx 和 FR4xx 是否可以这么处理呢？

如果“RAM”需要增加的量在 512 个字节之内，我们认为是 OK 的；如果希望将大于 512Bytes 的 FRAM “划分”给 RAM 作为变量存储用途，我们则不建议。为什么？

从本文的“FRAM 安全性”中我们可以知道，FR2xx 和 FR4xx 的 FRAM 默认是写保护的，如果划分的 FRAM 大于 512 个字节，我们只能从主程序存储区去划分，将这段 FRAM 划分出来用于变量存储之后必须保证该段 FRAM 的“易失性”，也就是必须要将 SYSCFG0 的 PFWP 位清 0，如此存储于该区的“程序”则将失去保护，此举岂非“捡了芝麻丢了西瓜”，因小而失大。

如果划分给“RAM”的内存单元空间小于等于 512Bytes，我们可以将信息存储区(0x1800~0x19FF)的 FRAM 划分出来作为“RAM”之用，这里我们需要将 SYSCFG0 的 DFWP 位清 0。

除此之外，我们还必须修改 XCL 文件方能使得编译器“清楚”此段 FRAM 已经用作 RAM，作为变量存储之用。具体可见 SS 的应用报告

《MSP430FRxx 系列的 FRAM 和 RAM 自由调配应用报告》。

在 MSP430FR413x 系列中需要格外注意一点：FRAM 上电默认是写保护的，如果 FRAM 区被划分为 RAM 用途之后，通常编译器都会将 RAM 的最顶端（地址最大）作为堆栈的栈顶。在 C 语言编程中 main 函数并非第一个

被执行的函数，当出现函数调用需要压栈的时候会导致压栈失败进而程序跑飞。所以上电后第一件要执行的就是清除对应的 FRAM 写保护位，不然会由于 FRAM 性质的栈区被 写保护而导致压栈失败。

最后，FRAM 内存的响应访问速度最大为 8MHz，所以当 MCLK 频率大于 8MHz 的时候，必须要设置 wait state 以保证每次 FRAM 的访问都是可靠的

$FRCTL0 = FRCTLPW + NWAITS_x$.

然后再将 MCLK 调整到 8MHz 以上。

4) JTAG 以及代码的安全性

在整个铁电 430 系列里是不存在“物理”性的熔丝的，但是，有“电子”熔丝，这就意味着禁止 JTAG 或者 SBW 对内存单元的访问不需要大电流（熔丝内容详见 SS 的《MSP430 烧熔丝应用报告》）。

对于 FR4xx 以及 FR2xx 系列，JTAG 的电子密钥（熔丝）区为 0xFF80~0xFF83 共 4 个字节，如果这 4 个字节为 0000_0000 或者 FFFF_FFFF，则 JTAG 访问允许，除此之外的其它数据都将禁止 JTAG 的访问权限。如果 BSL 访问没有被禁止的话，可以使用 BSL（当然，得有正确的 BSL 密码）将 FF80~FF83 擦除，这样 JTAG 就可以重新访问，可以看出这个机制和以前的 FLASH 型的 MSP430 机制有不一样的地方，因为 FLASH 型的 JTAG “熔丝”烧断后都是不可逆的。另外，BSL 访问也可以被禁止，BSL 的电子密钥区为 0xFF84~0xFF87，如果这四个字节为 5555_5555，那么 BOR 之后 BSL 也无法访问了，如果此 4 个字节存储的是 ~~5555~~ 5555 的值，那么得到正确的 BSL 密码之后，BSL 还是可以访问。（注意这里 FR4xx、FF2xx 的密钥机制和后续 FR58xx、FR59xx、FR68xx、FR69xx 是不一样的。）

附注：

BSL 密码指的是存在于芯片内地址范围为 FFE0~FFFF 共 32 个字节的数据，这个地址空间恰好也是 MSP430 的中断向量存储区。当一款芯片确认后，一个应用程序往往不可能使用全部的 16 个中断向量，不使用的中断向量编译器默认就为 0xFFFF。此外，编译器产生的中断向量值肯定都是在主程序存储区地址空间内，这就给一些别有用心的人造成了可趁之机，利用穷举法破解 BSL 密码。所以在 MSP430F1xx 以及 F4xx 系列内，我们推荐用户往不使用的中断向量区间内写入一些随机值，尤其是那些不在主程序存储区地址空间内的值，以增加破解难度。除此之外，别的系列内，由于 TI 出厂的 BSL 代码做了升级，如果 BSL 错误一次即清除整个 FLASH，杜绝了穷举法进行破解的途径。

Table 1-5. Device Password, BSL Signatures, and JTAG/SBW Signatures

Name	Addresses	Value	Device Security	BSL and JTAG/SBW Behavior After Reset
Device Password	FFE0h-FFFFh	Depending on Vector Table configuration		The value is used to protect BSL and JTAG/SBW.
BSL Signature	FF84h-FF87h	5555_5555h	Secured, password not required	BSL is bypassed. User code starts immediately.
		Any Other Values	Secured, password required through BSL	BSL is invoked before user code starts.
JTAG/SBW Signature	FF80h-FF83h	FFFF_FFFFh	Not secured	JTAG/SBW is not locked.
		0000_0000h		
		Any Other Values	Secured	JTAG/SBW is locked

5) MPU 模块 (Memory Protection Unit)

并非所有铁电 430 都有 MPU 模块，比如上文讲述到的 FR2xx 以及 FR4xx 都是没有 MPU 模块的。但是，像 FR6972 就有 MPU 模块，MPU 模块可以分出三个权限：“读”、“写”，“执行”。不过上电以后，MPU 默认是把这三个 3 个权限都放开的，**在实际应用中，有 MPU 模块的 FR430 产品，一定要使能 MPU，并禁止“写”权限！**

MPU 模块相比 FR4133 只是依靠 SYSCFG0 中的两个位来对 FRAM 的数据进行“锁存”，它的功能更加强大。主要体现在两个方面：

1) 用户可以自己划分保护区

像 FR4133，芯片实际上是分两个“区间”进行保护的，一个是常说的代码存储区（code 区），一个是信息存储区。

1.16.2.1 MSP430FR413x SYSCFG0 Register (offset = 00h) [reset = 0003h]

System Configuration Register 0

Figure 1-31. SYSCFG0 Register

15	14	13	12	11	10	9	8
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
r0	r0	r0	r0	r0	r0	DFWP rw-1	PFWP rw-1

Table 1-28. SYSCFG0 Register Description

Bit	Field	Type	Reset	Description
15-2	Reserved	R	0h	Reserved. Always read as 0.
1	DFWP	RW	1h	Data FRAM write protection 0b = Data FRAM write enable 1b = Data FRAM write protected (not writable)
0	PFWP	RW	1h	Program FRAM write protection 0b = Program FRAM write enable 1b = Program FRAM write protected (not writable)

但是，有 MPU 模块的铁电 430，仅在代码存储区（code 区）就可以自定义 3 个“区间”，用户可以对这 3 个区间进行分别保护。所以它给用户有更多的“划分自由度”。

需要注意的 MPU 给用户带来了更多的自由度，但是这个“划分自由度”也并非是无限制的。以 FR6972

为例，它的程序存储区范围是 4400h~FFFFh。但是，分区的边界地址必须是以 4400h 为 BaseAddress 加上 1024 的整数倍所处的地址位置。即边界地址必须符合如下规则：

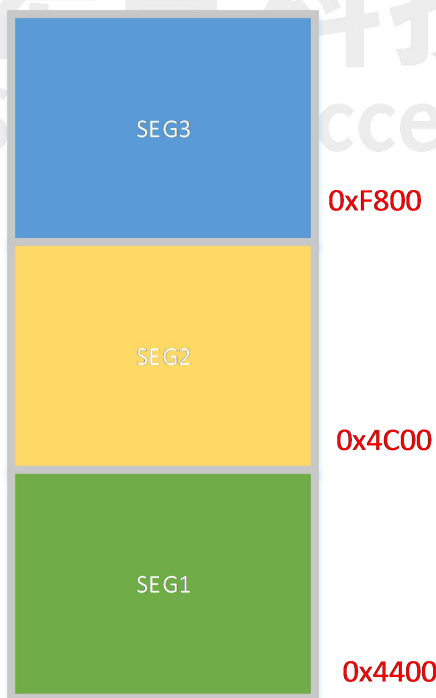
$$\text{BorderAddress} = \text{BaseAddress} \text{ (这里是 } 4400\text{h)} + 1024 * N \text{ (} N=1、2、3 \dots\dots \text{)}$$

2) 用户在对权限操作上可以更“具体”和“细致”

在 FR2xx 或者 FR4xx 的 430 芯片内, 用户只能对芯片早已划分好的信息段和程序存储段进行是否“可写”进行权限调度。但是有 MPU 模块的铁电 430 还能更加细分到是否“可读”，“可执行”。所以有 MPU 模块的铁电 430，它可以实行“可读”，“可写”，“可执行”三个权限的划分。这有什么好处呢？比方说表类客户中常有校准数据需要存储，那么可以将这些校准数据放置到信息段，然后将信息段权限设置为“可读”，“不可写”，“不可执行”。当这个校准数据需要更新的时候，将信息段权限设置为“可读”，“可写”，“不可执行”，然后进行铁电数据的更新操作，数据更新完成后重新设置为“可读”，“不可写”，“不可执行”。

同样，对于存在于程序存储区（主 code 区）由用户划分出来的 3 个段，也可以如此对权限进行细分或者更改。这在很多自升级应用中也是比较常见和有用的方法。

我们以 FR6972 为例, 将主 Memory 划分为三个地址块, (详情见 user guider 的 MPU 章节) example code 如下：

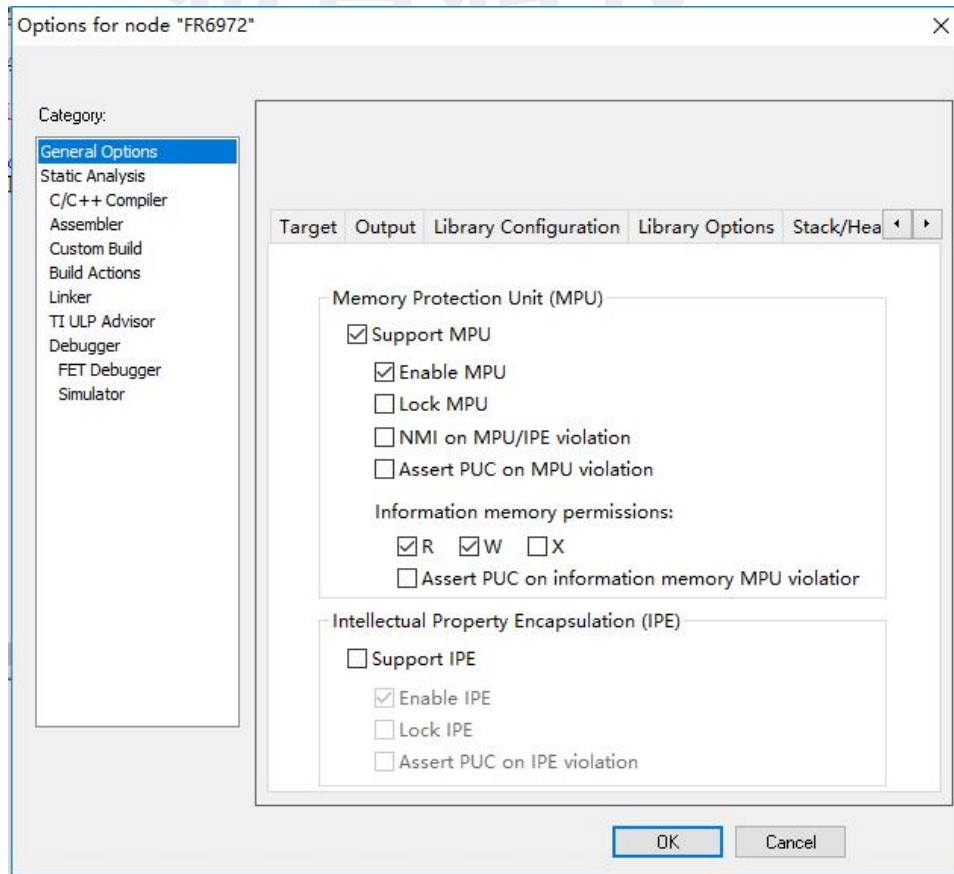


```
void Mpu_Setting(void)
{
    //MPU 模块使能，并且密码要正确
    MPUCTL0 = MPUPW;
    MPUSEGB1 = 0x04C0;           \\B1 分界地址右移 4 位
    MPUSEGB2 = 0x0F80;           \\B2 分界地址右移 4 位

    MPUSAM =
        |MPUSEG3RE|MPUSEG3XE  \    \\Seg3 只允许读和执行
        |MPUSEG2RE|MPUSEG2XE  \    \\Seg2 允许读和执行
    |MPUSEG1XE|MPUSEG1RE ;       \\Seg1 允许读和执行

    //MPU 模块使能，并且密码要正确
    MPUCTL0 = MPUPW | MPUENA;
    //故意在 MPUCTL0 的高字节写错，禁止 MPU 寄存器的再次写入
    MPUCTL0_H = 0xDD;
}
```

我们在这个 example code 中主要都只放开了“读”和“执行”权限，一般“写”权限是不宜放开的，除非应用中确实需要保存掉电不丢失的数据。那最好的做法也是“先放开写权限，写完之后，立马禁止”。虽然在 IDE（比如 IAR for MSP430）总也有通过选项框来设置 MPU 的方法，**但是我们不建议使用这种方法**。以通过设置 IAR 内的 MPU 模块为例，我们能设置的项是有限的，如下图：



设置完后，编译器会自动在 Main 函数之前加入一段 mpu_init 的代码，如下图：

```
Disassembly
?cstart_begin:
__program_start:
004400 4031 2400      mov.w    #0x2400,SP
?cstart_call_mpu_init:
004404 13B0 4410      calla   #?mpu2_init
?cstart_call_main:
004408 13B0 4444      calla   #main
00440C 13B0 444E      calla   #exit
?mpu2_init:
__iar_430_mpu_init:
004410 40B2 A500 05A0  mov.w    #0xA500,&MPUCTLO
004416 40B2 0440 05A6  mov.w    #0x440,&MPUSEGB1
00441C 40B2 0440 05A4  mov.w    #0x440,&MPUSEGB2
004422 40B2 3573 05A8  mov.w    #0x3573,&MPUSAM
004428 40B2 A501 05A0  mov.w    #0xA501,&MPUCTLO
00442E 0110          reta
void __exit(int status)
```

从反汇编内我们可以看到，我们很难从 IDE 上直接去设置如何分段，实际上要结合 xcl 文件一块才能对 Main Memory 进行分段操作。如果使用默认 xcl 文件，IAR 编译器实际使用的分界地址都是 Main Memory 的首地址。但是，既要改 xcl 文件，又要去设置，还不如用户直接软件设置来的简洁、明了、直接。

另外，我们在看 **MOV.W #0x3573, &MPUSAM**

9.7.5 MPUSAM Register

Memory Protection Unit Segmentation Access Management Register

Figure 9-11. MPUSAM Register

15	14	13	12	11	10	9	8
MPUSEG1VS	MPUSEG1XE	MPUSEG1WE	MPUSEG1RE	MPUSEG3VS	MPUSEG3XE	MPUSEG3WE	MPUSEG3RE
rw-[0]	rw-[1]	rw-[1]	rw-[1]	rw-[0]	rw-[1]	rw-[1]	rw-[1]
7	6	5	4	3	2	1	0
MPUSEG2VS	MPUSEG2XE	MPUSEG2WE	MPUSEG2RE	MPUSEG1VS	MPUSEG1XE	MPUSEG1WE	MPUSEG1RE
rw-[0]	rw-[1]	rw-[1]	rw-[1]	rw-[0]	rw-[1]	rw-[1]	rw-[1]

结合寄存器的具体信息内容我们可以看到，实际上 **0x3573** 这个设置把 SEG1 和 SEG2 的写权限都是放开，也就意味着它不能有效保护数据不被“篡改”。**Really?**

这里我们插进一个话题：如果段的分界地址，B1 和 B2 设置相同，且不等于 Main Memory 的首地址会如何？

我们已经做过测试，如果 B1 和 B2 设置相同，且不等于 Main Memory 的首地址，那么实际只存在 SEG1 和 SEG3。所以你只能设置 SEG1 和 SEG3 的权限。SEG2 实际就被“忽略”掉了。

如果分界地址 B1 和 B2 相同，并且还等于 Main Memory 的首地址，那么实际只存在 SEG3 是有效的，所以这里 IAR 编译器生成如下代码的结果是：整个 Main Memory 都属于 SEG3，且只放开了“读”和“执行”两个权限，“写”权限是被禁止的。

```
MOV.W    #0x440,&MPUSEGB1  
MOV.W    #0x440,&MPUSEGB2  
MOV.W    #0x3573,&MPUSAM
```

但是，我们依然强烈推荐用户自己软件设置 MPU 来满足你的应用需求，不要偷懒选择使用开发软件本身的设置方式。原因主要如下：

MSP430 的常见开发环境有 IAR、CCS、AQ430， 你需要每个都去试验，需要对开发环境较熟。即使同一个 IDE 开发环境，我们也难保证其编译规则会不会更新变化。至少 IAR 版本升级时，曾出现过函数形参调用规则的变化，导致一大部分 C 和汇编混合开发的程序不能运行。如果是全部依靠编程者自己软件写入，至少不会因为 IDE 问题让后期可能的出现的变数成为一种隐性危害，而且 Programmer 对于 Main Memory 的分段和权限，也更直观和可控。

希賢科技
Serial Success