Application Report

MSP430™ FRAM Technology – How To and Best Practices

William Goh, Andreas Dannenberg, Johnson He

ABSTRACT

FRAM is a nonvolatile 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.

Table of 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........ 10
5 Ease-of Use Compiler Extensions for FRAM........ 10
6 FRAM Protection and Security........ 12
7 References........ 19
8 Revision History........ 19

List of Figures

Figure 3-1. Manual Override Linker Command File for IAR........ 7
Figure 6-1. IAR Project Option to Enable Map File........ 13
Figure 6-2. Address to be Written in MPUSEGBx Registers........ 14
Figure 6-3. Registers Window Showing MPU Enabled........ 15
Figure 6-4. MPU Wizard Under CCS Project Properties........ 16
Figure 6-5. MPU and IPE Wizard in IAR........ 17
Figure 6-6. FR215x and FR235x Memory Protection........ 18

List of Tables

Table 3-1. MSP430 C/C++ Data Types........ 2
Table 3-2. MSP430FR2311 Memory Organization........ 4
Table 3-3. Summary of Segments in IAR........ 7
Table 6-1. Memory Segmentation Inside CCS Map File........ 12
Table 6-2. MPU Memory Segmentation........ 13
Table 6-3. MPU Memory Segmentation Example........ 14

Trademarks

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

Copyright © 2021 Texas Instruments Incorporated

1 FRAM and Universal Memory

FRAM is a nonvolatile 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*.

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 nonvolatile. 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, while 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

Because 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 the application memory layout to make the 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.

In the MSP430 MCU, a reasonable data type can be defined according to the size of the data used. Table 3-1 lists the commonly used data types. For more details, see MSP430 Optimizing C/C++ Compiler.

Table 3-1. MSP430 C/C++ Data Types

Type	Size	Alignment	Representation	Range	
				Minimum	Maximum
signed char	8 bits	8	Binary	-128	127
char	8 bits	8	ASCII	0 or -128	255 or 127
unsigned char	8 bits	8	Binary	0	255
bool(C99)	8 bits	8	Binary	0(false)	1(true)
short, signed short	16 bits	16	Binary	-3 2768	32 767
unsigned short	16 bits	16	Binary	0	65 535
int, signed int	16 bits	16	Binary	-3 2768	32 767
unsigned int	16 bits	16	Binary	0	65 535
long, signed long	32 bits	16	Binary	-2 147 483 648	2 147 483 647
unsigned long	32 bits	16	Binary	0	4 294 967 295
long long, signed long long	64 bits	16	Binary	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 bits	16	Binary	0	18 446 744 073 709 551 615
float	32 bits	16	IEEE 32-bit	1.175 494e-38	3.40 282 346e+38
double	64 bits	16	IEEE 64-bit	2.22 507 385e-308	1.79 769 313e+308
long double	64 bits	16	IEEE 64-bit	2.22 507 385e-308	1.79 769 313e+308

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.

Copyright © 2021 Texas Instruments Incorporated

3.2 Variables

Variables are allocated in SRAM by the default linker command files. FRAM-based MSP430 devices would have 0.5KB to 8KB 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 show cases 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.

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 nonvolatile 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                */
```


Copyright © 2021 Texas Instruments Incorporated

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. 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             */
```

The linker command file can be directly modified in CCS. The following code provides an explanation and modification example of the .cmd file for MSP430FR2311.

In the linker file, memory allocation contains two contents : MEMORY and SECTIONS. MEMORY mainly divides the memory into RAM, FRAM, BSL or user-defined area, and SECTIONS selects the defined memory segment for each used section. For more details, see MSP430 Optimizing C/C++ Compiler.

Table 3-2. MSP430FR2311 Memory Organization

	Access	MSP430FR2311
Memory (FRAM) Main: interrupt vectors and signatures Main: code memory	Read/Write (Optional Write Protect)	3.75KB FFFFh to FF80h FFFFh to F100h
RAM	Read/Write	1KB 23FFh to 2000h
Bootloader (BSL1) Memory (ROM) (TI Internal Use)	Read only	2KB 17FFh to 1000h
Bootloader (BSL2) Memory (ROM) (TI Internal Use)	Read only	1KB FFFFFh to FFC00h
Peripherals	Read/Write	4KB 0FFFh to 0000h

Table 3-2 shows the memory organization for MSP430FR2311, the following program is the default memory and section allocation depend on memory organization of the chip in CCS. In the memory allocation, the initial address and length of the memory need to be defined using the 'origin' and 'length' keywords. SECTIONS can directly point to the already allocated MEMORY, users can define the MEMORY size and the MEMORY type of SECTIONS as needed.

```
/****************************************************************************/
/* SPECIFY THE SYSTEM MEMORY MAP                                            */
/****************************************************************************/
 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
  }

/****************************************************************************/
/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY                              */
/****************************************************************************/
 SECTIONS {
  GROUP(ALL_FRAM)
  {
    GROUP(READ_WRITE_MEMORY)
    {
      .TI.persistent : {}              /* For #pragma persistent            */
```


Copyright © 2021 Texas Instruments Incorporated

```
    }
    GROUP(READ_ONLY_MEMORY)
    {
      .cinit      : {}                  /* Initialization tables            */
      .pinit      : {}                  /* C++ constructor tables           */
      .binit      : {}                  /* Boot-time Initialization tables  */
      .init_array : {}                  /* C++ constructor tables           */
      .mspabi.exidx : {}                /* C++ constructor tables           */
      .mspabi.extab : {}                /* C++ constructor tables           */
      .const      : {}                  /* Constant data                    */
    }
    GROUP(EXECUTABLE_MEMORY)
    {
      .text       : {}                  /* Code                             */
      .text:_isr  : {}                  /* Code ISRs                        */
     }
  } > 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 buffer                     */
  .sysmem     : {} > RAM                /* Dynamic memory allocation area   */
  .bss        : {} > RAM                /* Global & static vars             */
  .data       : {} > RAM                /* Global & static vars             */
  .TI.noinit  : {} > RAM                /* For #pragma noinit               */
  .stack      : {} > RAM (HIGH)         /* Software system stack            */

   /* MSP430 interrupt vectors */
   .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 Modify the Linker File Example 1

According to the description of Section 2, FRAM can be used as RAM, this example is to allocate 0.5KB of FRAM as RAM.

The FRAM memory segment defined as RAM2 in the following program.

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

And the '|' operation can be used in SECTIONS to allocate the .bss and .data segments into the RAM or RAM2 memory.

```
  .bss        : {} > RAM | RAM2         /* Global & static vars             */
  .data       : {} > RAM | RAM2         /* Global & static vars             */
```


Copyright © 2021 Texas Instruments Incorporated

Note

The write function of FRAM must be enable in program if there are some write operation within the allocated FRAM memory.

3.4.1.2 Modify the Linker File Example 2

This example defines the TI.noinit segment into the FRAM memory, which can be defined as a function that does not erase when power is lost.

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

Note

The write function of FRAM must be enable in program if there are some write operation within the allocated FRAM memory.

3.4.1.3 Modify the Linker File Example 3

The function of defining variables in the specified memory area can be realized by modifying the linker file. The following steps and example outline how to achieve this function.

1. Allocate a new memory block(MY_SECTION).
2. Define a segment(.Image) that store in this memory block(MY_SECTION).
3. Use #pragma DATA_SECTION in program to define variables in this segment.

```
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

The write function of FRAM must be enable in program if there are some write operation within the allocated FRAM memory.

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.

The IAR 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 https://www.iar.com/support/user-guides/user-guidesiar-embedded-workbench-for-ti-msp430/.

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 link430xxxx.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.

Copyright © 2021 Texas Instruments Incorporated

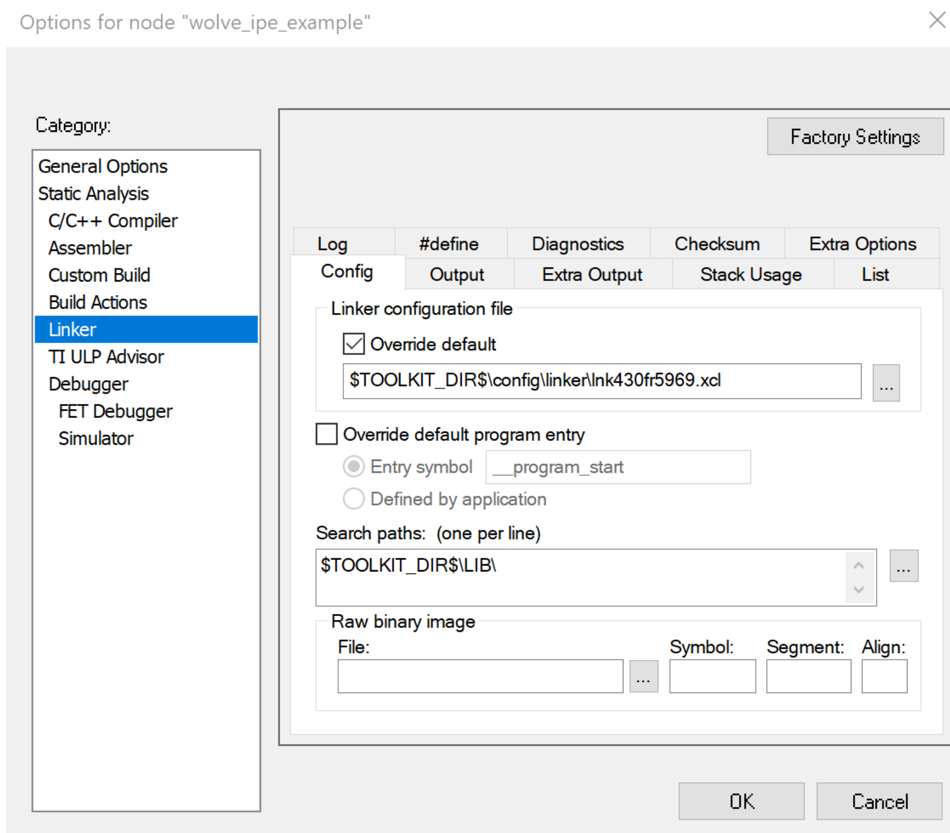

Figure 3-1. Manual Override Linker Command File for IAR

The linker command file can be directly modified in IAR. The following code provides an explanation and modification example of the .xcl file for MSP430FR2311.

Table 3-3 show some segments in IAR, these segments can be used to allocate memory. More segments description, see the chapter of *Segment Reference* in IAR C/C++ Compiler User Guide(For MSP430).

Table 3-3. Summary of Segments in IAR

Segment	Description
DATA16_AC	Holds __data16 located constant data.
DATA16_AN	Holds __data16 located uninitialized data.
DATA16_C	Holds __data16 constant data.
DATA16_HEAP	Holds the heap used for dynamically allocated data in data16 memory.
DATA16_I	Holds __data16 static and global initialized variables.
DATA16_ID	Holds initial values for __data16 static and global variables in DATA16_I.
DATA16_N	Holds __no_init __data16 static and global variables.
DATA16_P	Holds __data16 variables defined with the __persistent keyword.
DATA16_Z	Holds zero-initialized __data16 static and global variables.
DATA20_AC	Holds __data20 located constant data.
DATA20_AN	Holds __data20 located uninitialized data.
DATA20_C	Holds __data20 constant data.
DATA20_HEAP	Holds the heap used for dynamically allocated data in data20 memory.
DATA20_I	Holds __data20 static and global initialized variables.
DATA20_ID	Holds initial values for __data20 static and global variables in DATA20_I.
DATA20_N	Holds __no_init __data20 static and global variables.
DATA20_P	Holds __data20 variables defined with the __persistent keyword.

Copyright © 2021 Texas Instruments Incorporated

Table 3-3. Summary of Segments in IAR (continued)

Segment	Description
DATA20_Z	Holds zero-initialized __data20 static and global variables.

The memory block allocation can be directly to each section in IAR, and the memory space can be discontinuous, separated by ', ' symbol. If no address space is given after the -Z command, the default is the same as the previous line. Depend on memory organization of MSP430FR2311 in Table 3-2, the default memory allocation file is provided in IAR. The main contents are as shown in the following code, which are divided into RAM, FRAM, Vectors segment and so on. For more information on memory allocation, see the IAR C/C++ Compiler User Guide (For MSP430).

```
// -----------------------------------------------
// RAM memory
//
  -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 memory
//
// 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

// Constant data
  -Z(CONST)DATA16_C,DATA16_ID,TLS16_ID,DIFUNCT,CHECKSUM=F100-FF7F
  -Z(CONST)DATA20_C,DATA20_ID

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

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

3.4.2.1 Modify the Linker File Example 1

This example is to allocate 0.5KB of FRAM as RAM. Therefore, the 0xF100 to 0xF2FF memory segment is added to the memory allocation in RAM, and the FRAM memory segment starts from 0xF300.

```
// -----------------------------------------------
// RAM memory
//
  -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 memory
//
// Read/write data in FRAM
  -Z(CONST)DATA16_P,DATA20_P=F300-FF7F
  -Z(DATA)DATA16_HEAP+_DATA16_HEAP_SIZE,DATA20_HEAP+_DATA20_HEAP_SIZE

// Constant data
  -Z(CONST)DATA16_C,DATA16_ID,TLS16_ID,DIFUNCT,CHECKSUM=F300-FF7F
  -Z(CONST)DATA20_C,DATA20_ID

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

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


Copyright © 2021 Texas Instruments Incorporated

Note

The write function of FRAM must be enable in program if there are some write operation within the allocated FRAM memory.

3.4.2.2 Modify the Linker File Example 2

This example defines the DATA16_N and DATA20_N segment into the FRAM memory, which can be defined as a function that does not erase when power is lost.

```
// ---------------------------------------------
// RAM memory
//
  -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

The write function of FRAM must be enable in program if there are some write operation within the allocated FRAM memory.

3.4.2.3 Modify the Linker File Example 3

The function of defining variables in the specified memory area can be realized by modifying the linker file. The following steps and example outline how to achieve this function.

1. Define a new segment(MY_SEGMENT) and allocate some memory space.
2. Use '@' symbol in program to define variable.

```
  -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

The write function of FRAM must be enable in program if there are some write operation within the allocated FRAM memory.

Copyright © 2021 Texas Instruments Incorporated

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 nonvolatile 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 (tens 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.

Note

To allow the compiler to better support the FRAM family of MCUs and better use their internal resources, it is recommended to use the higher version of CCS and IAR development software. It is strongly recommended to use CCS version 9.0.1 and IAR version 7.12.13.

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. For more detailed information on these pragma directives, see the *MSP430 Optimizing C/C++ Compiler User's Guide*.

`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

Copyright © 2021 Texas Instruments Incorporated

the C startup initialization process. Declaring variables as PERSISTENT causes them to get allocated into the .TI.persistent linker memory segment.

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* available from https://www.iar.com/support/user-guides/user-guidesiar-embedded-workbench-for-ti-msp430/.

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;
```


Copyright © 2021 Texas Instruments Incorporated

6 FRAM Protection and Security

6.1 Memory Protection

FRAM is easy to write. 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. Some 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, and others have register that can control FRAM memory write protection.

Note

It is very important to always appropriately configure and enable the memory protection before any software deployment or production code release to ensure maximum application robustness and data integrity. The memory protection 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.

For MCUs of the FR5xx and FR6xx series, the MPU be able to protect variables, constants, and program code. 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, and MSP430FR6xx Family User's Guide* or the *MSP430FR57xx Family User's Guide* before proceeding with this section.

For MCUs of the FR2xx and FR4xx series, the write protection of the FRAM is implemented using the SYSCFG0 register. It is recommended to first read the *System Resets, Interrupts, and Operating Modes, System Control Module (SYS)* chapter in the *MSP430FR4xx and MSP430FR2xx Family User's Guide* 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 TI Code Composer Studio

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 6-1 shows the starting addresses for each of these segments.

Table 6-1. Memory Segmentation Inside CCS Map File

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

6.2.2 IAR Embedded Workbench for MSP430

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 6-1.

Copyright © 2021 Texas Instruments Incorporated

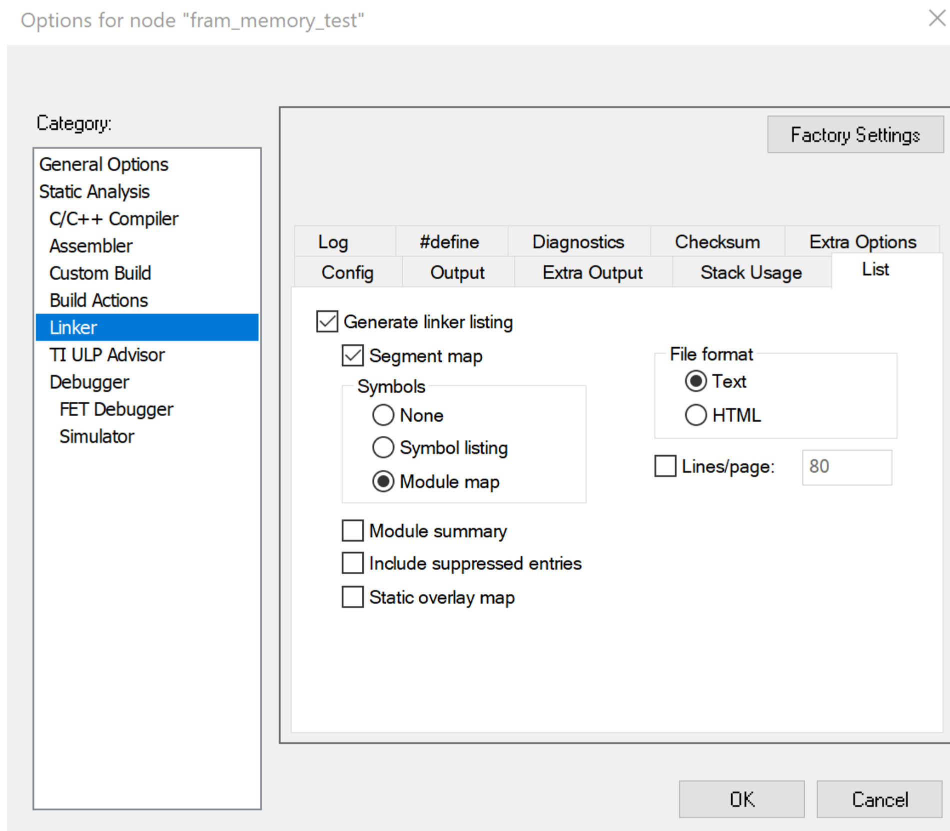

Figure 6-1. IAR Project Option to Enable Map File

When 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 3-3 shows several of the general segment names that are used by IAR.

6.3 Memory Protection Setting

6.3.1 FR5xx and FR6xx Series MCU 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 6-2 summarizes the typical memory segmentation.

Table 6-2. 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 6-3.

Copyright © 2021 Texas Instruments Incorporated

Table 6-3. MPU Memory Segmentation Example

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

When the memory segmentation has been decided for segment 1, 2, and 3, as shown in Table 6-3, 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 6-2 shows the application example of how the memory has been partitioned and MPU registers are configured.

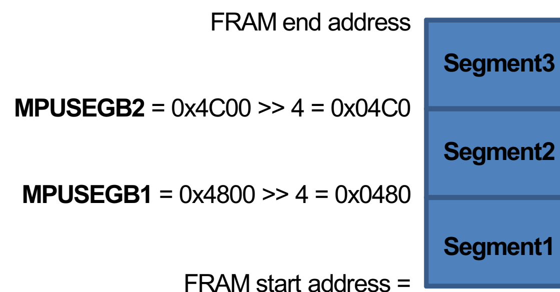

Figure 6-2. Address to be Written in MPUSEGBx Registers

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 6.3.1.1 and Section 6.3.1.2 outline the steps for CCS and IAR, respectively.

6.3.1.1 CCS MPU Implementation

6.3.1.1.1 Manual MPU Configuration

Create a new C-file called *system_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
```


Copyright © 2021 Texas Instruments Incorporated

```
    /*==================================*/
    /* 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 (see Figure 6-3) showing the MPU register contents being configured.

MPU		
MPUCTL0	0x9611	MPU Control Register 0 [Memor
MPUCTL1	0x0000	MPU Control Register 1 [Memor
MPUSEGB2	0x04C0	MPU Segmentation Border 2 Re
MPUSEGB1	0x0480	MPU Segmentation Border 1 Re
MPUSAM	0x0513	MPU Access Management Regis
MPUIPC0	0x0000	MPU IP Control 0 Register [Mem
MPUIPSEGB2	0x0000	MPU IP Segment Border 2 Regis
MPUIPSEGB1	0x0000	MPU IP Segment Border 1 Regis

Figure 6-3. Registers Window Showing MPU Enabled

6.3.1.1.2 IDE Wizard-Based MPU Configuration

Figure 6-4 shows the built-in MSP430 MPU Wizard in Code Composer Studio v9, which is accessible through the CCS Project Properties. To open this dialog, right-click on the project in the CCS Project Explorer view and select *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.

Copyright © 2021 Texas Instruments Incorporated

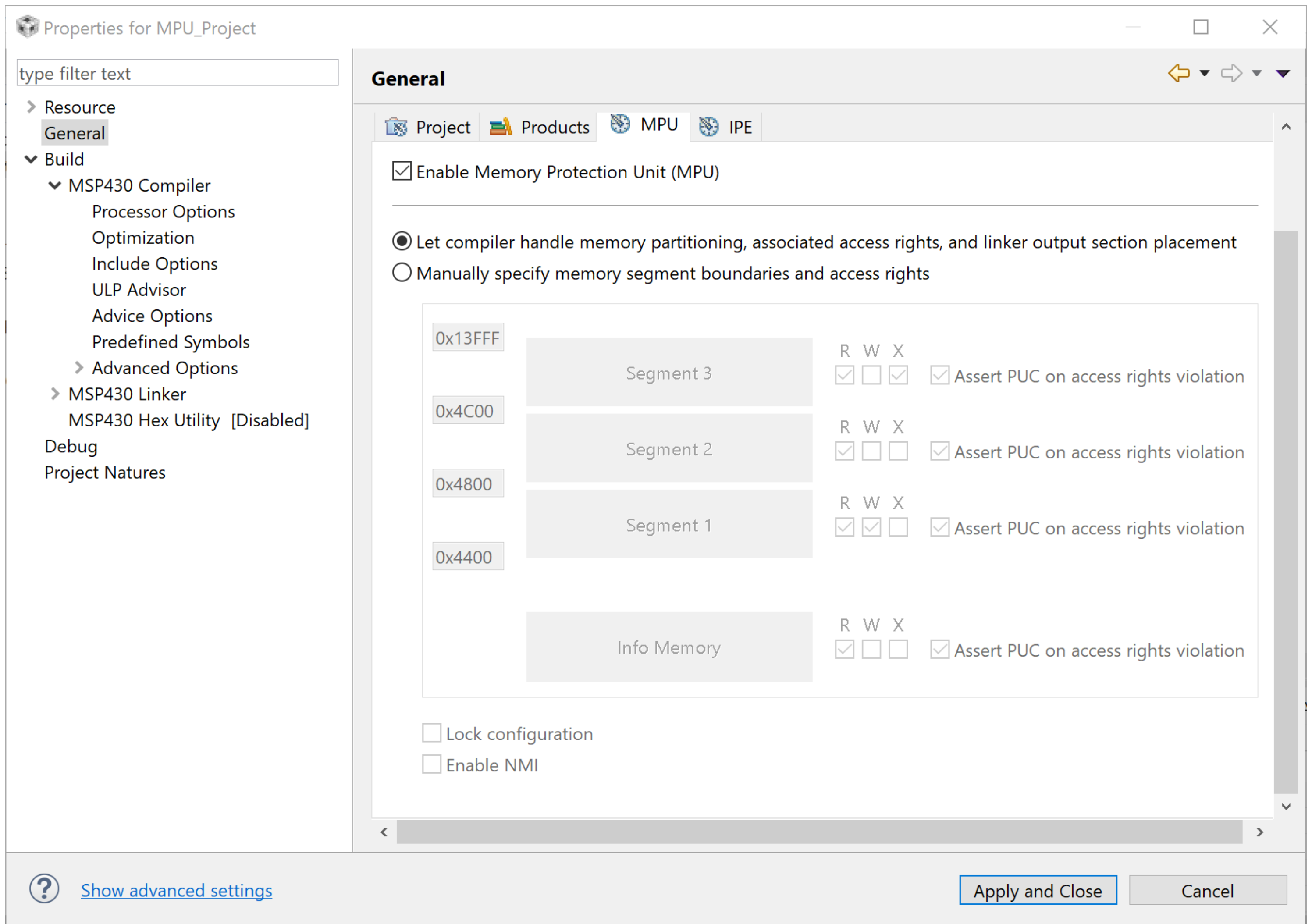

Figure 6-4. MPU Wizard Under CCS Project Properties

6.3.1.2 IAR MPU Implementation

6.3.1.2.1 Manual MPU Configuration

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. The IAR 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
```


Copyright © 2021 Texas Instruments Incorporated

```
  /*
   * Return value:
   *
   *  1 - Perform data segment initialization.
   *  0 - Skip data segment initialization.
   */

  return 1;
}
```

6.3.1.2.2 IDE Wizard-Based MPU Configuration

The IAR IDE option of configuring the MPU through the MPU Wizard is located under *Project Options → General Options → MPU/IPE/FRWP* as shown in Figure 6-5. 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.

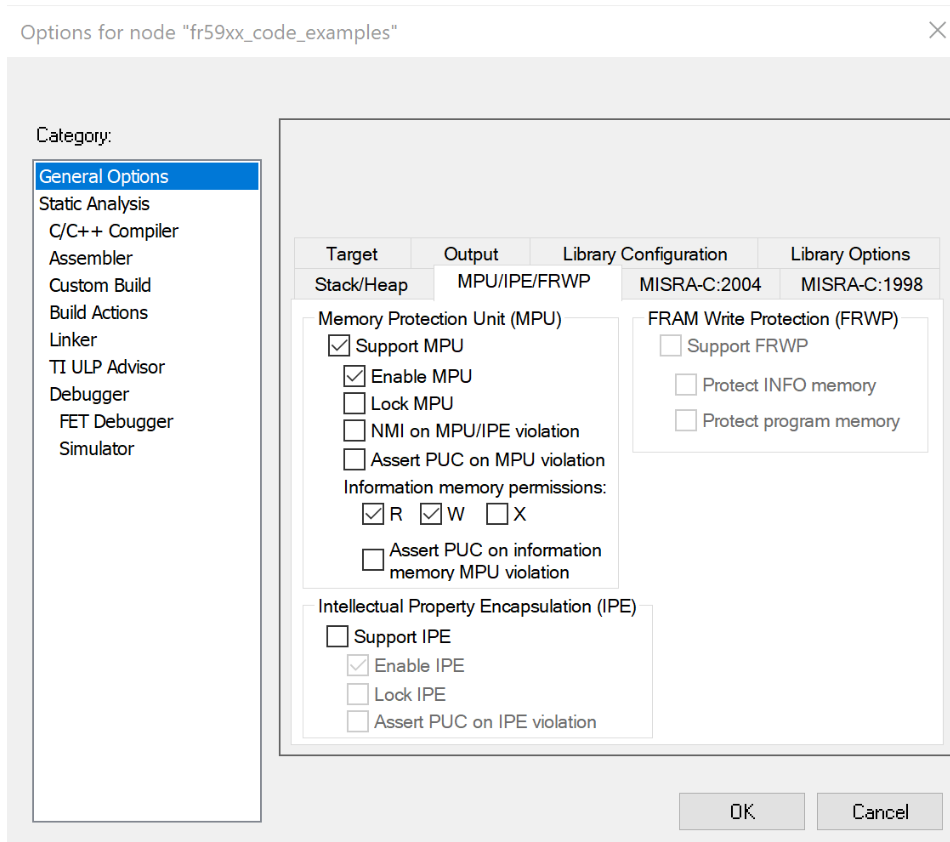

Figure 6-5. MPU and IPE Wizard in IAR

6.3.2 FR2xx and FR4xx Series MCU FRAM Write Protection Configuration

For FR2xx FR4xx series MCUs, except for the FR235x and FR215x devices, can operate the PFWP and DFWP register to enable or disable the write protection for main memory and the Information Memory in the FRAM. After a PUC reset, those bit defaults to 1, and writes to FRAM are disabled. User code must write the correct password and clear the corresponding bit before write operation.

In FR215x and FR235x devices, the program FRAM can be partially protected by an offset specified by FRWPOA from the starting address of main FRAM memory. When PFWP is set, the main memory after this offset is protected, and the part before this address is unprotected. This unprotected range can be used like RAM for random frequent writes. The 6-bit FRWPOA can specify the offset from 0KB to 63KB with 1KB resolution. After reset, FRWPOA defaults to zero, and the entire program FRAM is under the protection of PFWP. FRWPOA can be modified in parallel when FRWPPW is correctly written. Figure 6-6 shows how the data and program FRAM are protected in these devices.

Copyright © 2021 Texas Instruments Incorporated

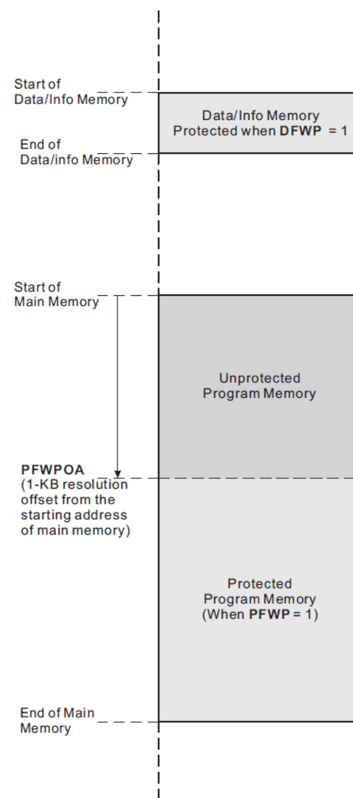

Figure 6-6. FR215x and FR235x Memory Protection

6.4 IP Encapsulation

IP Encapsulation (IPE) is a feature found on some MSP430 FRAM microcontrollers, like the MSP430FR59xx/69xx family. 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 bootloader (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 learn more about IP Encapsulation, refer to *MSP Code Protection Features*.

Copyright © 2021 Texas Instruments Incorporated

7 References

- *MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide*
- *MSP430FR57xx Family User's Guide*
- *MSP430FR4xx and MSP430FR2xx Family User's Guide*
- *MSP430 Optimizing C/C++ Compiler User's Guide*
- *IAR Embedded Workbench for MSP430 C/C++ Compiler User's Guide*
- *MSP430™ FRAM Quality and Reliability*

8 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Revision A (June 2020) to Revision B (August 2021)	Page
• Updated the numbering format for tables, figures and cross-references throughout the document	2

Changes from June 23, 2014 to June 30, 2020	Page
• Added Table 3-1, *MSP430 C/C++ Data Types*	2
• Changes throughout document, including updated figures, for current versions of IDEs	3

Copyright © 2021 Texas Instruments Incorporated

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (https:www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2021, Texas Instruments Incorporated