Application Report

MSP430™ FRAM 技术 – 操作方法和最佳实践

William Goh, Andreas Dannenberg, Johnson He

摘要

FRAM 是一种非易失性存储器技术，其行为与 SRAM 类似，不仅支持大量新应用，还改变了固件的设计方式。该应用报告从嵌入式软件开发视角概述了 FRAM 技术在 MSP430 中的使用方法和最佳实践，其中讨论了如何根据应用特定的代码、常量和数据空间要求来实施存储器布局，如何使用 FRAM 来优化应用能耗以及如何使用存储器保护单元 (MPU) 为程序代码提供意外写访问保护，从而更大限度提高应用的稳健性。

内容

插图清单

表格清单

商标

1 FRAM 和通用存储器

FRAM 是一种非易失性存储器技术，具备独特的灵活性，可用于程序或数据存储器。可以逐位向 FRAM 写入数据，它拥有几乎不受限的写入周期（10^{15} 个周期 - 请参阅器件特定数据表）。若要了解关于 FRAM 的更多信息，请访问 www.ti.com/fram 并参阅《*MSP430™ FRAM 质量和可靠性*》。

2 可以将其视为 RAM

与 SRAM 类似，FRAM 也具有几乎不受限的写入寿命，且其性能不会出现任何下降。FRAM 不需要预擦除，每次向 FRAM 写入的数据都是非易失性数据。不过，在某些用例中，使用 FRAM（而非 RAM）时需要付出些微代价。其中一个代价是，针对 MSP430™ 平台，FRAM 访问速度被限制在 8MHz，而 SRAM 访问速度可以达到器件的最大工作频率。如果 CPU 以 8MHz 以上的速度访问 FRAM，则会进入等待状态。另一个代价是，访问 FRAM 时所需的功耗要比 SRAM 稍高一些。有关详细信息，请参阅器件特定数据表。

3 存储器布局分区

因为 FRAM 存储器可用作程序代码、变量、常量、栈等的通用存储器，所以必须针对应用对该存储器进行分区。Code Composer Studio™ 和适用于 MSP430 IDE 的 IAR Embedded Workbench® 都可用于设置应用的存储器布局，以便根据应用需求尽可能充分利用底层 FRAM。这些存储器分区方案通常位于特定于 IDE 的链接器命令文件内部。默认情况下，链接器命令文件通常会将变量和栈分配到 SRAM 中。此外，会将程序代码和常量分配到 FRAM 中。可以根据应用需求移动或按大小排列这些存储器分区。有关详细信息，请参阅节 3.4。

在 MSP430 MCU 中，可以根据所用数据的大小定义合理的数据类型。表 3-1 列出了常用的数据类型。更多详细信息，请参阅《MSP430 优化 C/C++ 编译器》。

表 3-1. MSP430 C/C++ 数据类型

类型	大小	对齐	表示	范围	
				最小值	最大值
signed char	8 位	8	二进制	-128	127
char	8 位	8	ASCII	0 或 -128	255 或 127
unsigned char	8 位	8	二进制	0	255
bool(C99)	8 位	8	二进制	0（假）	1（真）
short、signed short	16 位	16	二进制	-3 2768	32 767
unsigned short	16 位	16	二进制	0	65 535
int、signed int	16 位	16	二进制	-3 2768	32 767
unsigned int	16 位	16	二进制	0	65 535
long、signed long	32 位	16	二进制	-2 147 483 648	2 147 483 647
unsigned long	32 位	16	二进制	0	4 294 967 295
long long、signed long long	64 位	16	二进制	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 位	16	二进制	0	18 446 744 073 709 551 615
float	32 位	16	IEEE 32 位	1.175 494e-38	3.40 282 346e+38
double	64 位	16	IEEE 64 位	2.22 507 385e-308	1.79 769 313e+308
long double	64 位	16	IEEE 64 位	2.22 507 385e-308	1.79 769 313e+308

3.1 程序代码和常量数据

应以与闪存中类似的方式在 FRAM 中分配程序代码和常量数据。此外，为了确保实现最高的稳健性和数据完整性，应当为这些区域启用 MPU 功能，从而为其提供写入访问保护。这样有助于防止由于程序出现故障（软件崩溃）、缓冲区溢出、指针损坏和其他类型的异常而对这些存储器区域进行错误的写入访问，进而意外修改数据。

3.2 变量

变量由默认的链接器命令文件分配到 SRAM 中。基于 FRAM 的 MSP430 器件通常具有 0.5KB 到 8KB 的 SRAM。有关确切的规格，请参阅器件特定数据表。如果变量由于过大而不适用于 SRAM，则可以修改链接器命令文件或者使用 C 语言 `#pragma` 指令将特定的变量或结构分配到 FRAM 存储器中。节 3.4 所示为修改链接器命

令文件以将变量从 SRAM 移到 FRAM 的方法示例。除了 SRAM 存储器限制以外，应当为变量使用 FRAM 的另一个原因是缩短启动时间，如节 4 中所述。

3.3 软件栈

尽管在典型的应用中可以对栈使用 FRAM，但建议将栈分配到片上 SRAM 中。CPU 可随时全速访问 SRAM，而且无论选择什么样的 CPU 时钟频率 (MCLK)，都不会出现等待状态。由于在大多数应用中，栈是最为频繁访问的存储器区域，这一特性有助于确保实现极高的应用性能。同样，访问 SRAM 存储器所需的功耗小于 FRAM 写入访问的功耗，因此将栈分配到 SRAM 中也可以降低工作功耗。最后但同样重要的是，在大多数乃至所有用例中，下电上电时无需保留栈的内容，这是因为应用代码总是会执行冷启动并重新初始化基本 C 运行时上下文。

3.4 支持在 MSP430 IDE 中进行存储器分区

可用于 MSP430 的工具链都附带链接器命令文件，这些文件定义了默认的存储器设置和分区，而且通常将程序代码和常量数据分配到 FRAM 中，将变量数据和系统堆栈分配到 SRAM 中。此外，还提供了 C 编译器语言扩展，以使您能够按照节 5 中的说明将所选的变量和数据结构放到 FRAM 中，并利用 FRAM 的优势实现持续数据存储，而无需再考虑对存储器进行分区或对链接器命令文件进行修改。

FRAM 同样能够存储代码、常量数据和变量数据，因此通常可以让链接器来完成对存储器进行分区这一任务。例如，如果按照节 5 中的说明利用编译器扩展来将应用数据分配到 FRAM 中，则会自动从可供程序代码使用的空间中适当扣除变量占用的空间，以使链接器能够将其全部输出段放到 FRAM 中的同一个“池”中。

不过，有些应用用例可能需要更高水平的定制。例如，您可能希望将链接器的某些段强制放入存储器的特定固定区域中，以便能够更轻松地手动设置 MPU 模块。而另一个应用用例可能希望将某些变量分配到为了以非易失性方式存储数据而专门保留的存储器区域中，以使这些变量即使在系统内的固件更新之后也可供使用。此外，某个应用所需的数据大小或栈大小可能超过了片上 SRAM 的大小，并希望将链接器的相应段分配到 FRAM 中，以确保有大量的可用存储空间。

要定制存储器分区，通常需要修改特定工程的链接器命令文件，此文件是基于 IDE 附带的默认文件而生成的。虽然有些更改看起来非常简单、直观，但强烈建议查阅链接器文档，以熟练掌握链接器及其命令文件的应用知识。本节将介绍此类定制工作的入门知识。

3.4.1 TI Code Composer Studio

每个 CCS 工程都有一个链接器命令文件 (.cmd)，它会在创建工程时填充到工程文件夹中。此文件说明了用于器件程序代码、变量、常量和栈的分配方式，还介绍了在器件中排列每个存储器段时遵循的优先级。下面列出的段名称是大多数应用最常用的项目。

```
    .const    /* 常量数据                                                    */
    .text     /* 应用代码                                                 */
    .bss      /* 未初始化的全局和静态变量 - RAM 中的默认值   */
    .data     /* 初始化的全局和静态变量 - RAM 中的默认值     */
    .stack    /* 软件系统堆栈 - RAM 中的默认值                          */
```

此外，当为 FRAM 分配了名为 .TI.noinit（将与 #pragma NOINIT 配合使用）和 .TI.persistent（将与 #pragma PERSISTENT 配合使用）的链接器段时，编译器还能够自动将所选的变量、数组或结构放到 FRAM 中。对于 .TI.persistent，此定义已经存在于链接器命令文件中，并将声明为 #pragma PERSISTENT 的变量放到 FRAM 中。对于 .TI.noinit，如果应使用 #pragma NOINIT 功能将不需要执行 C 启动初始化的变量和结构放到 FRAM 中，则客户可以像对现有的 .TI.persistent 那样执行此类分配。

```
    .TI.noinit     : {} > FRAM         /* 对于 #pragma NOINIT                */
    .TI.persistent : {} > FRAM         /* 对于 #pragma PERSISTENT            */
```

可以在 CCS 中直接修改链接器命令文件。以下代码提供适用于 MSP430FR2311 的 .cmd 文件的解释和修改示例。

在链接器文件中，存储器分配包含两项内容：MEMORY 和 SECTIONS。MEMORY 主要将存储器划分为 RAM、FRAM、BSL 或用户定义的区域，而 SECTIONS 为每个使用的段选择定义的存储器段。更多详细信息，请参阅《MSP430 优化 C/C++ 编译器》。

表 3-2. MSP430FR2311 存储器组织

	访问	MSP430FR2311
存储器 (FRAM) 主存储器：中断向量和签名 主存储器：代码存储器	读/写 (可选写保护)	3.75KB FFFFh 至 FF80h FFFFh 至 F100h
RAM	读/写	1KB 23FFh 至 2000h
引导加载程序 (BSL1) 存储器 (ROM) (TI 内部使用)	只读	2KB 17FFh 至 1000h
引导加载程序 (BSL2) 存储器 (ROM) (TI 内部使用)	只读	1KB FFFFFh 至 FFC00h
外设	读/写	4KB 0FFFh 至 0000h

表 3-2 显示 MSP430FR2311 的存储器组织情况，以下程序是默认的存储器和段分配，具体取决于 CCS 中芯片的存储器的组织情况。在存储器分配中，需要使用关键字“origin”和“length”来定义存储器的初始地址和长度。SECTIONS 可以直接指向已分配的 MEMORY，用户可以根据需要定义 SECTIONS 的 MEMORY 大小和 MEMORY 类型。

```
/*************************************************************************/
/* 指定系统内存映射                                                      */
/*************************************************************************/
 MEMORY {
    BSL0                    : origin = 0x1000, length = 0x800
    RAM                     : origin = 0x2000, length = 0x400
    FRAM                    : origin = 0xF100, length = 0xE80
    BSL1                    : origin = 0xFFC00, length = 0x400
    JTAGSIGNATURE           : origin = 0xFF80, length = 0x0004, fill = 0xFFFF
    BSLSIGNATURE            : origin = 0xFF84, length = 0x0004, fill = 0xFFFF
    INT00                   : origin = 0xFF88, length = 0x0002
    INT01                   : origin = 0xFF8A, length = 0x0002
    ...........................................................
    INT57                   : origin = 0xFFFA, length = 0x0002
    INT58                   : origin = 0xFFFC, length = 0x0002
    RESET                   : origin = 0xFFFE, length = 0x0002
  }

/*************************************************************************/
/* 指定 SECTIONS 分配至存储器                                            */
/*************************************************************************/
 SECTIONS {
  GROUP(ALL_FRAM)
  {
    GROUP(READ_WRITE_MEMORY)
    {
      .TI.persistent : {}               /* 对于 #pragma persistent            */
    }
    GROUP(READ_ONLY_MEMORY)
    {
      .cinit       : {}                 /* 初始化表              */
      .pinit       : {}                 /* C++ 构建函数表            */
      .binit       : {}                 /* 引导时初始化表    */
      .init_array  : {}                 /* C++ 构造函数表            */
      .mspabi.exidx : {}                /* C++ 构造函数表            */
      .mspabi.extab : {}                /* C++ 构造函数表            */
      .const       : {}                 /* 常量数据                      */
    }
    GROUP(EXECUTABLE_MEMORY)
    {
      .text        : {}                 /* 代码                               */
      .text:_isr   : {}                 /* 代码 ISR                          */
     }
  } > FRAM

  #ifdef __TI_COMPILER_VERSION__
    #if __TI_COMPILER_VERSION__ >= 15009000
      .TI.ramfunc : {} load=FRAM, run=RAM, table(BINIT)
    #endif
  #endif
  .jtagsignature      : {} > JTAGSIGNATURE
```

```
   .bslsignature       : {} > BSLSIGNATURE
   .cio         : {} > RAM                  /* C I/O 缓冲器                        */
   .sysmem      : {} > RAM                  /* 动态存储器分配区     */
   .bss         : {} > RAM                  /* 全局和静态变量              */
   .data        : {} > RAM                  /* 全局和静态变量              */
   .TI.noinit   : {} > RAM                  /* 对于 #pragma noinit                */
   .stack       : {} > RAM (HIGH)           /* 软件系统堆栈             */

    /* MSP430 中断矢量 */
    .int00       : {}                > INT00
    .int01       : {}                > INT01
    ....................................
    .int43       : {}                > INT43
    .int44       : {}                > INT44
    ECOMP0       : { * ( .int45 ) } > INT45 type = VECT_INIT
    PORT2        : { * ( .int46 ) } > INT46 type = VECT_INIT
    PORT1        : { * ( .int47 ) } > INT47 type = VECT_INIT
    ADC          : { * ( .int48 ) } > INT48 type = VECT_INIT
    EUSCI_B0     : { * ( .int49 ) } > INT49 type = VECT_INIT
    EUSCI_A0     : { * ( .int50 ) } > INT50 type = VECT_INIT
    WDT          : { * ( .int51 ) } > INT51 type = VECT_INIT
    RTC          : { * ( .int52 ) } > INT52 type = VECT_INIT
    TIMER1_B1    : { * ( .int53 ) } > INT53 type = VECT_INIT
    TIMER1_B0    : { * ( .int54 ) } > INT54 type = VECT_INIT
    TIMER0_B1    : { * ( .int55 ) } > INT55 type = VECT_INIT
    TIMER0_B0    : { * ( .int56 ) } > INT56 type = VECT_INIT
    UNMI         : { * ( .int57 ) } > INT57 type = VECT_INIT
    SYSNMI       : { * ( .int58 ) } > INT58 type = VECT_INIT
    .reset       : {}                > RESET
}
```

3.4.1.1 修改链接器文件示例 1

根据节 2 的说明，FRAM 可用作 RAM，本例分配 0.5KB 的 FRAM 作为 RAM。

FRAM 存储器段在以下程序中被定义为 RAM2。

```
    RAM                     : origin = 0x2000, length = 0x400
    RAM2                    : origin = 0xF100, length = 0x200
    FRAM                    : origin = 0xF300, length = 0xC80
```

可在 SECTIONS 中执行“|”操作以将 .bss 和 .data 段分配到 RAM 或 RAM2 存储器中。

```
   .bss         : {} > RAM | RAM2           /* 全局和静态变量              */
   .data        : {} > RAM | RAM2           /* 全局和静态变量              */
```

NOTE

如果在分配的 FRAM 存储器中有一些写入操作，则必须在程序中启用 FRAM 的写入功能。

3.4.1.2 修改链接器文件示例 2

本例将 TI.noinit 段定义到 FRAM 存储器中，这样便可定义为一个断电时不会擦除的功能。

```
   .TI.noinit   : {} > FRAM                 /* 对于 #pragma noinit                */
```

NOTE

如果在分配的 FRAM 存储器中有一些写入操作，则必须在程序中启用 FRAM 的写入功能。

3.4.1.3 修改链接器文件示例 3

通过修改链接器文件，可以实现在指定的存储器区域中定义变量的功能。以下步骤和示例概述了如何实现此功能。

1. 分配一个新的存储块 (MY_SECTION)。
2. 定义一个段 (.Image)，此段存储在该存储块 (MY_SECTION) 中。

3. 在程序中使用 #pragma DATA_SECTION 定义此段中的变量。

```
RAM                     : origin = 0x2000, length = 0x400
RAM2                    : origin = 0xF100, length = 0x100
MY_SECTION              : origin = 0xF200, length = 0x100
FRAM                    : origin = 0xF300, length = 0xC80

.Image      : {} > MY_SECTION

#pragma DATA_SECTION(a,".Image")
unsigned char a;
```

NOTE

如果在分配的 FRAM 存储器中有一些写入操作，则必须在程序中启用 FRAM 的写入功能。

3.4.2 适用于 MSP430 的 IAR Embedded Workbench

在 IAR 中，如果按节 5.2 中所述使用 `__persistent` 属性将变量置于 FRAM 中，则很多用例将无需修改链接器命令文件 (.xcl)。但是，如果要将声明为 `__noinit` 的变量放到 FRAM 中，则可以在链接器命令文件中进行少量修改，以便将所分配的 `DATA16_N` 和 `DATA20_N` 段从 RAM 移到 FRAM 区域中，以达成这一目的。

IAR 链接器命令文件通常在所有工程之间共享，并位于 C:\<IAR installation directory>\430\config\linker\ 文件夹中。若要详细了解链接器命令文件中的每个存储器段名称，请访问 https://www.iar.com/support/user-guides/user-guidesiar-embedded-workbench-for-ti-msp430/，参阅《*IAR C/C++ 编译器用户指南*》中的“*段参考*”一章。

如果仍然需要定制的链接器命令文件，则需要为 lnk430xxxx.xcl 制作一个副本。以下步骤概述了如何创建定制的 IAR 链接器命令文件。

1. 导航到 C:\<IAR installation directory>\430\config\linker\ folder。
2. 为您本地工程的 link430xxxx.xcl 文件制作一个副本，如果需要，可以重命名此文件。
3. 打开 .xcl 文件的新副本并进行定制
4. 将 IAR 工程配置为指向定制的链接器命令文件。

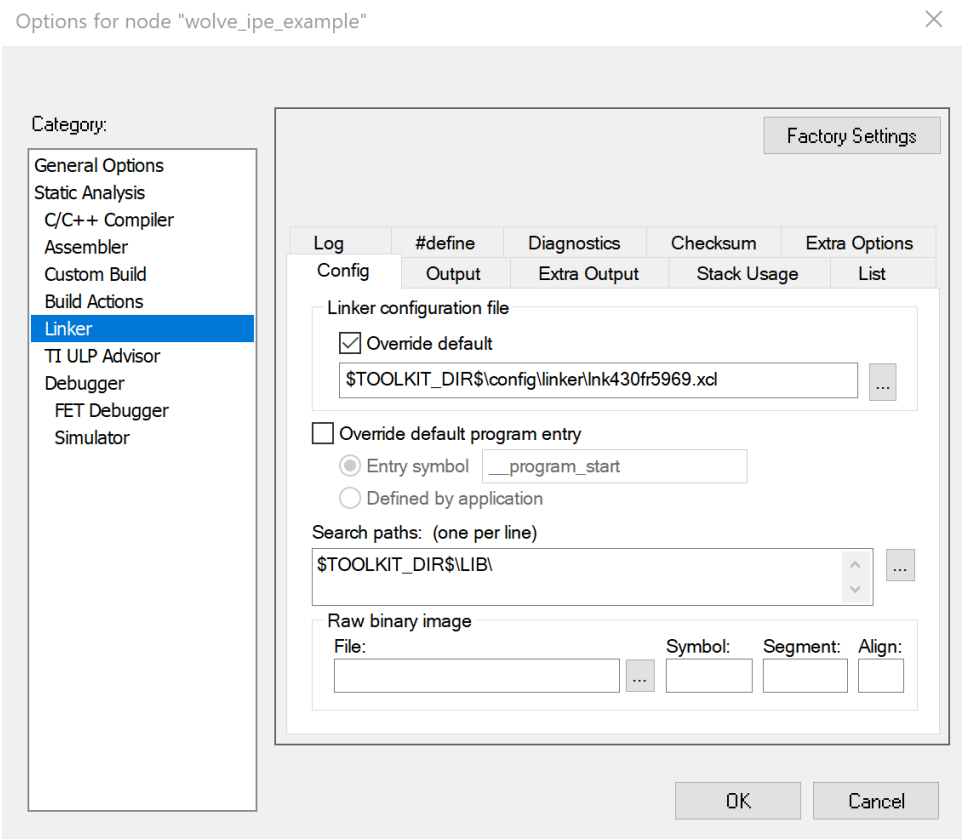

图 3-1. 为 IAR 手动覆盖链接器命令文件

可以在 IAR 中直接修改链接器命令文件。以下代码提供适用于 MSP430FR2311 的 .xcl 文件解释和修改示例。

表 3-3 显示了 IAR 中的一些段，这些段可用于分配存储器。有关更多段说明，请参阅《IAR C/C++ 编译器用户指南（适用于 MSP430）》中的*段参考*一章。

表 3-3. IAR 中的段摘要

段	说明
DATA16_AC	保留 __data16 定位的常量数据。
DATA16_AN	保留 __data16 定位的未初始化数据。
DATA16_C	保留 __data16 常量数据。
DATA16_HEAP	在 data16 存储器中保留用于动态分配数据的堆。
DATA16_I	保留 __data16 静态和全局初始化变量。
DATA16_ID	在 DATA16_I 中保存 __data16 静态和全局变量的初始值。
DATA16_N	保留 __no_init __data16 静态和全局变量。
DATA16_P	保留使用 _persistent 关键字定义的 __data16 变量。
DATA16_Z	保留初始化为 0 的 __data16 静态和全局变量。
DATA20_AC	保留 __data20 定位的常量数据。
DATA20_AN	保留 __data20 定位的未初始化数据。
DATA20_C	保留 __data20 常量数据。
DATA20_HEAP	在 data20 存储器中保留用于动态分配数据的堆。
DATA20_I	保留 __data20 静态和全局初始化变量。
DATA20_ID	在 DATA20_I 中保存 __data20 静态和全局变量的初始值。
DATA20_N	保留 __no_init __data20 静态和全局变量。
DATA20_P	保留使用 _persistent 关键字定义的 __data20 变量。

表 3-3. IAR 中的段摘要 (continued)

段	说明
DATA20_Z	保留初始化为 0 的 __data20 静态和全局变量。

存储块可以直接分配到 IAR 中的每个段，并且存储器空间可以是不连续的，用“,”符号分隔。如果在 -Z 命令之后没有给出地址空间，则默认值与前一行相同。根据表 3-2 中 MSP430FR2311 的存储器组织情况，IAR 中提供了默认存储器分配文件。主要内容如以下代码所示，分为 RAM、FRAM、矢量段等等。更多有关存储器分配的信息，请参阅《IAR C/C++ 编译器用户指南（适用于 MSP430）》。

```
// ------------------------------------------------
// RAM 存储器
//
  -Z(DATA)DATA16_I,DATA16_Z,DATA16_N,TLS16_I=2000-23FF
  -Z(DATA)CODE_I -Z(DATA)DATA20_I,DATA20_Z,DATA20_N
  -Z(DATA)CSTACK+_STACK_SIZE#

// ------------------------------------------------
// FRAM 存储器
//
// Read/write data in FRAM
  -Z(CONST)DATA16_P,DATA20_P=F100-FF7F
  -Z(DATA)DATA16_HEAP+_DATA16_HEAP_SIZE,DATA20_HEAP+_DATA20_HEAP_SIZE

// 常量数据
  -Z(CONST)DATA16_C,DATA16_ID,TLS16_ID,DIFUNCT,CHECKSUM=F100-FF7F
  -Z(CONST)DATA20_C,DATA20_ID

// 代码
  -Z(CODE)CSTART,ISR_CODE,CODE_ID=F100-FF7F
  -P(CODE)CODE,CODE16=F100-FF7F

// 特殊矢量
  -Z(CONST)JTAGSIGNATURE=FF80-FF83
  -Z(CONST)BSLSIGNATURE=FF84-FF87
  -Z(CODE)INTVEC=FF88-FFFF
  -Z(CODE)RESET=FFFE-FFFF
```

3.4.2.1 修改链接器文件示例 1

本例分配 0.5KB 的 FRAM 作为 RAM。因此，0xF100 到 0xF2FF 存储器段被添加到 RAM 中进行存储器分配，并且 FRAM 存储器段从 0xF300 开始。

```
// ------------------------------------------------
// RAM 存储器
//
  -Z(DATA)DATA16_I,DATA16_Z,DATA20_I,DATA20_Z=2000-23FF,F100-F2FF
  -Z(DATA)DATA16_N,TLS16_I=2000-23FF -Z(DATA)CODE_I -Z(DATA)DATA20_N
  -Z(DATA)CSTACK+_STACK_SIZE#

// ------------------------------------------------
// FRAM 存储器
//
// 在 FRAM 中读取/写入数据
  -Z(CONST)DATA16_P,DATA20_P=F300-FF7F
  -Z(DATA)DATA16_HEAP+_DATA16_HEAP_SIZE,DATA20_HEAP+_DATA20_HEAP_SIZE

// 常量数据
  -Z(CONST)DATA16_C,DATA16_ID,TLS16_ID,DIFUNCT,CHECKSUM=F300-FF7F
  -Z(CONST)DATA20_C,DATA20_ID

// 代码
  -Z(CODE)CSTART,ISR_CODE,CODE_ID=F300-FF7F
  -P(CODE)CODE,CODE16=F300-FF7F

// 特殊矢量
  -Z(CONST)JTAGSIGNATURE=FF80-FF83
  -Z(CONST)BSLSIGNATURE=FF84-FF87
  -Z(CODE)INTVEC=FF88-FFFF
  -Z(CODE)RESET=FFFE-FFFF
```

NOTE

如果在分配的 FRAM 存储器中有一些写入操作，则必须在程序中启用 FRAM 的写入功能。

3.4.2.2 修改链接器文件示例 2

本例将 DATA16_N 和 DATA20_N 段定义到 FRAM 存储器中，可以将此存储器定义为断电时不会擦除的功能。

```
// ------------------------------------------------
// RAM 存储器
//
  -Z(DATA)DATA16_I,DATA16_Z,DATA20_I,DATA20_Z=2000-23FF,F100-F2FF
  -Z(DATA)DATA16_N,DATA20_N=F100-F2FF
  -Z(DATA)TLS16_I=2000-23FF
  -Z(DATA)CODE_I
  -Z(DATA)CSTACK+_STACK_SIZE#
```

NOTE

如果在分配的 FRAM 存储器中有一些写入操作，则必须在程序中启用 FRAM 的写入功能。

3.4.2.3 修改链接器文件示例 3

通过修改链接器文件，可以实现在指定的存储器区域中定义变量的功能。以下步骤和示例概述了如何实现此功能。

1. 定义一个新段 (MY_SEGMENT) 并分配一些存储器空间。
2. 在程序中使用“@”符号来定义变量。

```
  -Z(DATA)DATA16_I,DATA16_Z,DATA20_I,DATA20_Z=2000-23FF,F100-F1FF
  -Z(DATA)DATA16_N,DATA20_N=F100-F1FF
  -Z(DATA)MY_SEGMENT=F200-F2FF
  -Z(DATA)TLS16_I=2000-23FF
  -Z(DATA)CODE_I -Z(DATA)CSTACK+_STACK_SIZE#

  __no_init int a @ "MY_SEGMENT";
```

NOTE

如果在分配的 FRAM 存储器中有一些写入操作，则必须在程序中启用 FRAM 的写入功能。

4 优化应用能耗和性能

4.1 缩短从 LPMx.5 唤醒的时间

MSP430 上可实现超低功耗的模式是 LPM3.5 和 LPM4.5 模式，这是因为器件基本处于断电状态，可供使用的功能非常有限。

低功耗模式 (LPMx)	可用的器件功能	唤醒源
LPM3.5	RTC、32kHz 振荡器	RTC 和 GPIO 中断
LPM4.5	无	GPIO 中断

不过，从这些模式唤醒类似于脱离复位状态，需要在进入 LPMx.5 之前将应用上下文存储到非易失性存储器中，并在从 LPMx.5 唤醒器件之后将其恢复。其他微控制器具有类似的功能受限的深度睡眠模式，并可能提供一个“备用 RAM”段，此备用 RAM 会在这些模式期间保持通电状态，以帮助存储应用上下文。这些存储器的每个段通常都是非常小的（几十个字节），因此可以存储的上下文非常有限。另一方面，可以选择用闪存来存储更大量的应用上下文的另一个选项；但这样做会对应用的功耗和实时性能产生重大影响，更何况闪存通常还具有其他局限性，例如擦除和写入寿命有限。与 FRAM 相比，能够做到自动在器件的 FRAM 中存储并保留不超过 FRAM 可用大小的整个应用上下文，例如数据缓冲区、状态变量和各种标志，而且根本不需要存储或恢复数据，也不会对应用的功耗或实时行为产生任何影响。

为了在使用 LPMx.5 的应用中充分利用 FRAM，需要特别注意变量的声明和使用方式。具体来说，对于保留应用上下文（状态变量和标志、结果寄存器、应用特定的校准设置和基线值、中间计算或信号处理结果等等）的所有对象，应将变量声明为永久性变量或不初始化变量，以免在通电之后不得不重新计算或重新获得这些上下文。

作为并非 FRAM 特有的通用惯例，为了尽可能缩短从 LPMx.5 深度睡眠模式恢复之后的应用唤醒时间，还应将初始值不重要的所有变量（比如用作缓冲器的大型数组，它们可能不必被初始化为 0）声明为 no-init，这样有助于在应用启动时节省 C 自动初始化启动例程内的处理器周期，并对应用的启动时间和能耗直接产生积极影响。

5 面向 FRAM 且易于使用的编译器扩展

本节将概述如何利用内置的编译器扩展将特定的变量放到 FRAM 中，以便能够在电源循环期间或在系统完全断电的任意时限内保留这些变量的值。如节 4 中所述，利用本文中讨论的永久性或不初始化机制将变量放到 FRAM 中还有助于缩短应用唤醒时间，进而降低应用能耗，原因在于 C 启动例程不会对这些变量进行初始化。

NOTE

为了使编译器更好地支持 FRAM MCU 系列并更好地利用其内部资源，建议使用更高版本的 CCS 和 IAR 开发软件。强烈建议使用 CCS 版本 9.0.1 和 IAR 版本 7.12.13。

5.1 TI Code Composer Studio

CCS 中具有两个可以使用的 C 语言 pragma 语句：`#pragma PERSISTENT` 和 `#pragma NOINIT`。在使用这些 pragma 语句之前，请参阅节 3.4.1，了解链接器命令文件要求。有关这些 pragma 指令的更多详细信息，请参阅《*MSP430 优化 C/C++ 编译器用户指南*》。

`PERSISTENT` 会导致变量不会被 C 启动例程初始化，而是在首次将应用代码加载到目标器件中时被调试工具链初始化。因此，在某些情况下（例如下电上电之后）这些变量不会初始化，因为它们已被彻底排除在 C 启动初始化过程之外。将变量声明为 `PERSISTENT` 会将它们分配到 `.TI.persistent` 链接器存储器段中。

下面这个代码片段显示了变量如何被声明为 persistent（永久性）：

```
#pragma PERSISTENT(x)
unsigned int x = 5;
```

`NOINIT` 的工作方式与 `PERSISTENT` 类似，但这些变量起初从未被工程的二进制映像文件初始化，也不会在代码下载期间被调试工具链初始化。将变量声明 为 `NOINIT` 会将它们分配到 `.TI.noinit` 链接器存储器段中。请注

意，与 PERSISTENT 不同，声明为 NOINIT 的变量不会被默认的链接器命令文件分配到 FRAM 中，因此当需要此功能时，需要对链接器命令文件进行少量修改。下面是相应的代码片段：

```
#pragma NOINIT(x)
unsigned int x;
```

5.2 适用于 MSP430 的 IAR Embedded Workbench

IAR 提供了两个分别名为 __persistent 和 __no_init 的 C 语言扩展属性，便于使用 FRAM 来存储数据。若要了解有关这两个属性的更多信息，请参阅 https://www.iar.com/support/user-guides/user-guidesiar-embedded-workbench-for-ti-msp430/ 上的《*IAR C/C++ 编译器用户指南*》。

对于 IAR 中的永久性存储功能，可以使用 __persistent 属性来声明变量。使用此属性声明的变量将分配到 DATA16_P 和 DATA20_P 链接器存储器段中，默认的 IAR 链接器命令文件 (.xcl) 会自动将这些存储器段放到 FRAM 中。如下示例显示了被声明的变量 x，此变量在 C 启动期间不会初始化，并且会自动分配到 FRAM 存储器中。此外，与 CCS 中的行为类似，此变量只会在下载初始代码时被调试工具链初始化，而不会在应用启动或运行时初始化。

```
__persistent unsigned int x = 5;
```

同样，在 IAR 中也可以利用 __no_init 属性来获得不初始化存储功能。使用此属性来声明变量会将它们分配到 DATA16_N 和 DATA20_N 链接器存储器段中。同样也不同于 __persistent 的是，被声明为 __no_init 的变量默认不会分配到 FRAM 中。如果需要使用此类功能， 需要对链接器命令文件进行少量修改。

```
__no_init unsigned int x;
```

6 FRAM 保护和安全性

6.1 存储器保护

FRAM 易于写入。需要为应用代码、常量和驻留在 FRAM 中的部分变量提供保护，以防止由于无效的指针访问、缓冲器溢出和可能导致应用崩溃的其他异常而造成意外写入。某些 MSP430 FRAM 器件具有一个内置的 MPU，其作用是监视和监督软件中定义的存储器段，以使它们受到读保护、写保护、执行保护或者这几种保护的组合，而其他此类器件具有可控制 FRAM 存储器写保护的寄存器。

NOTE

非常重要的是，在部署任何软件或发布生产代码之前，必须始终适当配置并启用存储器保护功能，以确保实现极高的应用稳健性和数据完整性。在开始执行 C 启动例程时甚至是进入 *main()* 例程之前，应在器件开始执行来自通电或复位的代码之后尽早启用存储器保护。

在保护存储器之前，需要对 FRAM 存储器进行分区。为了进行分区，需要了解执行程序链接之后的程序大小和存储器段类型，这样才能决定保护各个存储器段的方法。这些信息通常位于在构建应用时生成的工程映射文件中，并填充到特定 IDE 的输出文件夹中。

对于 FR5xx 和 FR6xx 系列的 MCU，MPU 能够保护变量、常量和程序代码。配置可以由 MPU 自动执行，也可以手动执行以获得极高的灵活性。在继续了解本节的内容之前，建议首先阅读《*MSP430FR58xx、MSP430FR59xx 和 MSP430FR6xx 系列用户指南*》或《*MSP430FR57xx 系列用户指南*》中的*存储器保护单元*一章。

对于 FR2xx 和 FR4xx 系列的 MCU，使用 SYSCFG0 寄存器实现 FRAM 的写保护。在继续了解本节的内容之前，建议首先阅读《*MSP430FR4xx 和 MSP430FR2xx 系列用户指南*》中的*系统复位、中断和操作模式，系统控制模块 (SYS)* 一章。

6.2 检查链接器映射文件

第一步是分析链接器所生成的映射文件，以确定构成应用固件映像的存储器段的起始地址和大小、常量段、变量段、不初始化段、永久性段和程序代码段。

6.2.1 TI Code Composer Studio

在映射文件中，查找 `.bss`、`.data`、`.TI.noinit`、`.TI.persistent`、`.const` 和 `.text` 存储器起始地址及其大小。这些信息可用于确定如何手动配置 MPU。请注意，表 6-1 展示了所有这些段的起始地址。

表 6-1. CCS 映射文件内的存储器分段

段名称	存储器区域	建议的保护类型（如果是在 FRAM 中）
.bss/.data	变量	读取和写入
.TI.noinit	使用 `#pragma NOINIT` 定义的数据	读取和写入
.TI.persistent	使用 `#pragma PERSISTENT` 定义的数据	读取和写入
.sysmem	“malloc”和“free”使用的堆	读取和写入
.const	常量	只读
.text	程序代码	读取和执行

6.2.2 适用于 MSP430 的 IAR Embedded Workbench

IAR 默认不生成映射文件。需要如图 6-1 中所示，选中“Project Options”下的 *Generate linker listing* 框，以启用此功能。

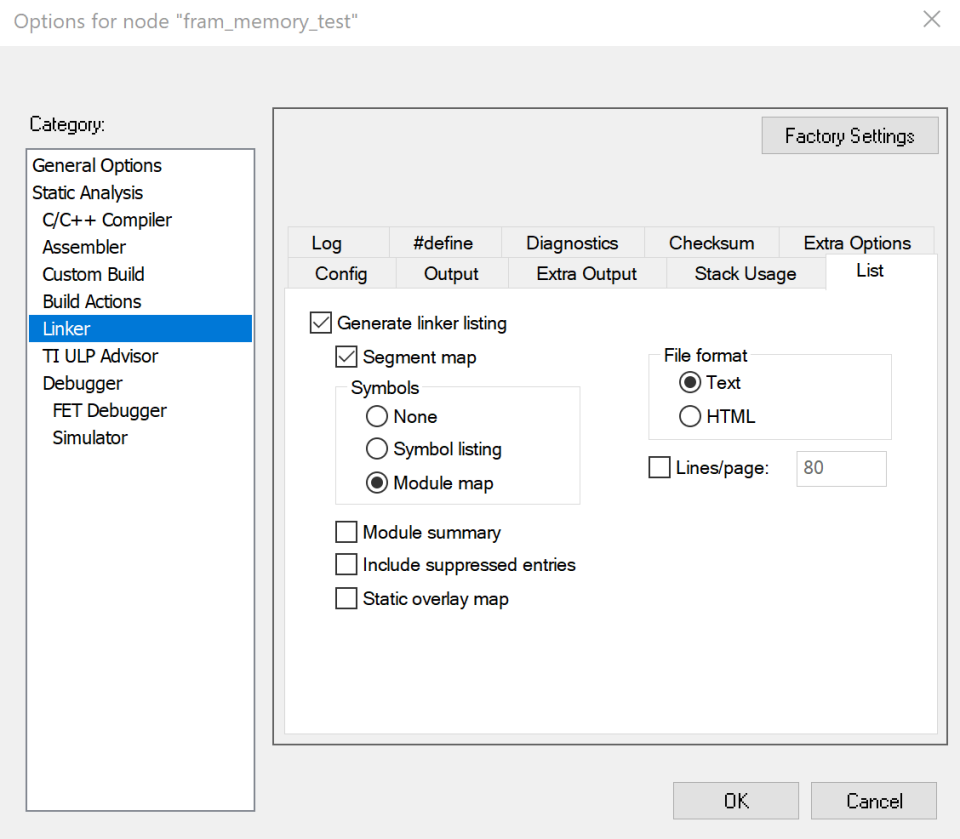

图 6-1. 用于启用映射文件的 IAR 工程选项

启用之后，应将成功编译好的映射文件放到工程中。打开映射，并为以下段名称分析此文件。

表 3-3 展示了 IAR 使用的几个通用段名称。

6.3 存储器保护设置

6.3.1 FR5xx 和 FR6xx 系列 MCU MPU 配置

MPU 可配置为保护软件中的三个不同存储器段，每个段可以单独配置为读、写、执行或三者的组合。大多数应用都具有某种形式的应当受到读写保护的变量、只读的常量以及只能读取和执行的程序代码。表 6-2 总结了典型的存储器分段。

表 6-2. MPU 存储器分段

存储器区域	保护类型	MPU 段
变量	读取和写入	段 1
不初始化	读取和写入	段 1
永久性	读取和写入	段 1
常量	只读	段 2
程序代码	读取和执行	段 3

在节 6.2 中生成的映射文件中为应用的读取和写入、只读以及读取和执行段找到起始地址之后，下一步是为 MPU 确定和配置段边界。请记住，最小的 MPU 段大小分配单位为 1KB 或 0x0400。如需更多信息，请参阅特定器件系列用户指南。在该示例中，应用只使用 5 个字节的常量数组，其中 2 个字节用于永久性变量，其余字节用于应用代码。因此，在分配此示例应用时，链接器应当为变量分配 1KB，为常量分配 1KB，如表 6-3 中所示。

表 6-3. MPU 存储器分段示例

存储器区域	保护类型	MPU 段	存储器分区示例
变量	读取和写入	段 1	0x4400 到 0x47FF
常量	只读	段 2	0x4800 到 0x4BFF
程序代码	读取和执行	段 3	0x4C00 到 0xYYYYY

按照表 6-3 中所示为段 1、2 和 3 确定了存储器分段之后，可以使用两个寄存器来定义如何配置段边界：存储器保护单元分段边界 1 (MPUSEGB1) 和存储器保护单元分段边界 2 寄存器 (MPUSEGB2)。在向寄存器写入之前，需要将地址右移 4 位。

图 6-2 所示为有关如何对存储器进行分区和对 MPU 寄存器进行配置的应用示例。

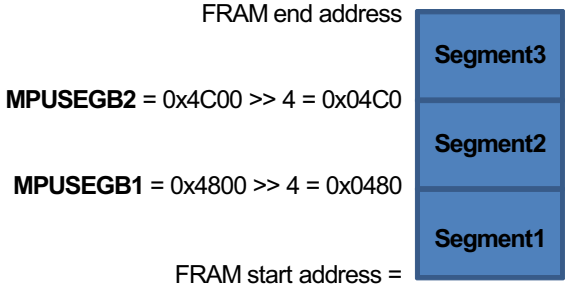

图 6-2. 要写入到 MPUSEGBx 寄存器中的地址

现在需要将配置实施到代码中。如节 6.1 中所述，应在器件启动过程中尽早确定 MPU 配置。若要进行此配置，请分别参阅节 6.3.1.1 和节 6.3.1.2 中概述的 CCS 和 IAR 配置步骤。

6.3.1.1 CCS MPU 实现方式

6.3.1.1.1 手动配置 MPU

创建一个名为 *system_pre_init.c* 的新 C 文件，并将其添加到工程中。接下来，在此文件内放置一个 `int _system_pre_init(void)` 函数。如果此函数是工程的一部分，则工具链将确保在任何 C 启动初始化例程之前以及代码执行过程转移到 `main()` 之前首先执行此函数。此函数通常用于需要在器件启动之后尽早执行的关键例程。

下面是用于启用 MPU 的代码片段示例。基于应用，根据需要确定 MPU 配置。

```
#include <msp430.h>

int _system_pre_init(void)
{
  /* 在此插入低级初始化 */

  /* 在长整型变量初始化序列期间禁用看门狗计时器 */
  /* 以防止复位。*/
  WDTCTL = WDTPW | WDTHOLD;

  // 配置 MPU
  MPUCTL0 = MPUPW;                          // 写入 PWD 以访问 MPU 寄存器
  MPUSEGB1 = 0x0480;                        // B1 = 0x4800; B2 = 0x4C00
  MPUSEGB2 = 0x04c0;                        // 边界已分配至段
  // 段 1 – 允许只读和只写
  // 段 2 – 允许只读
  // 段 3 – 允许只读和只执行
  MPUSAM = (MPUSEG1WE | MPUSEG1RE | MPUSEG2RE | MPUSEG3RE | MPUSEG3XE);
  MPUCTL0 = MPUPW | MPUENA | MPUSEGIE;      // 启用 MPU 保护
                                            // 在 BOR 之前锁定 MPU 寄存器

  /*==================================*/
  /* 选择应执行或不执行 */
  /* 段初始化。              */
  /* 返回：0 则省略初始化 */
  /* 1 则执行初始化          */
  /*==================================*/
```

```
    return 1;
}
```

正确配置之后，_system_pre_init() 函数应当在进入主存储器之前执行完毕。进入 main() 时，观察 CCS 调试器的寄存器视图（图 6-3），其中显示了正在配置的 MPU 寄存器内容。

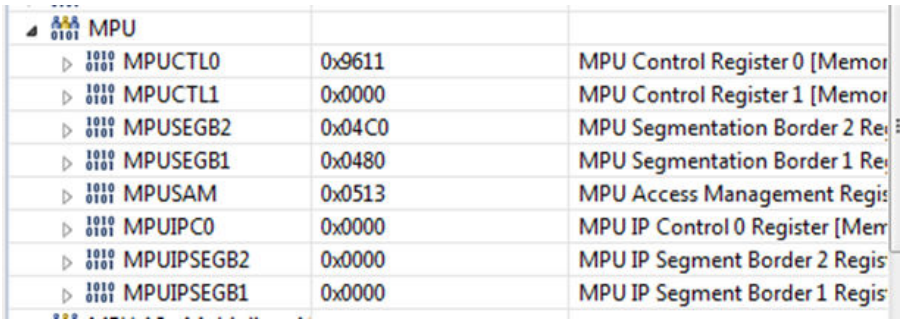

图 6-3. 寄存器窗口表明已启用 MPU

6.3.1.1.2 基于 IDE 向导的 MPU 配置

图 6-4 所示为 Code Composer Studio v9 中的内置 MSP430 MPU 向导，可通过“CCS Project Properties”访问此向导。若要打开此对话框，请在 CCS 的“Project Explorer”视图中右键单击工程，然后选择 *Properties*。

选中 *Enable Memory Protection Unit (MPU)* 框，以启用 MPU。然后，应将配置保留为默认值，以允许编译器基于应用的使用情况自动配置存储器区域并对其进行分区。例如，将常量配置为只读或将程序代码配置为只读取和执行。也可以使用手动配置模式进行更详细的配置。

通过 MPU 向导进行配置时，C 启动例程会在进入 main() 之前自动配置并启用 MPU，您无需执行任何额外的操作。

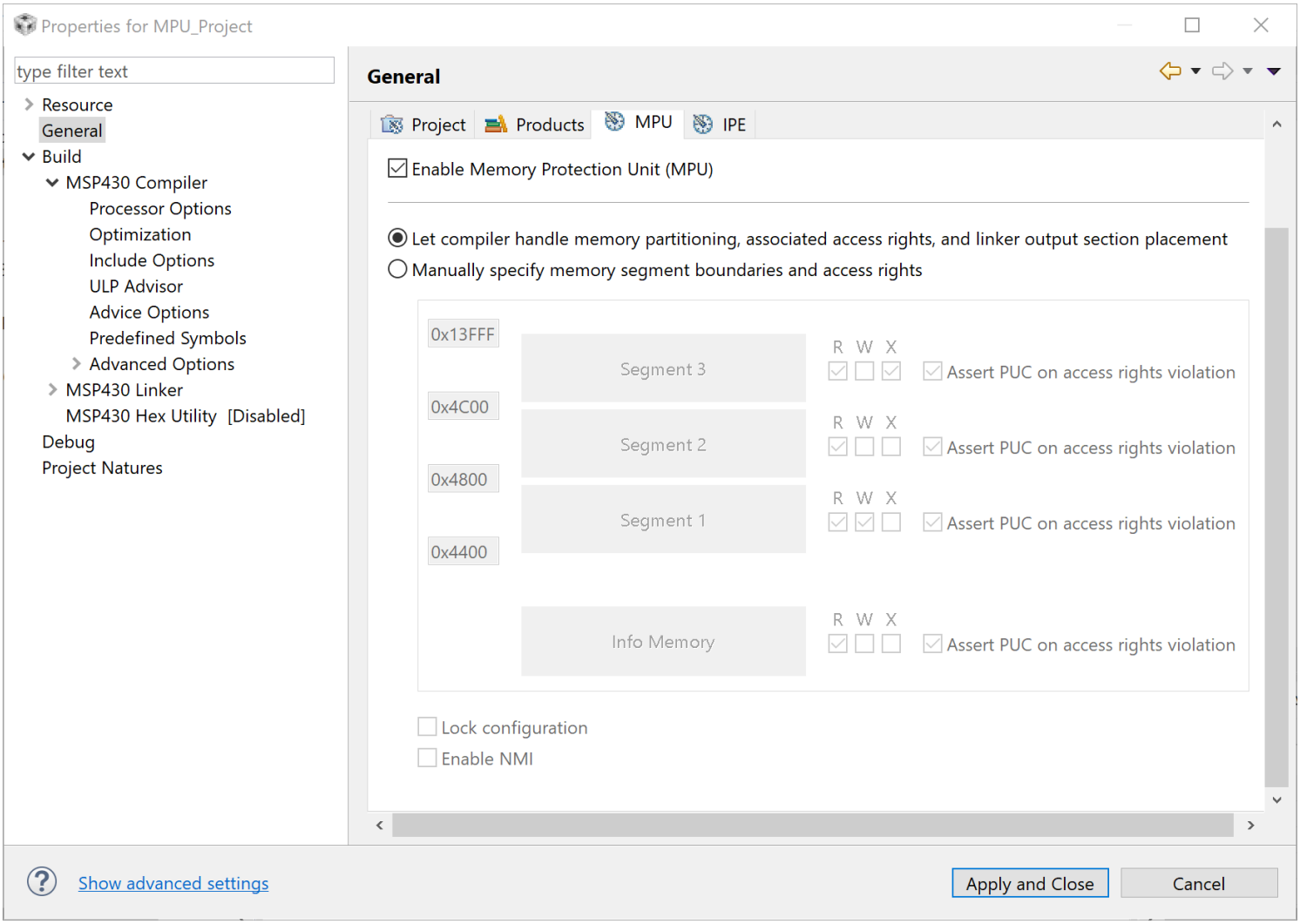

图 6-4. “CCS Project Properties”下的 MPU 向导

6.3.1.2 IAR MPU 实现方式

6.3.1.2.1 手动配置 MPU

若要在 IAR 中执行同样的操作，必须创建一个名为 *low_level_init.c* 的新 C 文件。需要将此文件包含到您的工程中。IAR 中用于在器件启动之后立即启用应用代码执行过程的等效函数是 `int __low_level_init(void)`。如下代码片段示例显示了 IAR 的等效 MPU 配置。

```
#include "msp430.h"

int __low_level_init(void)
{
  /* 在此处插入低级初始化 */

  WDTCTL = WDTPW+WDTHOLD;

  // 配置 MPU
  MPUCTL0 = MPUPW;                          // 写入 PWD 以访问 MPU 寄存器
  MPUSEGB1 = 0x0480;                        // B1 = 0x4800; B2 = 0x4C00
  MPUSEGB2 = 0x04c0;                        // 边界已分配至段
  // 段 1 – 允许只读和只写
  // 段 2 – 允许只读
  // 段 3 – 允许只读和只执行
  MPUSAM = (MPUSEG1WE | MPUSEG1RE | MPUSEG2RE | MPUSEG3RE | MPUSEG3XE);
  MPUCTL0 = MPUPW | MPUENA | MPUSEGIE;      // 启用 MPU 保护
                                            // 在 BOR 之前锁定 MPU 寄存器
```

```
    /*
     * 返回值：
     *
     *   1 - 执行数据段初始化。
     *   0 - 跳过数据段初始化。
     */

    return 1;
}
```

6.3.1.2.2 基于 IDE 向导的 MPU 配置

IAR 中用于通过 MPU 向导配置 MPU 的 IDE 选项位于 *Project Options* → *General Options* → *MPU/IPE/FRWP* 下，如图 6-5 中所示。选中 *Support MPU* 框，以启用 MPU。启用之后，IAR 工具链会自动确定哪些段是代码、常量和变量，以决定应当如何配置 MPU 分区。

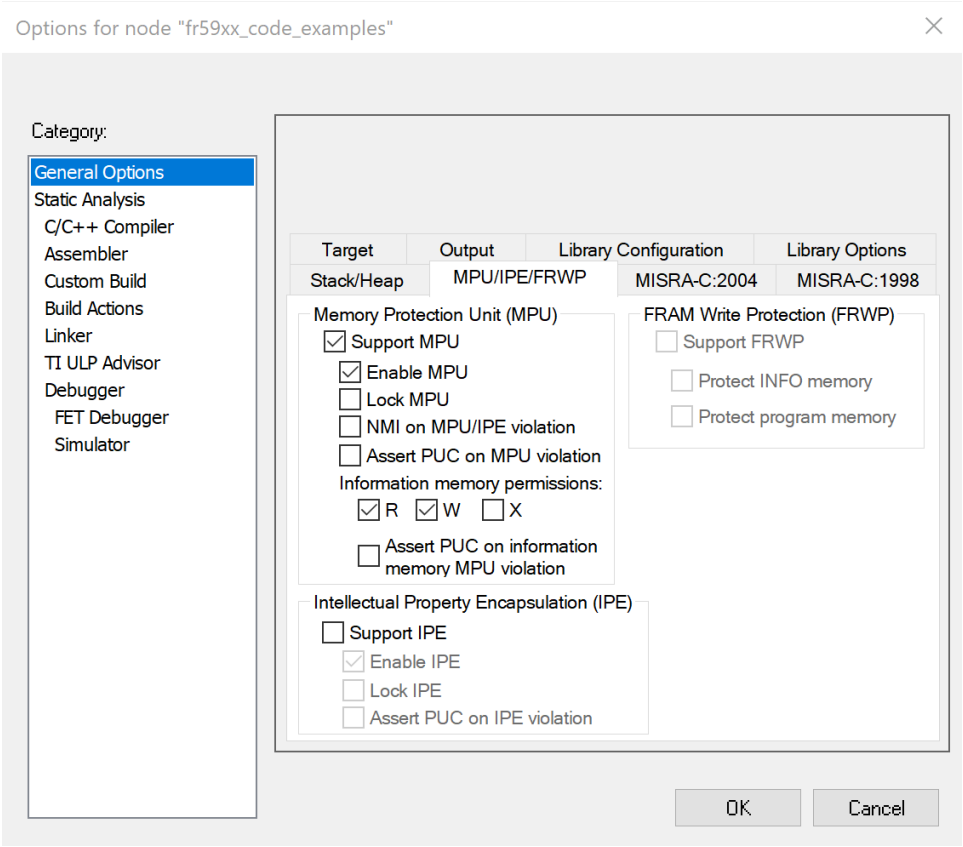

图 6-5. IAR 中的 MPU 和 IPE 向导

6.3.2 FR2xx 和 FR4xx 系列 MCU FRAM 写保护配置

对于 FR2xx FR4xx 系列 MCU，除 FR235x 和 FR215x 器件外，可以操作 PFWP 和 DFWP 寄存器，以启用或禁用 FRAM 中主存储器和信息存储器的写保护。PUC 复位之后，这些位默认为 1，而 FRAM 写访问被禁用。用户代码必须输入正确的密码并在清除相应位之后才能执行写操作。

在 FR215x 和 FR235x 器件中，程序 FRAM 可通过与主 FRAM 存储器起始地址的偏移（具体偏移量由 FRWPOA 规定）受到部分保护。设置 PFWP 后，发生偏移的主存储器受到保护，而此地址之前的部分不受保护。这一不受保护的范围可以像 RAM 一样用于随机频繁写入。6 位 FRWPOA 可以指定从 0KB 到 63KB 的偏移量，分辨率为 1KB。复位后，FRWPOA 默认为零，而整个程序 FRAM 受 PFWP 保护。正确写入 FRWPPW 时，可并行修改 FRWPOA。图 6-6 所示为如何在这些器件中保护数据和程序 FRAM。

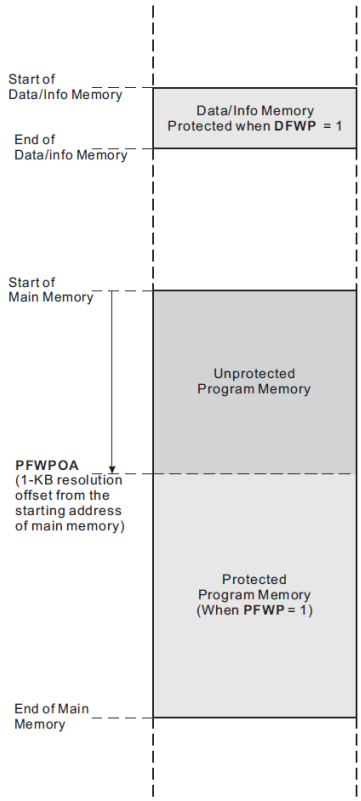

图 6-6. FR215x 和 FR235x 存储器保护

6.4 IP 封装

IP 封装 (IPE) 是诸如 MSP430FR59xx/69xx 系列的一些 MSP430 FRAM 微控制器上提供的一种功能。若要确定您的器件是否具有此功能，请参阅器件特定数据表。启用之后，可以使用 IPE 模块来防止 FRAM 存储器中的关键代码片段、配置数据或密钥被轻易访问或查看。启用之后，在 JTAG 调试期间，引导加载程序 (BSL)、DMA 访问或对 IPE 区域的读取访问都会返回 0x3FFF，同时还会保护其实际底层存储器内容。若要访问 IPE 区域内的任何内容，程序代码需要分支或调用存储在该段中的函数。只有 IPE 区域内的程序代码能够访问存储在该段中的任何数据。每当从非 IPE 区域直接访问 IPE 区域中的数据时，都会引发冲突。若要了解有关 IP 封装的更多信息，请参阅 *《MSP 代码保护功能》*。

7 参考文献

- *《MSP430FR58xx、MSP430FR59xx 和 MSP430FR6xx 系列用户指南》*
- *《MSP430FR57xx 系列用户指南》*
- *《MSP430FR4xx 和 MSP430FR2xx 系列用户指南》*
- *《MSP430 优化 C/C++ 语言编译器用户指南》*
- *《适用于 MSP430 的 IAR Embedded Workbench C/C++ 编译器用户指南》*
- *《MSP430™ FRAM 质量和可靠性》*

8 修订历史记录

注：以前版本的页码可能与当前版本的页码不同

Changes from Revision A (June 2020) to Revision B (August 2021)	Page
• 更新了整个文档中的表格、图和交叉参考的编号格式。	2

Changes from JUNE 23, 2014 to JUNE 30, 2020	Page
• 添加了表 3-1，*MSP430 C/C++ 数据类型*	2
• 更改了整个文档，包括更新 IDE 当前版本的图	3

重要声明和免责声明

TI 提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他安全、安保或其他要求。这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的应用。严禁对这些资源进行其他复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款 (https:www.ti.com/legal/termsofsale.html) 或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265

重要声明和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他功能安全、信息安全、监管或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的应用。严禁对这些资源进行其他复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265