应用报告
ZHCA930–June 2014

MSP430™FRAM 技术 – 操作方法和最佳实践

William Goh and Andreas Dannenberg

摘要

FRAM 采用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方面概述了 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，从而最大程度提高应用的稳健性。

内容

附图目录

附表目录

商标

SLAA628 — http://www-s.ti.com/sc/techlit/SLAA628

1 FRAM 和通用存储器

FRAM 是一种非易失性存储器技术，具备独一无二的灵活性，可用于程序或数据存储器。可以逐位向 FRAM 写入数据，它拥有几乎无限的写入周期（10^{15} 个周期 - 请参阅特定器件数据表）。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*》(SLAA526)。

2 可以将其视为 RAM

与 SRAM 类似，FRAM 也具有几乎无限的写入寿命，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 访问速度被限制为 8MHz，而 SRAM 访问速度可以达到器件的最高工作频率。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，请参阅特定器件数据表。

3 存储器布局分区

由于 FRAM 存储器可用作程序代码、变量、常量、栈等的通用存储器，因此必须针对应用对该存储器进行分区。 用于 MSP430 IDE 的 Code Composer Studio™和 IAR Embedded Workbench®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 的连接器命令文件内部。默认情况下，连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 3.4 节。

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

变量由默认的连接器命令文件分配到 SRAM 中。基于 FRAM 的 MSP430 器件通常具有 2KB 的 SRAM。有关确切的规格，请参阅特定器件数据表。如果变量由于过大而不适用于 SRAM，则可以修改连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。3.4 节显示了修改连接器命令文件以将变量从 SRAM 移到 FRAM 的方法。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4 节中所述。

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。在大多数 应用 中，栈是最频繁访问的存储器区域，这样有助于确保实现最高的应用性能。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，在大多数乃至所有用例中，循环通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 的工具链都附带连接器命令文件，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5 节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连接器命令文件进行修改。

由于 FRAM 同样能够存储代码、常量数据和变量数据，因此通常可以让连接器来完成对存储器进行分区这一任务。例如，如果按照5 节中的说明利用编译器扩展将应用数据分配到 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，以使连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，您可能希望将连接器的某些部分强制放到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，并希望将连接器的相应部分分配到 FRAM 中，以确保有大量的可用存储空间。

要定制存储器分区，通常需要修改特定项目的连接器命令文件，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，但强烈建议查阅连接器文档，以熟练掌握连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 项目都有一个连接器命令文件 (.cmd)，它会在创建项目时填充到项目文件夹中。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 应用最常用的存储器段。

```
.const              /* Constant data                                          */
.text               /* Application code                                       */
.bss                /* Uninitialized Global and static variables - default in RAM  */
.data               /* Initialized Global and static variables - default in RAM    */
.stack              /* Software system stack - default in RAM                 */
```

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）的连接器段时，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，此定义已经存在于连接器命令文件中，并将声明为 #pragma PERSISTENT 的变量放到 FRAM 中。有关 CSSv6.0.0 的重要信息，请参阅附录 A。对于 .TI.noinit，如果必须使用 #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         /* For #pragma NOINIT              */
        .TI.persistent : {} > FRAM         /* For #pragma PERSISTENT          */
```

3.4.2 用于 MSP430 的 IAR Embedded Workbench

在 IAR 中，如果按5.2 节中所述使用 __persistent 属性将变量置于 FRAM 中，则很多用例将无需修改连接器命令文件 (.xcl)。但是，如果要将声明为 __noinit 的变量放到 FRAM 中，则可以在连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 的连接器命令文件通常在所有项目之间共享，并位于 C:\<IAR installation directory>\430\config\linker\ 文件夹中。要详细了解连接器命令文件中的每个存储器段名称，请访问 http://www.iar.com/Products/IAR-Embedded-Workbench/TI-MSP430/User-guides/，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连接器命令文件，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 连接器命令文件。

1. 导航到 C:\<IAR 安装目录>\430\config\linker\ 文件夹。
2. 为您本地项目的 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. 配置 IAR 项目，以指向定制的连接器命令文件。

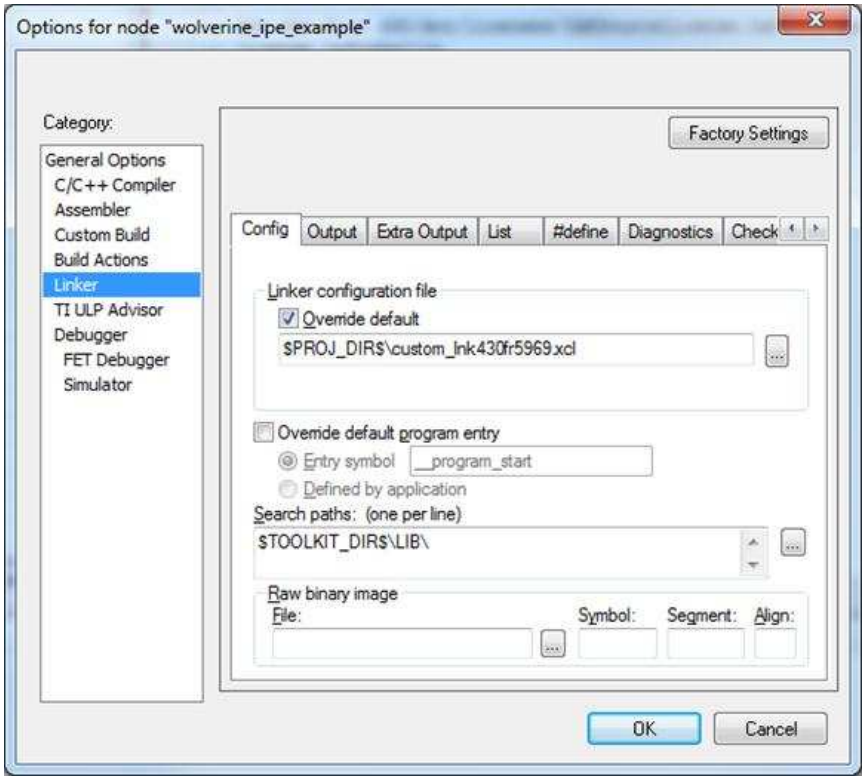

图 **1.** 为 **IAR** 手动覆盖连接器命令文件

4 优化应用能耗和性能

4.1 缩短从 LPMx.5 唤醒的时间

MSP430 上可实现最低功耗的模式是 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非常有限的深度休眠模式，并可能提供了一部分“备用 RAM”，此 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。这些存储器部分通常非常小（几十个字节），因此可以存储的上下文非常有限。另一方面，闪存可以是用来存储大量应用上下文的另一个选项，但这样会对 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，它能够自动在器件的 FRAM 中存储并保留不超过可用 FRAM 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，需要特别注意是如何声明和使用变量的。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、特定应用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 persistent 变量或 no-init 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 独有的通用惯例，为了尽可能缩短从 LPMx.5 深度休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产生直接的积极影响。

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4 节中所述，利用本文中讨论的 persistent 或 no-init 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。

5.1 TI Code Composer Studio

Code Composer Studio (CCS) 中具有两个可以使用的 C 语言 pragma 声明：`#pragma PERSISTENT` 和 `#pragma NOINIT`。在使用任一 pragma 之前，请参阅节 3.4.1了解连接器命令文件要求并参阅附录 A 了解有关 CSSv6.0.0 的重要信息，以确保将使用任一 pragma 指令声明的变量存储到 FRAM 存储器中。此外，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*》(SLAU132)。

`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（例如循环通电之后）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` 连接器存储器段中。有关 CSSv6.0.0 的重要信息，请参阅附录 A。

下面这个代码片段显示了变量如何被声明为 persistent：

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

NOINIT 的工作方式与 PERSISTENT 类似，但这些变量最初永远不会被项目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 连接器存储器段中。请注意，与 PERSISTENT 不同，声明为 NOINIT 的变量不会被默认的连接器命令文件分配到 FRAM 中，因此当需要此功能时，需要对连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 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，以便于使用 FRAM 来存储数据。要了解有关这两个属性的更多信息，请参阅 http://www.iar.com/Products/IAR-Embedded-Workbench/TI-MSP430/User-guides/ 上的《*IAR C/C++ 编译器用户指南*》。

对于 IAR 中的 persistent 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 连接器存储器段中，默认的 IAR 连接器命令文件 (.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 no-init 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 连接器存储器段中。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。如果需要使用此功能，需要对连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。MSP430 FRAM 器件具有一个内置的 MPU，它可以监视和监督软件中指定的存储器段，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。

注： 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 MPU，以确保实现最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 MPU。

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，需要了解链接程序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项目映射文件中，并填充到特定 IDE 的输出文件夹中。随后的几节将举例说明如何使用 MPU 来保护变量、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最高的灵活性。

在继续了解本节的内容之前，建议首先阅读《*MSP430FR58xx、MSP430FR59xx、MSP430FR68xx 和 MSP430FR69xx 系列用户指南*》(SLAU367) 或《*MSP430FR57xx 系列用户指南*》(SLAU272) 的“*存储器保护单元*”一章。

6.2 检查连接件映射文件

第一步是分析连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、变量、no-init、persistent 和程序代码。

6.2.1 对于 CCS

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 1 显示了所有这些段的起始地址。

表 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 对于 IAR

IAR 默认不生成映射文件。需要如图 2 中所示，选中“Project Options”下的*“Generate linker listing”*方框，以启用此功能。

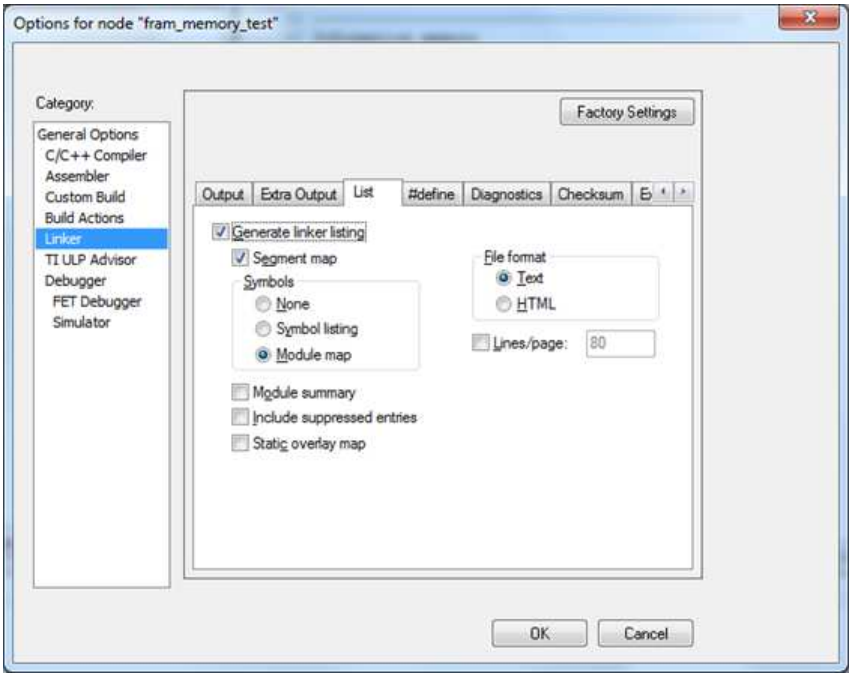

图 2. 用于启用映射文件的 IAR 保护选项

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

表 2 显示了 IAR 使用的几个通用段名称。

表 2. IAR 映射文件中的存储器分段

段名称	存储器区域	建议的保护类型（如果是在 FRAM 中）
DATAxx_Z	初始化为 0 的数据	读取和写入
DATAxx_I	初始化的数据	读取和写入
DATAxx_N	使用 `__no_init` 定义的数据	读取和写入
DATAxx_P	使用 `__persistent` 定义的数据	读取和写入
DATAxx_HEAP	“malloc”和“free”使用的堆	读取和写入
DATAxx_C	常量	只读
CODE	程序代码	读取和执行

6.3 手动配置 MPU

可以配置 MPU，以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 都有一些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 3 总结了典型的存储器分段。

表 3. 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
No-init	读取和写入	段 1
Persistent	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6.2 节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。有关更多信息，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 persistent 变量，其余字节用于应用代码。因此，在分配此示例应用时，连接器应当为变量分配 1KB，为常量分配 1KB，如表 4 中所示。

表 4. 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 - 0x47FF
常量	只读	段 2	0x4800 - 0x4BFF
程序代码	读取和执行	段 3	0x4C00 - 0xYYYYY

按照表 4 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 寄存器 (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 3 展示了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。

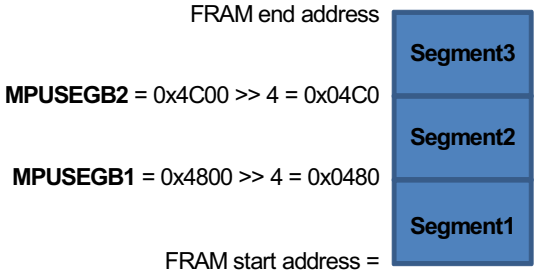

图 3. 要写入到 **MPUSEGBx** 寄存器中的地址

现在需要将配置实施到代码中。如6.1 节中所述，应在器件启动过程中尽早确定 MPU 配置。要进行此配置，请分别参阅节 6.3.1和节 6.3.2中概述的 CCS 和 IAR 配置步骤。

6.3.1 CCS MPU 实现方式

创建一个名为 *sytem_pre_init.c* 的新 C 文件，并将其添加到项目中。接下来，在此文件内放置一个 `int _system_pre_init(void)` 函数。如果此函数是项目的一部分，则工具链将确保在执行任何 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)
{
  /* Insert your low-level initializations here */

  /* Disable Watchdog timer to prevent reset during */
  /* long variable initialization sequences. */
  WDTCTL = WDTPW | WDTHOLD;

  // Configure MPU
  MPUCTL0 = MPUPW;                          // Write PWD to access MPU registers
  MPUSEGB1 = 0x0480;                        // B1 = 0x4800; B2 = 0x4C00
  MPUSEGB2 = 0x04c0;                        // Borders are assigned to segments
  // Segment 1 – Allows read and write only
  // Segment 2 – Allows read only
  // Segment 3 – Allows read and execute only
  MPUSAM = (MPUSEG1WE | MPUSEG1RE | MPUSEG2RE | MPUSEG3RE | MPUSEG3XE);
  MPUCTL0 = MPUPW | MPUENA | MPUSEGIE;      // Enable MPU protection
                                            // MPU registers locked until BOR

  /*=================================*/
  /* Choose if segment initialization */
  /* should be done or not.          */
  /* Return: 0 to omit initialization */
  /* 1 to run initialization         */
  /*=================================*/
  return 1;
}
```

正确配置之后，_system_pre_init() 函数应当在进入 main 之前执行完毕。进入 main() 时，观察 CCS 调试器的寄存器视图（图 4），其中显示了正在配置的 MPU 寄存器内容。

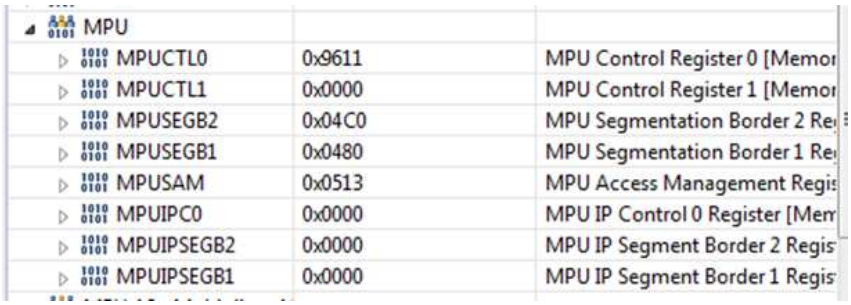

图 4. 寄存器窗口表明已启用 MPU

6.3.2 IAR MPU 实现方式

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项目中。IAR 中用于允许在器件启动之后立即执行应用代码的等效函数是 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)
{
  /* Insert your low-level initializations here */

  WDTCTL = WDTPW+WDTHOLD;

  // Configure MPU
  MPUCTL0 = MPUPW;                          // Write PWD to access MPU registers
  MPUSEGB1 = 0x0480;                        // B1 = 0x4800; B2 = 0x4C00
  MPUSEGB2 = 0x04c0;                        // Borders are assigned to segments
  // Segment 1 – Allows read and write only
  // Segment 2 – Allows read only
  // Segment 3 – Allows read and execute only
  MPUSAM = (MPUSEG1WE | MPUSEG1RE | MPUSEG2RE | MPUSEG3RE | MPUSEG3XE);
  MPUCTL0 = MPUPW | MPUENA | MPUSEGIE;      // Enable MPU protection
                                            // MPU registers locked until BOR
  /*
   * Return value:
   *
   *  1 - Perform data segment initialization.
   *  0 - Skip data segment initialization.
   */

  return 1;
}
```

6.4 基于 IDE 向导的 MPU 配置

6.4.1 CCS

图 5 显示了 Code Composer Studio v6 的内置 MSP430 MPU 向导，可通过“CCS Project Properties”访问此向导。要打开此对话框，请在 CCS 的“Project Explorer”视图中右键单击项目，然后选择 *Properties*。

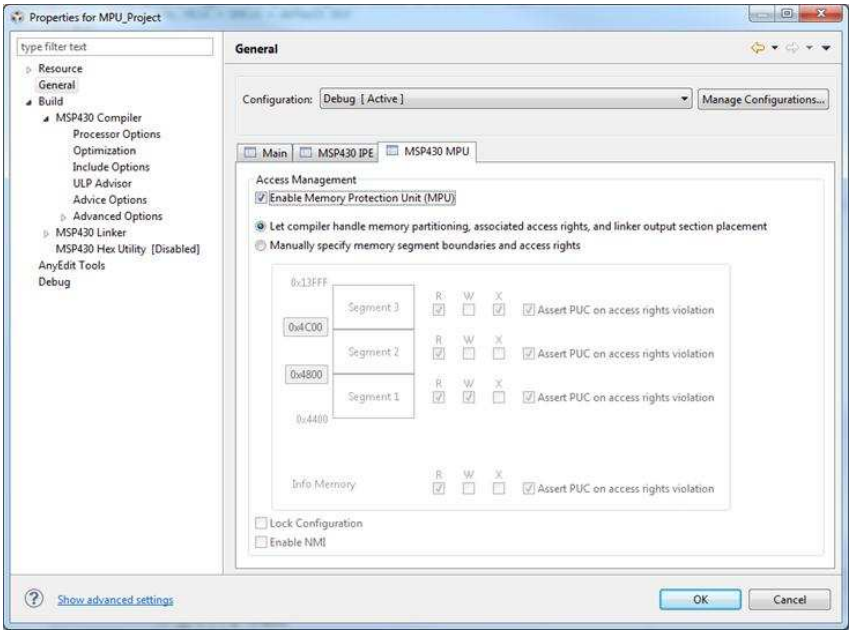

图 **5. CCS** 项目属性下的 **MPU** 向导

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。

6.4.2 IAR

IAR 中用于通过 MPU 向导配置 MPU 的 IDE 选项位于 *Project Options* → *General Options* → *MPU/IPE* 下，如图 6 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。

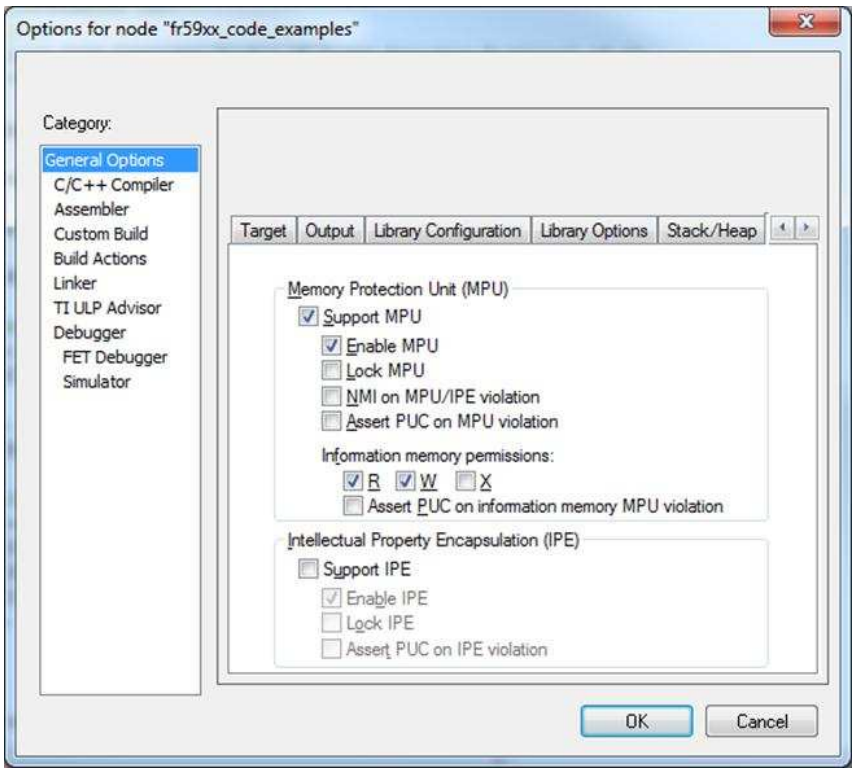

图 **6. IAR** 中的 **MPU** 和 **IPE** 向导

6.5 IP 封装

除 MSP430FR57xx 系列器件以外，采用 FRAM 技术的大多数 MSP430 器件都具有内置的 IP 封装 (IPE) 功能。要确定您的器件是否具有此功能，请参阅特定器件数据表。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。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。要启用 IPE，可以使用如下代码片段：

```
// This is the project dependent part
// Modify where you want the start and end address for IPE segment
#define IPE_START 0x04400 // This defines the Start of the IP protected area
#define IPE_END 0x04800 // This defines the End of the IP protected area

/*
 * Do not change this section - BEGIN
 */
// Hardcoded memory signature location
#define IPE_SIG_VALID   0xFF88   // IPE signature valid flag
#define IPE_STR_PTR_SRC 0xFF8A   // Source for pointer (nibble address) to MPU IPE structure

// Define the IPE signature location to 0xAAAA @ 0xFF88 to enable IPE
#pragma RETAIN(ipe_signalValid)
#pragma location=IPE_SIG_VALID
const unsigned int ipe_signalValid = 0xAAAA;
// Locate your IPE structure and it should be placed
// on the top of the IPE memory. In this example, IPE structure
// is at 0x4400
#pragma RETAIN(IPE_stringPointerSourceSource)
#pragma location=IPE_STR_PTR_SRC
const unsigned int IPE_stringPointerSourceSource = ((IPE_START)) >> 4;

// IPE data structures definition, reusable for ALL projects
#define IPE_SEGREG(a) (a >> 4)
#define IPE_BIP(a,b,c) (a ^ b ^ c ^ 0xFFFF)
#define IPE_FILLSTRUCT(a,b,c)
{a,IPE_SEGREG(b),IPE_SEGREG(c),IPE_BIP(a,IPE_SEGREG(b),IPE_SEGREG(c))}
typedef struct IPE_Init_Structure {
    unsigned int MPUIPC0 ;
    unsigned int MPUIPB2 ;
    unsigned int MPUIPB1 ;
    unsigned int MPUCHECK ;
} IPE_Init_Structure; // this struct should be placed inside IPB1/IPB2 boundaries

// Ensures the compiler do not optimize the structure since it is not called by the application
#pragma RETAIN(ipe_configStructure)
// IPE is defined in an adopted linker control file
#pragma DATA_SECTION(ipe_configStructure,".ipestruct");
// IPE is the section for protected data;
const IPE_Init_Structure ipe_configStructure = IPE_FILLSTRUCT(MPUIPLOCK + MPUIPENA,
IPE_END,IPE_START);
/*
 * Do not change this section - END
 */

// An example on how to place code inside the IPE region
#pragma SET_CODE_SECTION(".ipe")
extern void blinkLedFast(void);
extern void blinkLedSlow(void);
#pragma SET_CODE_SECTION()
```

使用上述代码片段中所示的 IPE 结构，器件的启动代码可以在执行任何代码之前初始化 IPE。这样可以提供最佳的 IP 保护，因为未经授权的代码将无法从此区域探查数据。当实施 IPE 结构时，一个良好的经验法则是将它分配到 IPE 区域内部。这样可以防止恶意代码修改 IPE 结构或者了解 IPE 的构建方式。可以在特定器件系列用户手册的*存储器保护单元 (MPU)* 一章中找到有关 IPE 模块工作方式的更多详细信息。

7 参考文献

- 《*MSP430FR58xx*、*MSP430FR59xx*、*MSP430FR68xx* 和 *MSP430FR69xx* 系列用户手册》(SLAU367)
- 《*MSP430xFR57xx* 系列用户指南》（文献编号：SLAU272）
- 《*MSP430* 优化 *C/C++* 编译器用户指南》(SLAU132)
- 《用于 MSP430 的 IAR Embedded Workbench C/C++ 编译器用户指南 (IAR)》：http://www.iar.com/Products/IAR-Embedded-Workbench/TI-MSP430/User-guides/
- 《*MSP430™ FRAM* 质量和可靠性》(SLAA526)

TI CCSv6.0.0 中的“FRAM 器件支持”

在本文档发布时，TI 的 Code Composer Studio v6.0.0 IDE 附带的 MSP430 器件支持包包含一个错误的连接器存储器布局和配置设置，因此可能会在开发期间导致一些问题并影响到所有的 MSP430FR5xx 和 MSP430FR6xx 器件。此外，本版本暂不支持自动将按照5.1 节中的说明通过 persistent 机制声明的变量放到 FRAM 中。随后的几节将概述如何识别受影响的 CCS 安装、可能的故障场景以及建议的解决办法。

A.1 在 CCS 中验证 MSP430 器件支持

MSP430 器件支持包是 CCS 内的一个无法追溯日期的组件；需要在验证 CCS IDE 自身版本的同时验证这个特定支持包的版本，才能确定 CCS 安装是否受到了影响。为此，在 Code Composer Studio IDE 内，选择 *Help* → *About Code Composer Studio* 菜单项。

图 7 显示了目前安装的 CCS IDE 的版本，在本例中是 6.0.0 内部版本 190。如果您的版本对话框显示的正是此版本，请继续执行下一步。如果您的版本高于此版本，则附录 A 中讨论的情况将不适用于您的 CCS 安装。

图 7. 目前安装的 Code Composer Studio

选择 *Installation Details* 并导航到 *MSP430 Emulators* 行条目之后，“Version”列中的数字会表明目前已安装 MSP430 器件支持包的版本。在这里显示的示例屏幕截图中，此版本是 6.0.0.12。如果您的对话框中显示的版本早于（但不包括）6.0.0.14，如图 8 中所示，则您将受到此附录中所概述的问题的影响。如果显示的版本是 6.0.0.14 或更高版本，表明您已更新 CCS 安装，它将不会出现此附录中讨论的问题。

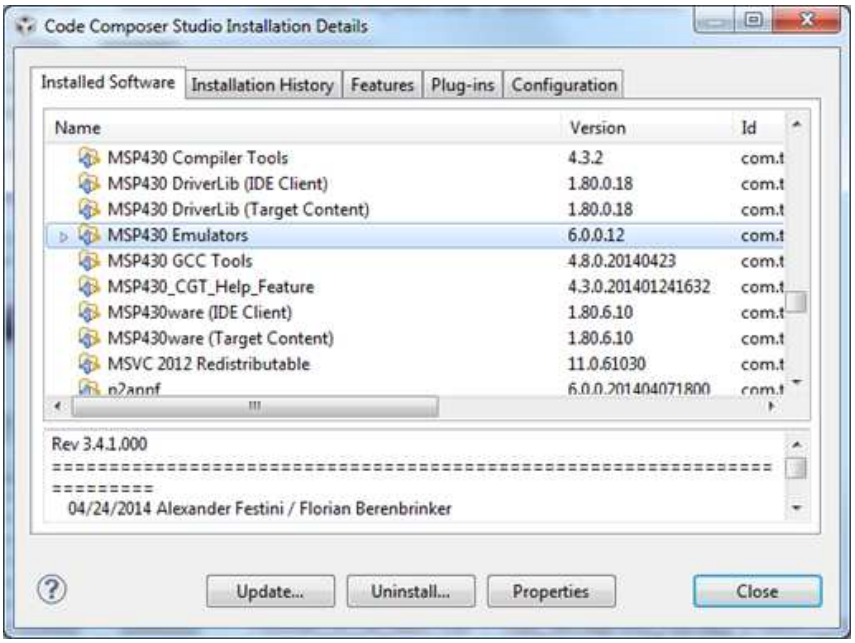

图 **8.** 目前安装的 **MSP430** 器件支持包的版本

A.2 故障场景 1

未显示任何问题或警告的项目构建。将项目下载到目标器件时，调试器报告此器件“正在运行”；而预期的行为是，当下载完成后并等待用户输入时，器件应在 `main()` 程序切入点处暂停。也可能是激活了中断函数之后器件发生了崩溃。此故障模式主要出现在具有 96KB 和 128KB FRAM 的器件型号上，故障是由于缺少对连接器命令文件的限制而导致的，必须为连接器命令文件设置这些限制，才能确保按照 MSP430 器件架构的要求将 C 低级初始化代码、中断矢量例程以及其他的一些关键存储器段分配到地址空间的低位 64KB 中。

A.3 故障场景 2

项目构建失败，表明出现了类似于如下所示的错误，尽管器件存储器不足以容纳整个项目文件。此故障模式主要出现在具有 64KB FRAM 的器件型号上，故障原因是当地址空间的低位 64KB 被填满时，连接器无法使用所分配的高于 0x10000 的存储器。

```
error #10099-D: program will not fit into available memory.
placement with alignment fails for section "ALL_FRAM"
```

A.4 故障场景 3

在器件循环通电期间，使用 `#pragma PERSISTENT` 声明的变量没有按照5.1 节中的说明保留它们的值。此问题出现在所有 FRAM 器件型号上，问题是由于 `.TI.persistent` 存储器段缺少连接器命令文件定义而导致的。

A.5 故障场景 4

项目利用 CCS IDE 的 MPU 或 IPE 配置机制（请参阅节 6.4.1），而且构建时没有出现任何问题或警告。但当运行代码时，应用会立即崩溃，或者虽然在运行，但器件在启动时修改了存储器内容。

此故障模式会出现在所有 MSP430FRxxx 器件上，故障是由于低位启动例程中进行了错误的栈指针初始化而导致的。

A.6 解决办法/权变措施

德州仪器 (TI) 目前正在更新 MSP430 器件支持包以解决上述问题，此支持包将通过 CCS IDE 的内置更新机制部署到所有 CCSv6.0.0 客户的器件中。更新后的支持包的版本号将是 6.0.0.14，暂定在 2014 年 6 月末/7 月初的这段时间内进行部署。有关如何识别已安装支持包的信息，请参阅A.1 节。

尽管可以手动修改受影响的 CCS 安装中的连接器命令文件以解决上述问题，但很少需要执行此操作，原因在于需要进行一些重大变更并考虑到由 CCS IDE 管理的 MPU 和 IPE 配置过程所涉及的相关性。因此，强烈建议等待部署 CCS 更新，以便彻底解决上述问题。与此同时，建议将受影响器件上代码和数据的总大小保持在 47KB 以下。此外，还建议禁止启用 CCS 的 MPU 和 IPE 配置对话框，并允许使用手动方法设置这些模块。

如果这种解决办法目前不奏效，请联系 TI 支持部门 (http://www.ti.com.cn)。此外，还可以考虑使用其他的工具链，例如 IAR Embedded Workbench。

请注意，一旦德州仪器 (TI) 更新了 MSP430 器件支持包，您应迁移您的项目，以使用最新的连接器命令文件。为此，请执行以下操作：

1. 创建一个与您的 MSP430FR5xx/6xx 器件型号相对应的空 CCS 项目。
2. 将自动填充到新创建项目中的连接器命令文件移到您自己的项目中。

重要声明和免责声明

TI 均以“原样”提供技术性及可靠性数据（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，不保证其中不含任何瑕疵，且不做任何明示或暗示的担保，包括但不限于对适销性、适合某特定用途或不侵犯任何第三方知识产权的暗示担保。

所述资源可供专业开发人员应用TI 产品进行设计使用。您将对以下行为独自承担全部责任：(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。所述资源如有变更，恕不另行通知。TI 对您使用所述资源的授权仅限于开发资源所涉及TI 产品的相关应用。除此之外不得复制或展示所述资源，也不提供其它TI或任何第三方的知识产权授权许可。如因使用所述资源而产生任何索赔、赔偿、成本、损失及债务等，TI对此概不负责，并且您须赔偿由此对TI 及其代表造成的损害。

TI 所提供产品均受TI 的销售条款 (http://www.ti.com.cn/zh-cn/legal/termsofsale.html) 以及ti.com.cn上或随附TI产品提供的其他可适用条款的约束。TI提供所述资源并不扩展或以其他方式更改TI 针对TI 产品所发布的可适用的担保范围或担保免责声明。

邮寄地址：上海市浦东新区世纪大道 1568 号中建大厦 32 楼，邮政编码：200122

重要声明和免责声明

TI 均以“原样”提供技术性及可靠性数据（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，不保证其中不含任何瑕疵，且不做任何明示或暗示的担保，包括但不限于对适销性、适合某特定用途或不侵犯任何第三方知识产权的暗示担保。

所述资源可供专业开发人员应用TI 产品进行设计使用。您将对以下行为独自承担全部责任：(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。所述资源如有变更，恕不另行通知。TI 对您使用所述资源的授权仅限于开发资源所涉及TI 产品的相关应用。除此之外不得复制或展示所述资源，也不提供其它TI或任何第三方的知识产权授权许可。如因使用所述资源而产生任何索赔、赔偿、成本、损失及债务等，TI对此概不负责，并且您须赔偿由此对TI 及其代表造成的损害。

TI 所提供产品均受TI 的销售条款 (http://www.ti.com.cn/zh-cn/legal/termsofsale.html) 以及ti.com.cn上或随附TI产品提供的其他可适用条款的约束。TI提供所述资源并不扩展或以其他方式更改TI 针对TI 产品所发布的可适用的担保范围或担保免责声明。

邮寄地址：上海市浦东新区世纪大道 1568 号中建大厦 32 楼，邮政编码：200122