

MSP430™ FRAM Technology – How To and Best Practices

William Goh and Andreas Dannenberg

ABSTRACT

FRAM is a non-volatile memory technology that behaves similar to SRAM while enabling a whole host of new applications, but also changing the way firmware should be designed. This application report outlines the how to and best practices of using FRAM technology in MSP430 from an embedded software development perspective. It discusses how to implement a memory layout according to application-specific code, constant, data space requirements, the use of FRAM to optimize application energy consumption, and the use of the Memory Protection Unit (MPU) to maximize application robustness by protecting the program code against unintended write accesses.

Contents

1	FRAM and Universal Memory	2
2	Treat it Just Like RAM	2
3	Memory Layout Partitioning	2
4	Optimizing Application Energy Consumption and Performance	5
5	Ease-of Use Compiler Extensions for FRAM	5
6	FRAM Protection and Security	6
7	References	14
Appendix A	FRAM Device Support in TI CCSv6.0.0	15

List of Figures

1	Manual Override Linker Command File for IAR	4
2	IAR Project Option to Enable Map File	7
3	Address to be Written in MPUSEGBx Registers	9
4	Registers Window Showing MPU Enabled	10
5	MPU Wizard Under CCS Project Properties.....	11
6	MPU and IPE Wizard in IAR.....	12
7	Code Composer Studio Currently Installed	15
8	MSP430 Device Support Package Version Currently Installed	16

List of Tables

1	Memory Segmentation Inside CCS Map File	7
2	Memory Segmentation in IAR Map File.....	8
3	MPU Memory Segmentation	8
4	MPU Memory Segmentation Example	8

1 FRAM and Universal Memory

FRAM is a non-volatile memory technology that is uniquely flexible and can be used for program or data memory. It can be written to in a bit-wise fashion and with virtually unlimited write cycles (10^{15} cycles – see device-specific data sheet). To learn more about FRAM, visit www.ti.com/fram and refer to *MSP430™ FRAM Quality and Reliability* ([SLAA526](#)).

2 Treat it Just Like RAM

Similar to SRAM, FRAM has virtually unlimited write endurance with no signs of degradation. FRAM does not require a pre-erase in which every write to FRAM is non-volatile. However, there are some minor trade-offs in using FRAM instead of RAM that may apply to a subset of use cases. One of the differences on the MSP430 platform is the FRAM access speed, which is limited to 8 MHz, whereas, SRAM can be accessed at the maximum device operating frequency. Wait-states are required if the CPU accesses the FRAM at speeds faster than 8 MHz. Another trade-off is that FRAM access results in a somewhat higher power consumption compared with SRAM. For more details, see the device-specific data sheet.

3 Memory Layout Partitioning

Since FRAM memory can be used as universal memory for program code, variables, constants, stacks, and so forth, the memory has to be partitioned for the application. Code Composer Studio™ and IAR Embedded Workbench® for MSP430 IDEs can both be used to set up an application's memory layout to make best-possible use of the underlying FRAM depending on the application needs. These memory partitioning schemes are generally located inside the IDE-specific linker command file. By default, the linker command files will typically allocate variables and stacks into SRAM. And, program code and constants are allocated in FRAM. These memory partitions can be moved or sized depending on your application needs. For more details, see [Section 3.4](#).

3.1 Program Code and Constant Data

Both program code and constant data should be allocated in FRAM just like it would be done in a Flash memory-based context. Furthermore, to ensure maximum robustness and data integrity, the MPU feature should be enabled for those regions such that they are protected against write accesses. This helps prevent accidental modification that could result from possible errant write accesses to those memory regions in case of program failures (software crash), buffer overflows, pointer corruption, and other types of anomalies.

3.2 Variables

Variables are allocated in SRAM by the default linker command files. FRAM-based MSP430 devices would typically have 2KB of SRAM. For the exact specification, see the device-specific data sheet. If the variable size is too large to fit in SRAM, the linker command file can be modified or C-language `#pragma` directives can be used to allocate specific variables or structures in FRAM memory. [Section 3.4](#) showcases how you would modify the linker command file to move variables from SRAM to FRAM. Aside from SRAM memory constraint, another reason you would use FRAM for variables is to decrease start-up time as outlined in [Section 4](#).

3.3 Software Stack

Although FRAM can be used for the stack in a typical application, it is recommended to allocate the stack in the on-chip SRAM. The CPU can always access the SRAM at full-speed with no wait-states independent of the chosen CPU clock frequency (MCLK). Since in most applications the stack is the most frequently accessed memory region, this helps ensure maximum application performance. Likewise, since SRAM memory accesses are even lower power than FRAM write accesses, allocating the stack in SRAM also yields to lower active power consumption numbers. Last but not least, the contents of the stack does not need to be preserved through a power cycle, in most if not all use cases, since the application code performs a cold start and re-initializes the basic C runtime context anyways.

MSP430, Code Composer Studio are trademarks of Texas Instruments.
IAR Embedded Workbench is a registered trademark of IAR Systems AB.
All other trademarks are the property of their respective owners.

3.4 Memory Partitioning Support in the MSP430 IDEs

The toolchains available for MSP430 all ship with linker command files that define a default memory setup and partitioning, typically allocating program code and constant data into FRAM, and variable data and the system stack to SRAM. In addition, C compiler language extensions are provided that allow you to locate selected variables and data structures into FRAM as described in [Section 5](#), allowing you to utilize the benefits of using FRAM for persistent data storage without any further considerations regarding memory partitioning or modifications of the linker command files.

Due to the nature of the FRAM being equally usable for both code, constant and variable data storage, the task of partitioning the memory can be typically left to the linker. For example, if you allocate application data into FRAM through the use of compiler extensions as described in [Section 5](#), the space available for program code automatically reduces by the amount that is consumed by such variables, as the linker places all its output segments into the same “pool” of FRAM.

There are, however, application use cases that may require a higher level of customization. For example, you may desire to limit certain linker sections to specific fixed memory regions to enable an easier manual setup of the MPU module. Or another application use case may want to locate certain variables into a memory region specifically reserved for the purpose of storing data in a non-volatile fashion such that they can then be made available later even after an in-system firmware update. And, another application may have large data or stack size requirements exceeding the size of the on-chip SRAM, and may want to allocate the corresponding linker sections into FRAM to ensure that a large amount of storage is available.

Customization of the memory partitioning typically involves making modifications to a project-specific linker command file that is based off the default file that ships with the IDE. While some changes may seem intuitive and obvious, it is highly recommended to obtain a good working knowledge of the linker and its command files by consult the linker documentation. This section provides a starting point into such customization efforts.

3.4.1 TI Code Composer Studio

Every CCS project has a linker command file (.cmd) that gets populated into the project folder upon project creation. This file describes the allocation of program code, variables, constants, and stacks for the device. It also describes the priority of how each memory segments are ordered in the device. The following segment names listed below are the most commonly used items for most applications.

```
.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
```

In addition, the compiler has the capability to automatically place selected variables, arrays, or structures into FRAM when linker segments named `.TI.noinit` (for use in conjunction with `#pragma NOINIT`) and `.TI.persistent` (for use with `#pragma PERSISTENT`) are assigned to the FRAM. In case of `.TI.persistent`, this definition is already present in the linker command file, locating variables declared as `#pragma PERSISTENT` into FRAM. For important information regarding CSSv6.0.0, see [Appendix A](#). In case of `.TI.noinit`, such an assignment can be made by the customer in analogy to the existing `.TI.persistent` if the `#pragma NOINIT` feature should be used to locate variables and structures not requiring C startup initialization into FRAM.

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

3.4.2 IAR Embedded Workbench for MSP430

In IAR, modifying the linker command (.xcl) file is not needed in many use cases if variables are located in FRAM through the use of the `__persistent` attribute, as outlined in [Section 5.2](#). However, if it is desired to locate variables declared as `__noinit` into FRAM, then a minor modification can be made to accommodate this by moving the `DATA16_N` and `DATA20_N` segment assignments in the linker command file from RAM into the FRAM region.

IAR's linker command files are generally shared across all projects and are located in the C:\<IAR installation directory>\430\config\linker\ folder. To get a detailed understanding of each memory segment name in the linker command file, see the *Segment Reference* chapter in the *IAR C/C++ Compiler User's Guide* located at <http://www.iar.com/Products/IAR-Embedded-Workbench/TI-MSP430/User-guides/>.

If a customized linker command file is still required, a copy of `lnk430xxxx.xcl` needs to be made. The following steps outline how to create a custom IAR linker command file.

1. Navigate to the C:\<IAR installation directory>\430\config\linker\ folder.
2. Make a copy of the `lnk430xxxx.xcl` file to your local project and rename the filename, if needed.
3. Open the new copy of the `.xcl` file and customize it
4. Configure the IAR project to point to the customized linker command file.

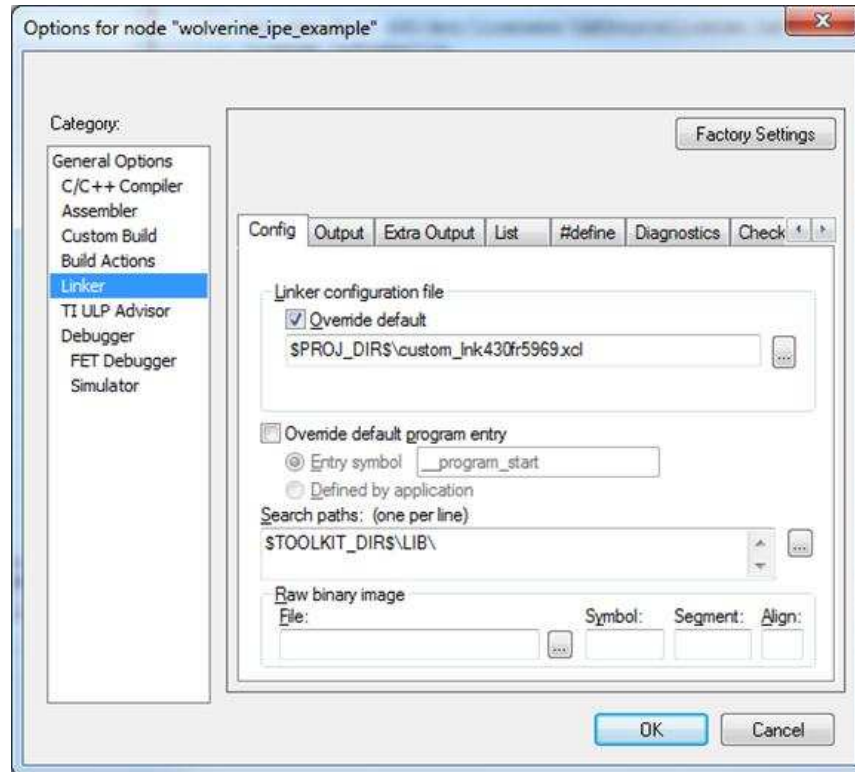


Figure 1. Manual Override Linker Command File for IAR

4 Optimizing Application Energy Consumption and Performance

4.1 Decrease Wake-Up Time From LPMx.5

The lowest-possible power modes on MSP430 are the LPM3.5 and LPM4.5 modes since the majority of the device is powered down and only limited functionality is available.

Low-Power Mode (LPMx)	Available Device Functionality	Wake-Up Sources
LPM3.5	RTC, 32-kHz Oscillator	RTC and GPIO Interrupts
LPM4.5	None	GPIO Interrupts

However, waking up from those modes is similar to coming out of RESET, posing the need to store the application context prior into entering LPMx.5 into non-volatile memory and restoring it after the device wakes up from LPMx.5. Other microcontrollers have similar limited-functionality deep-sleep modes and may offer a section of “backup RAM” that will stay powered during those modes to aid the storing of application context. Those memory sections typically are very small (10’s of bytes) so the context that can be stored is limited. On the other hand, Flash memory could be another option to store a larger amount of application context; however, doing so would have a significant impact on the applications power and real-time performance, aside from other limitations that Flash memory typically brings such as limited erase and write endurance. In contrast with FRAM, it is possible automatically store and maintain the entire application context such as data buffers, status variables, and various flags in FRAM, up to the size of the available FRAM, on a device without any need to store or restore data and without any impact on the applications power consumption or real-time behavior.

To take advantage of FRAM in LPMx.5-using applications, special attention has to be paid as to how variables are declared and used. Specifically, variables should be declared as either persistent or no-init for everything that holds application context (state variables and flags, result registers, application-specific calibration settings and baseline values, intermediate calculation or signal processing results, and so on) to prevent having to re-calculate or re-obtain those after power-on.

As a general practice, not specific to FRAM, in order to optimize application wakeup time after resuming from an LPMx.5 deep-sleep type of mode, all variables where the initial value does not matter (think of large arrays used as buffers, they may not need to be zero-initialized) should also be declared as no-init, which helps saving processor cycles within the C auto-initialization startup routine during application boot and has a direct positive impact on the application startup time and energy consumption.

5 Ease-of Use Compiler Extensions for FRAM

This section outlines how to leverage built-in compiler extensions to locate specific variables in FRAM so that their values can be preserved during power cycles or periods of any length where the system is completely powered down. Locating variables in FRAM through either persistent or no-init mechanisms discussed here also helps to reduce the application wake-up time and with this its energy consumption, as discussed in [Section 4](#), as those variables will not get initialized by the C startup routine.

5.1 TI Code Composer Studio

In CCS, there are two C language pragma statements that can be used: `#pragma PERSISTENT` and `#pragma NOINIT`. Before using either of these pragmas, see [Section 3.4.1](#) on linker command file requirements as well as [Appendix A](#) for important information regarding CSSv6.0.0 to ensure that variables declared using either pragma directive are stored in FRAM memory. Also, for more detailed information on these pragma directives, see the *MSP430 Optimizing C/C++ Compiler User’s Guide* ([SLAU132](#)).

`PERSISTENT` causes variables to not get initialized by the C startup routine, but rather the debug tool chain initializes them for the first time as the application code is loaded onto the target device. Subsequently, those variables do not get initialized, for example, after a power cycle as they have been completely excluded from the C startup initialization process. Declaring variables as `PERSISTENT` causes them to get allocated into the `.TI.persistent` linker memory segment. For important information regarding CSSv6.0.0, see [Appendix A](#).

Here is a code snippet showing how the variable is declared as persistent:

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

NOINIT works similar to PERSISTENT but the variables are never initially initialized by the project's binary image file and the debug tool chain during code download. Declaring variables as NOINIT causes them to get allocated into the .TI.noinit linker memory segment. Note that unlike PERSISTENT, variables declared as NOINIT do not get located in FRAM by the default linker command files, requiring a minor modification of the linker command file if such functionality is required. Here is a corresponding code snippet:

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

5.2 IAR Embedded Workbench for MSP430

In IAR, two C language extension attributes named `__persistent` and `__no_init` are provided that facilitate the use of FRAM for data storage. For additional information regarding these attributes, see the *IAR C/C++ Compiler User's Guide* located at <http://www.iar.com/Products/IAR-Embedded-Workbench/TI-MSP430/User-guides/>.

For persistent storage functionality in IAR, variables can be declared using the `__persistent` attribute. Variables declared with this attribute are allocated into the DATA16_P and DATA20_P linker memory segments, which the default IAR linker command files (.xcl) automatically locate in FRAM. Below shows an example of a variable x declared such that it is not initialized during C startup and automatically allocated in FRAM memory. Furthermore, similar to the behavior in CCS, this variable only gets initialized by the debug tool chain during the initial code download but not at application startup or runtime.

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

Similarly, no-init storage functionality also exists in IAR through the use of the `__no_init` attribute. Declaring variables with this attribute causes them to be allocated into the DATA16_N and DATA20_N linker memory segments. And also unlike in the case of `__persistent`, variables declared as `__no_init` will not get allocated into FRAM by default. If such functionality is required, a minor modification to the linker command file is needed.

```
__no_init unsigned int x;
```

6 FRAM Protection and Security

6.1 Memory Protection

FRAM is easy to write to. Application code, constants, and some variables residing in FRAM need to be protected against unintended writes that may result from invalid pointer accesses, buffer overflows, and other anomalies that could potentially corrupt your application. MSP430 FRAM devices have a built-in MPU that monitors and supervises memory segments as defined in software to be protected as read, write, execute or a combination of them.

NOTE: It is very important to always appropriately configure and enable the MPU before any software deployment or production code release to ensure maximum application robustness and data integrity. The MPU should be enabled as early as possible after the device starts executing code coming from a power-on or reset at the beginning of the C startup routine even before the *main()* routine is entered.

Before protecting the memory, the FRAM memory needs to be partitioned. To partition, understanding the program size and types of memory segments after program linking is important to decide how each memory segments are protected. This information is generally located in the project map file that is generated during application build and gets populated to an IDE-specific output folder. The following sections describe an example of how variables, constants, and program code can be protected using the MPU. The configuration can be performed automatically by the MPU, or can be done manually for maximum flexibility.

It is recommended to first read the *Memory Protection Unit* chapter in the *MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family User's Guide* ([SLAU367](#)) or the *MSP430FR57xx Family User's Guide* ([SLAU272](#)) before proceeding with this section.

6.2 Inspecting the Linker MAP File

The first step is to analyze the linker-generated map file to determine the start and size of the memory segments that constitute the application firmware image; constants, variables, no-init, persistent, and program code.

6.2.1 For CCS

In the map file, look for `.bss`, `.data`, `.TI.noinit`, `.TI.persistent`, `.const`, and `.text` memory start addresses and their size. This information can be used to determine how the MPU should be manually configured. Note that [Table 1](#) shows the starting addresses for each of these segments.

Table 1. Memory Segmentation Inside CCS Map File

Segment Name	Memory Region	Recommended Protection Type (if in FRAM)
<code>.bss/.data</code>	Variables	Read and Write
<code>.TI.noinit</code>	Data defined using <code>#pragma NOINIT</code>	Read and Write
<code>.TI.persistent</code>	Data defined using <code>#pragma PERSISTENT</code>	Read and Write
<code>.sysmem</code>	Heap used by 'malloc' and 'free'	Read and Write
<code>.const</code>	Constants	Read only
<code>.text</code>	Program Code	Read and Execute

6.2.2 For IAR

IAR does not generate the map file by default. This feature needs to be enabled by checking *Generate linker listing* box under the Project Options, as shown in [Figure 2](#).

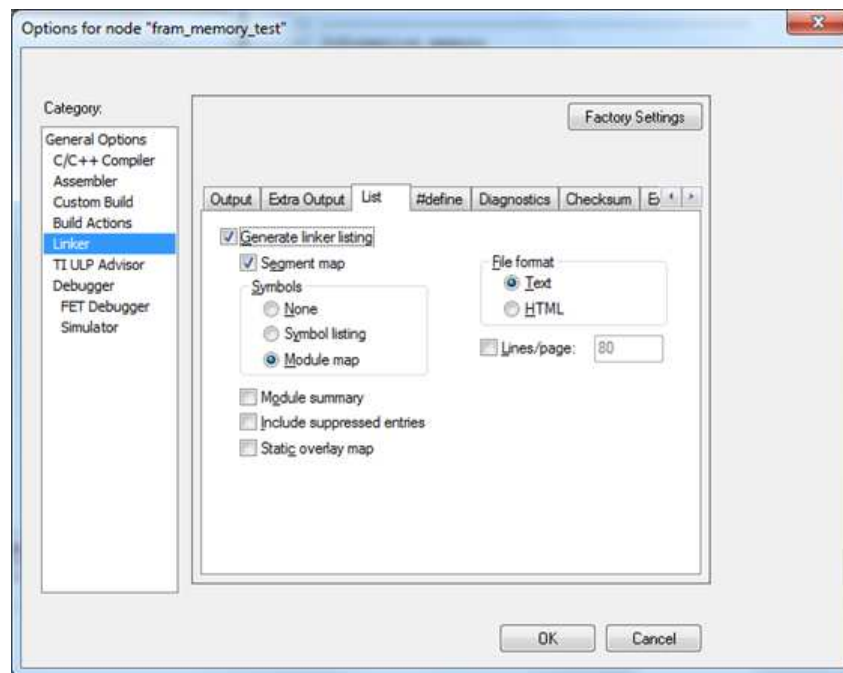


Figure 2. IAR Project Option to Enable Map File

Once enabled, the map file should be located in the project once it has been successfully compiled. Open up the map file and analyze it for the following segment names.

Table 2 shows several of the general segment names that are used by IAR.

Table 2. Memory Segmentation in IAR Map File

Segment Name	Memory Region	Recommended Protection Type (if in FRAM)
DATAxx_Z	Data initialized to zero	Read and Write
DATAxx_I	Initialized data	Read and Write
DATAxx_N	Data defined using <code>__no_init</code>	Read and Write
DATAxx_P	Data defined using <code>__persistent</code>	Read and Write
DATAxx_HEAP	Heap used by 'malloc' and 'free'	Read and Write
DATAxx_C	Constants	Read only
CODE	Program Code	Read and Execute

6.3 Manual MPU Configuration

The MPU can be configured to protect three different memory segments in software. Each segment can be individually configured to read, write, execute, or a combination of them. Most applications would have some form of variables that should be protected as read and write, constants to be read only, and program code should be read and execute only. Table 3 summarizes the typical memory segmentation.

Table 3. MPU Memory Segmentation

Memory Region	Protection Type	MPU Segment
Variables	Read and Write	Segment 1
No-init	Read and Write	Segment 1
Persistent	Read and Write	Segment 1
Constants	Read only	Segment 2
Program Code	Read and Execute	Segment 3

Once the starting address for the application's read and write, read only, and read and execute segment has been identified from the generated map file in Section 6.2, the next step is to determine and configure the segment boundaries for the MPU. Do keep in mind that the smallest MPU segment size allocation is 1KB or 0x0400. For additional information, see the device-specific family user's guide. In this example, the application uses only 5-bytes of constant array, 2-bytes used for persistent variable, and remainder is application code. Therefore, the linker should allocate this example application in which 1KB for variables and 1KB for constants, as shown in Table 4.

Table 4. MPU Memory Segmentation Example

Memory Region	Protection Type	MPU Segment	Example Memory Partition
Variables	Read and Write	Segment 1	0x4400 – 0x47FF
Constants	Read only	Segment 2	0x4800 – 0x4BFF
Program Code	Read and Execute	Segment 3	0x4C00 – 0xYYYYY

Once the memory segmentation has been decided for segment 1, 2, and 3, as shown in Table 4, there are two registers to define how the segment boundaries are configured: Memory Protection Unit Segmentation Border 1 (MPUSEGB1) and Memory Protection Unit Segmentation Border 2 Register (MPUSEGB2). Before writing to the register, the address needs to be shifted to the right by 4 bits.

Figure 3 illustrates the application example of how the memory has been partitioned and MPU registers are configured.

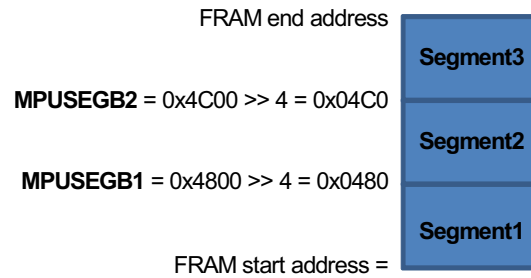


Figure 3. Address to be Written in MPUSEGBx Registers

Now it is time to implement the configuration in code. As mentioned in [Section 6.1](#), the MPU configuration should be made as early as possible in the device's boot process. To implement this, [Section 6.3.1](#) and [Section 6.3.2](#) outline the steps for CCS and IAR, respectively.

6.3.1 CCS MPU Implementation

Create a new C-file called *sytem_pre_init.c* and include it inside the project. Next, a function is placed inside this file, `int _system_pre_init(void)`. If such a function is part of the project, the toolchain makes sure it is executed first before any C startup initialization routines and well before the code execution is transferred to `main()`. This function is typically used for critical routines needing to be executed early as soon as the device starts up.

Here is a code snippet example to enable the MPU. Configure the MPU configuration as needed based on the application.

```
#include <msp430.h>

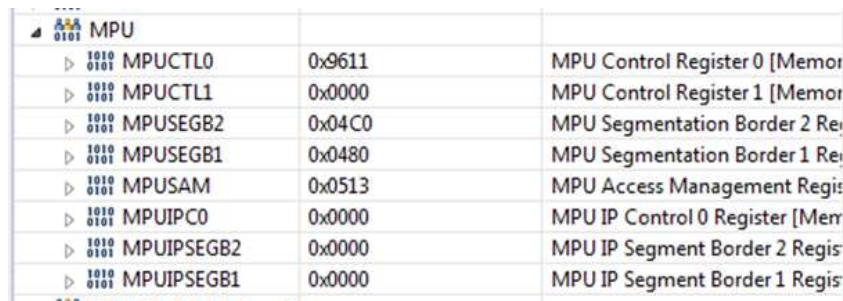
int _system_pre_init(void)
{
    /* Insert your low-level initializations here */

    /* 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;
}
```

When properly configured, the `_system_pre_init()` function should have been executed before entering `main()`. Upon entering `main()`, observe the CCS debugger's register view (Figure 4) showing the MPU register contents being configured.



Register	Value	Description
MPUCTL0	0x9611	MPU Control Register 0 [Memory Protection Unit]
MPUCTL1	0x0000	MPU Control Register 1 [Memory Protection Unit]
MPUSEGB2	0x04C0	MPU Segmentation Border 2 Register
MPUSEGB1	0x0480	MPU Segmentation Border 1 Register
MPUSAM	0x0513	MPU Access Management Register
MPUIPC0	0x0000	MPU IP Control 0 Register [Memory Protection Unit]
MPUIPSEGB2	0x0000	MPU IP Segment Border 2 Register
MPUIPSEGB1	0x0000	MPU IP Segment Border 1 Register

Figure 4. Registers Window Showing MPU Enabled

6.3.2 IAR MPU Implementation

To do the equivalent in IAR, a new C-file has to be created with the name `low_level_init.c`. This file would need to be included in your project. IAR's equivalent function to enable the execution of application code as soon as the device starts up is `int __low_level_init(void)`. The following code snippet example shows an equivalent MPU configuration for IAR.

```
#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 IDE Wizard-Based MPU Configuration

6.4.1 CCS

Figure 5 depicts Code Composer Studio v6's built-in MSP430 MPU Wizard accessible through the CCS Project Properties. To open this dialog, right-click on the project in the CCS' Project Explorer view and select *Properties*.

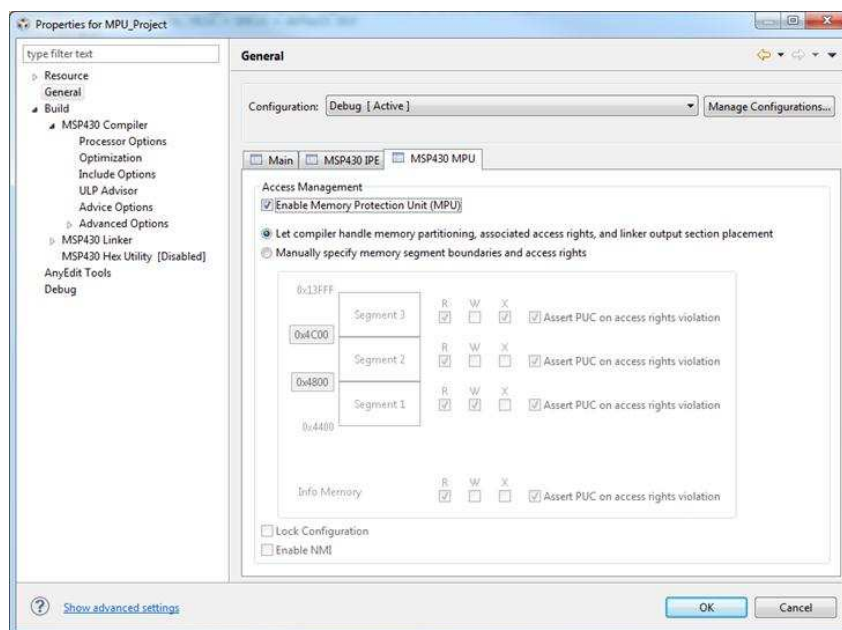


Figure 5. MPU Wizard Under CCS Project Properties

Enable the MPU by checking the *Enable Memory Protection Unit (MPU)* box. Then, the configuration should be left at default for allowing the compiler to automatically configure and partition the memory regions based in the application usage. For example, constants are configured as read only or program code is configured as read and execute only. The manual configuration mode is also available for more granular configuration.

When configured through the MPU Wizard, the C startup routine automatically configures and enables the MPU before entering `main()` without any additional steps needed by you.

6.4.2 IAR

IAR's IDE option of configuring the MPU via the MPU Wizard is located under *Project Options* → *General Options* → *MPU/IPE* as depicted in [Figure 6](#). Enable the MPU by checking the *Support MPU* box. Once enabled, the IAR toolchain automatically determines which segments are code, constants, and variables to establish how the MPU partitions should be configured.

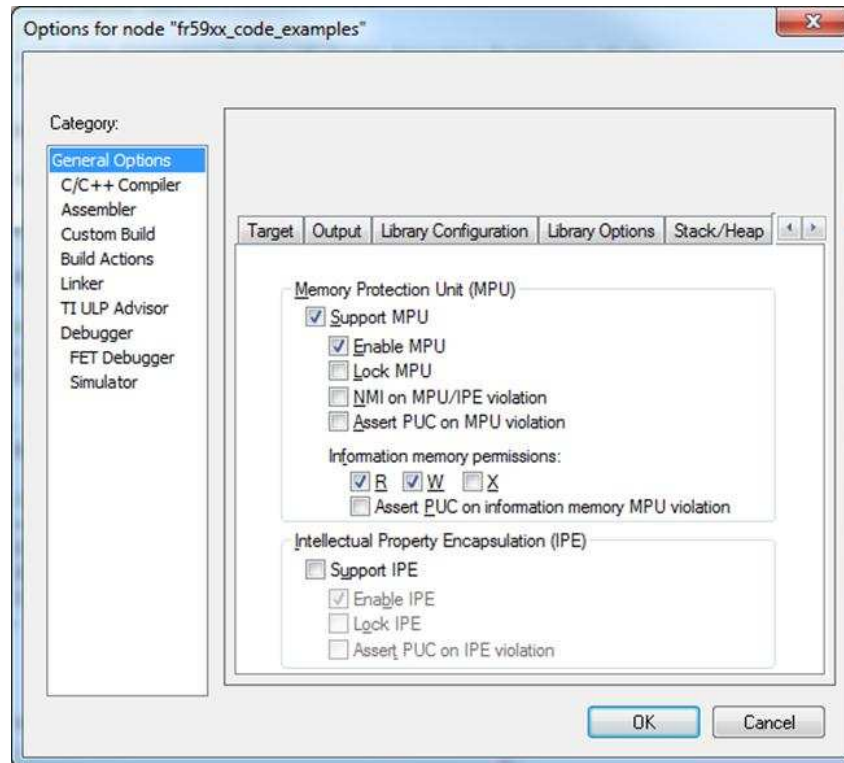


Figure 6. MPU and IPE Wizard in IAR

6.5 IP Encapsulation

Most MSP430 devices with FRAM technology except the MSP430FR57xx family of devices have built-in IP Encapsulation (IPE) capabilities. To determine whether your device has this feature or not, see the device-specific data sheet. When enabled, the IPE module can be used to protect critical pieces of code, configuration data, or secret keys in FRAM memory from being easily accessed or viewed. Once enabled, during JTAG debug, the bootstrap loader (BSL), or DMA access, read access to the IPE region would result in 0x3FFF being returned while protecting its actual underlying memory contents. To access anything inside the IPE region, the program code needs to branch or call functions stored in that segment. Only program code inside the IPE region code is able to access any data stored in this segment. Any direct data access into the IPE region from a non-IPE region would cause a violation. To enable IPE, the following code snippet could be used:

```
// 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(MPUIPCLOCK + 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()
```

Using the IPE structure, as shown in the above code snippet, allows the device's boot code to initialize IPE before any code is executed. This provides optimal IP protection as unauthorized code is not allowed to sniff data from this region. A good rule of thumb in implementing the IPE structure is to allocate it inside the IPE region itself. This prevents malicious code to modify the IPE structure or learn how the IPE is structured. More details on the operation of the IPE module can be found in the *Memory Protection Unit (MPU)* chapter of the device-specific family user's guide.

7 References

- *MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family User's Guide* ([SLAU367](#))
- *MSP430FR57xx Family User's Guide* ([SLAU272](#))
- *MSP430 Optimizing C/C++ Compiler User's Guide* ([SLAU132](#))
- IAR Embedded Workbench for MSP430 C/C++ Compiler User's Guide (IAR): <http://www.iar.com/Products/IAR-Embedded-Workbench/TI-MSP430/User-guides/>
- *MSP430™ FRAM Quality and Reliability* ([SLAA526](#))

Appendix A FRAM Device Support in TI CCSv6.0.0

Since the MSP430 device support package that ships with TI's Code Composer Studio v6.0.0 IDE at the time of publishing this document contains an improper linker memory layout and configuration setup that can lead to issues during development affecting all MSP430FR5xx and MSP430FR6xx devices. Furthermore, the mechanism that automatically places variables declared with the persistent mechanism into FRAM, as discussed in [Section 5.1](#), is not yet available in this version. The next sections outline how to identify affected CCS installations, possible failure scenarios, and recommended solutions.

A.1 Verifying MSP430 Device Support in CCS

The MSP430 device support package is an undateable component within CCS; it is necessary to verify the version of this specific package in addition to the version of the CCS IDE itself to determine whether a CCS installation is affected or not. For this, within the Code Composer Studio IDE, select the *Help* → *About Code Composer Studio* menu item.

[Figure 7](#) shows the version of the CCS IDE that is currently installed, in this case 6.0.0 build 190. If your version dialog shows this exact version, proceed to the next step. If your version is later than this, the concerns discussed in [Appendix A](#) do not apply to your CCS installation.



Figure 7. Code Composer Studio Currently Installed

After selecting *Installation Details* and navigating to the *MSP430 Emulators* line item, the number in the Version column indicates the version of the MSP430 device support package that is currently installed. In the example screen capture shown here, this version is 6.0.0.12. If the version shown in your dialog is anything earlier than (but not including) 6.0.0.14, as shown in [Figure 8](#), you are affected by the issues outlined in this Appendix. If the version shown is 6.0.0.14 or later then you already have an updated CCS installation that does not exhibit the issue discussed in this Appendix.

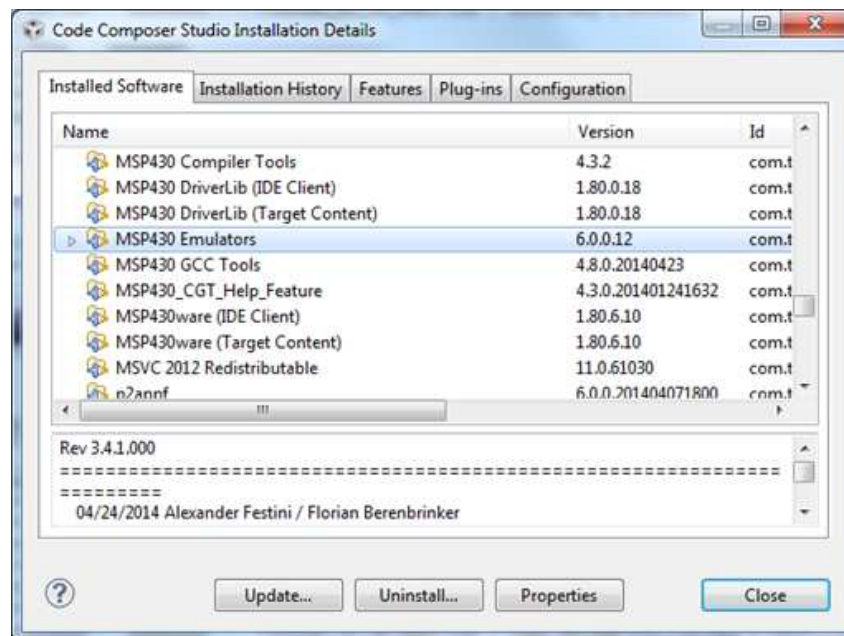


Figure 8. MSP430 Device Support Package Version Currently Installed

A.2 Failure Scenario 1

Project builds with no problems or warnings shown. When downloading the project to the target device, the debugger reports that the device is "Running"; whereas, the expected behavior would be that the device gets halted at the `main()` program entry point after the download has finished waiting for user inputs. It is also possible that the device crashes when an interrupt function is activated. This failure mode mainly applies to device variants with 96KB and 128KB of FRAM and is caused by missing linker command file constraints that are needed to assure that C low-level initialization code, interrupt vector routines, and certain other vital memory segments are allocated into the lower 64KB of the address space as required by the MSP430 device architecture.

A.3 Failure Scenario 2

The project build fails indicating an error similar to the one shown below although there is sufficient device memory available for the entire project file to fit. This failure mode mainly applies to device variants with 64KB of FRAM and is caused by the inability of the linker to use memory located above 0x10000 once the memory in the lower 64KB of address space has been filled up.

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

A.4 Failure Scenario 3

Variables declared with `#pragma PERSISTENT` do not retain their value during a power cycle of the device as discussed in [Section 5.1](#). This issue applies to all FRAM device variants and is caused by a missing linker command file definition for the `.TI.persistent` memory segment.

A.5 Failure Scenario 4

The project makes use of the CCS IDE's MPU or IPE configuration mechanisms (see [Section 6.4.1](#)) and builds with no problems or warnings shown. However, when running the code, the application may crash right away or it may work but the device's memory contents may have gotten modified during device startup.

This failure mode applies to all MSP430FRxxx devices and is caused by an improper stackpointer initialization in a low-level startup routine.

A.6 Solution/Workaround

TI is currently working on an updated MSP430 device support package addressing the outlined issues, which will be deployed to all CCSv6.0.0 customers through the CCS IDE's built-in update mechanism. The updated support package will have a version number of 6.0.0.14 and its deployment is targeted for the late June/early July 2014 time frame. For information on how to identify your installed support package, see [Section A.1](#).

While it is possible to manually modify the linker command files of an affected CCS installation to work around the outlined issues, this process is non-trivial due to major changes that would be required all while considering dependencies that the CCS IDE-managed MPU and IPE configuration process imposes. It is, therefore, highly recommended to wait for the CCS update to be deployed to fully address the outlined issues. In the meantime, it is recommended to keep the size of combined code and data on affected devices below 47KB. Furthermore, it is recommended to refrain from enabling CCS' MPU and IPE configuration dialogs and instead use a manual approach for setting up these modules.

If this is not a workable solution for the meantime, contact TI Support (<http://www.ti.com>). Furthermore, the use of an alternate toolchain such as IAR Embedded Workbench can also be considered.

Note that once TI has updated the MSP430 device support package, you should migrate your project to use the latest linker command file. To do this:

1. Create an empty CCS project corresponding to your MSP430FR5xx/6xx device variant.
2. Move the linker command file that was automatically populated into the newly created project over to your own project.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI's terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers' products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers' products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have **not** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products

Audio	www.ti.com/audio
Amplifiers	amplifier.ti.com
Data Converters	dataconverter.ti.com
DLP® Products	www.dlp.com
DSP	dsp.ti.com
Clocks and Timers	www.ti.com/clocks
Interface	interface.ti.com
Logic	logic.ti.com
Power Mgmt	power.ti.com
Microcontrollers	microcontroller.ti.com
RFID	www.ti-rfid.com
OMAP Applications Processors	www.ti.com/omap
Wireless Connectivity	www.ti.com/wirelessconnectivity

Applications

Automotive and Transportation	www.ti.com/automotive
Communications and Telecom	www.ti.com/communications
Computers and Peripherals	www.ti.com/computers
Consumer Electronics	www.ti.com/consumer-apps
Energy and Lighting	www.ti.com/energy
Industrial	www.ti.com/industrial
Medical	www.ti.com/medical
Security	www.ti.com/security
Space, Avionics and Defense	www.ti.com/space-avionics-defense
Video and Imaging	www.ti.com/video

TI E2E Community

e2e.ti.com