Application Report

SLAA685–December 2015

MSP Code Protection Features

Katie Pier

MSP Applications

ABSTRACT

MSP microcontrollers (MCUs) offer a number of features to help control code accessibility in the device, to add different layers of code access management and protection strategies. These include features that can lock or password protect the JTAG/SBW access, IP Encapsulation (IPE) to isolate sensitive code with different permissions than the rest of the program, and bootloader (BSL) access features for field firmware updates. This application report details some of the features available in different MSP device families and considerations that can be taken to add additional layers of protection to the device.

Related source code and additional information is available from http://www.ti.com/lit/zip/slaa685.

Contents

List of Figures

List of Tables

Trademarks

MSP430, MSP432, Code Composer Studio are trademarks of Texas Instruments.
IAR Embedded Workbench is a trademark of IAR Systems.

1 Introduction

As more products include a microcontroller (MCU) or other embedded device, embedded security is becoming more and more important. A common concern for embedded system applications is preventing readout of the device in order to help protect the intellectual property (IP) of proprietary code. While there is no perfect solution to protect against unauthorized access or reverse engineering of code, designers can take precautions to make their code more difficult to access without more sophisticated techniques or equipment. Using a layered approach with different methodologies for trying to control code access is typically a good idea to add layers of security against someone trying to read out the device. This application report discusses some features in the MSP430™ and MSP432™ microcontroller families that can be used to help provide some of these layers—locking JTAG, using IP Encapsulation (on devices that support this feature), and bootloader (BSL) security options. Using a combination of these methodologies helps to increase the layered protection of the device, to increase the difficulty for anyone trying to read the device.

NOTE: This application report primarily covers features of the MSP430 microcontrollers. For information on securing an MSP432 microcontroller or using the bootloader on the MSP432 microcontrollers, see the application report *Configuring Security and Bootloader (BSL) on MSP432P4xx* (SLAA659). For the IP Protection secure zones feature found on MSP432 devices, see the application report *Software IP Protection on MSP432P4xx Microcontrollers* (SLAA660).

2 Locking JTAG Across Different MSP Families

Table 1 lists the locking features that are available in each MSP family.

Table 1. JTAG Locking Features Across MSP Families

Device Family	Physical Fuse	Boot Override	Electronic Fuse	JTAG Signatures	Reversible	JTAG Access With Password
MSP430F1xx/F2xx/F4xx	✓					
MSP430F5xx/F6xx			✓	0x17FC–0x17FF	✓ with bootloader	
MSP430FR5xx/FR6xx			✓	0xFF80–0xFF83	✓ with bootloader	✓
MSP430FR2xx/FR4xx			✓	0xFF80–0xFF83	✓ with bootloader	
MSP430i2xx			✓	0xFFDC–0xFFDF	No factory bootloader available	
MSP432Pxx		✓			✓ with boot override factory reset	

2.1 Physical JTAG Fuse (F1xx/F2xx/F4xx)

MSP430F1xx, F2xx, and F4xx family devices JTAG can be secured through a physical JTAG security fuse. The fuse is blown by the programming tool after JTAG or SBW programming. The tool applies a fuse blowing voltage (6.5 V ±0.5 V) on the TEST pin (on devices with a TEST pin) or TDI pin (on devices without a TEST pin) [see *MSP430 Programming Via the JTAG Interface* (SLAU320)] for more details. Blowing the fuse completely disables the JTAG port and is not reversible, and further JTAG or SBW access is not possible. After the physical fuse is blown, the device is accessible only through the password-protected bootloader, if supported and enabled.

2.2 Electronic Fuse or Lock Without Password

Most MSP430 devices have an electronic fuse, sometimes called the e-Fuse or "JTAG Lock Without Password". The JTAG/SBW interface is locked by setting a value into 2 words in memory, the JTAG signatures. After the JTAG lock signatures are programmed and the device is reset, the device cannot be accessed through JTAG or SBW. The device will only be accessible through password-protected bootloader. It is possible to regain access to the device by using the BSL to access the device and clear the JTAG signatures, but this requires either the correct BSL password, or doing a mass erase of the device (depending on the device family). The implementation of the e-Fuse varies slightly amongst MSP430 subfamilies, as explained in the following sections.

2.2.1 F5xx/F6xx Electronic Fuse Implementation

On MSP430 F5xx/F6xx devices, the JTAG signatures are located in the bootloader memory area (F5xx/F6xx devices contain a Flash-based BSL in a protected area of flash memory) at addresses 17FCh–17FFh. If anything other than 00000000h or FFFFFFFFh is programmed to these addresses, then the JTAG/SBW interface is locked. To program these addresses, the protected area of BSL flash must first be unlocked by clearing the SYSBSLPE bit in the SYSBSLC register. After programming the signatures, the BSL protection should be re-enabled.

To clear JTAG/SBW lock protection, the BSL can be used to clear the JTAG signatures to 00000000h. The BSL is password protected by the last 32 bytes of the interrupt vector table FFE0h–FFFFh (see Section 4). Because the JTAG signature is located in the protected BSL area, the BSL must first clear the SYSBSLPE bit in the SYSBSLC register (write 0003h to the address 0182h), before writing 00000000h to the JTAG signatures.

Table 2. JTAG Locking on F5xx/F6xx

Name	Addresses	Value	Device Security
JTAG/SBW Signature	17FCh–17FFh(1)	FFFF_FFFFh	JTAG/SBW is unlocked.
		0000_0000h	
		Any other value	JTAG/SBW is locked.

(1) These addresses are in the protected BSL flash area, which must be unlocked by writing 0003h to clear SYSBSLPE to the address 0182h (SYSBSLC register address) before the signatures can be changed.

2.2.2 FR5xx/FR6xx Electronic Fuse Implementation (Lock without password)

On MSP430FR5xx/FR6xx devices, the JTAG signatures are located in the main area of FRAM at addresses FF80h–FF83h. The JTAG/SBW interface is locked without password by writing 55555555h to the JTAG signatures.

To clear JTAG/SBW lock protection, the bootloader can be used to clear the JTAG signatures to any values other than 5555h and AAAAh. The BSL is password protected and the last 32 bytes of the interrupt vector table (FFE0h–FFFFh) are used as the BSL password (see Section 4). Unlocking JTAG/SBW access can be easily performed with the BSL mass erase command on these FRAM devices, because the JTAG signatures are located in main memory (rather than in a protected BSL area like on F5xx/F6xx).

Table 3. JTAG Locking on FR5xx/FR6xx

Name	Addresses	Value	Device Security
JTAG/SBW Signatures	FF80h–FF83h	5555h_5555h	JTAG/SBW is locked without password.
		@FF80h = AAAAh @FF82h = password length in words	JTAG/SBW is locked with password.(1)
		Any other value	JTAG/SBW is unlocked.

(1) See Section 2.3 for more details.

2.2.3 FR2xx/FR4xx Electronic Fuse Implementation

On MSP430FR2xx/FR4xx devices, the JTAG signatures are located in the main area of FRAM at addresses FF80h–FF83h (just like on other FRxx devices). The JTAG/SBW interface is locked without password by writing anything other than 00000000h or FFFFFFFFh to the JTAG signatures (this is different from other FRxx devices).

To clear JTAG/SBW lock protection, the bootloader can be used to clear the JTAG signatures to 00000000h or FFFFFFFFh. The BSL is password protected and the last 32 bytes of the interrupt vector table (FFE0h–FFFFh) are used as the BSL password (see Section 4). Unlocking JTAG/SBW access can be easily performed with the BSL mass erase command on these FRAM devices, because the JTAG signatures are located in main memory (rather than some protected BSL area like on F5xx/F6xx).

Table 4. JTAG Locking on FR2xx/FR4xx

Name	Addresses	Value	Device Security
JTAG/SBW Signature	FF80h–FF83h	FFFF_FFFFh	JTAG/SBW is unlocked.
		0000_0000h	
		Any other value	JTAG/SBW is locked.

2.2.4 MSP430i2xx Electronic Fuse Implementation – Start-Up Code (SUC)

On MSP430i2xx devices, the JTAG/SBW device security is controlled by the Start-Up Code (SUC). The SUC must set the device as secured or unsecured within the first 64 MCLK clock cycles after a BOR or POR reset. The SUC determines if JTAG/SBW access should be enabled or disabled by checking the signatures at addresses 0xFFDC-0xFFDF.

The JTAG/SBW interface is disabled by writing the addresses FFDCh–FFDFh with any value other than 00000000h or FFFFFFFFh. The SUC checks the value in these addresses after a POR or BOR reset. If the value is different from 00000000h or FFFFFFFFh, then the SUC writes A5A5h into the SYSJTAGDIS register to disable the JTAG/SBW interface and secure the device. The SUC must complete this within 64 MCLK cycles after the BOR or POR reset.

Because the MSP430i2xx device does not have a bootloader, it is not possible to clear the addresses 0xFFDC-0xFFDF to re-enable JTAG/SBW access. If the user has implemented a custom bootloader in main memory, they can use this to write the addresses FFDCh–FFDFh to 00000000h. The SUC checks the value in these addresses after a POR or BOR reset. If the value is 00000000h or FFFFFFFFh, then the SUC does nothing and continues on to the calibration routine—after 64 MCLK cycles, the JTAG is enabled the device is unsecured automatically.

See the *MSP430i2xx Family User's Guide* (SLAU335) and the device code examples for more information about the SUC.

Table 5. JTAG Locking on i2xx

Name	Addresses	Value	Device Security
JTAG/SBW Signature	FFDCh–FFDFh	FFFF_FFFFh	JTAG/SBW is unlocked.(1)
		0000_0000h	
		Any other value	JTAG/SBW is locked.(1)

(1) The correct SUC code must be implemented to check these addresses and perform the locking or unlocking within 64 cycles of MCLK after start-up. Use the SUC code included with the i2xx code examples, which already has this implemented.

2.3 JTAG Lock With Password (FR5xx/FR6xx)

Some MSP430 FRAM devices additionally allow a JTAG lock with password option. This is different from the eFuse/JTAG Lock without Password, because with this mechanism access can be regained without using the bootloader, by providing a user-defined password through the toolchain.

On MSP430FR5xx/FR6xx devices, the JTAG signatures are located in the main area of FRAM at addresses FF80h–FF83h. The JTAG/SBW is locked with password by writing AAAAh to JTAG signature 1 (FF80h), and writing JTAG signature 2 (FF82h) with anything but 5555h. The value written into JTAG signature 2 is used to define the length in words of the desired password. The password starts at address FF88h and continues for the number of words set by JTAG signature 2. Note that if the password is long enough, it may overlap with addresses used by the interrupt vector table. The password value for these addresses should be the interrupt vector table value, and this is necessary for proper interrupt handling and code execution. The password protection takes effect on the next BOR event.

Table 6. JTAG Lock With Password on FR5xx/FR6xx

Name	Addresses	Value	Device Security
JTAG/SBW Signatures	FF80h–FF83h	5555h_5555h	JTAG/SBW is locked without password.
		@FF80h = AAAAh @FF82h = password length in words	JTAG/SBW is locked with password.
		Any other value	JTAG/SBW is unlocked.
JTAG Password	FF88h–length	User Defined + Vector Table Configuration	If JTAG/SBW is locked with password, the password value defined in these addresses must be provided by the tool-chain through the JTAG mailbox.

When a device is secured with JTAG lock with password, the tool-chain must supply the correct password to access the device. After the correct password is provided, the device is unlocked until the next BOR event occurs.

2.3.1 Using JTAG Lock With Password in CCS

The code example JTAG_lock_FR5xx_with_password.c from the zip file (http://www.ti.com/lit/zip/slaa685) shows how to set the JTAG lock with password on an MSP430FR5969 device in Code Composer Studio™ IDE (CCS). Note however that some production programming tools may also offer the option to set the password from a GUI interface instead of having to hard-code it into the project.

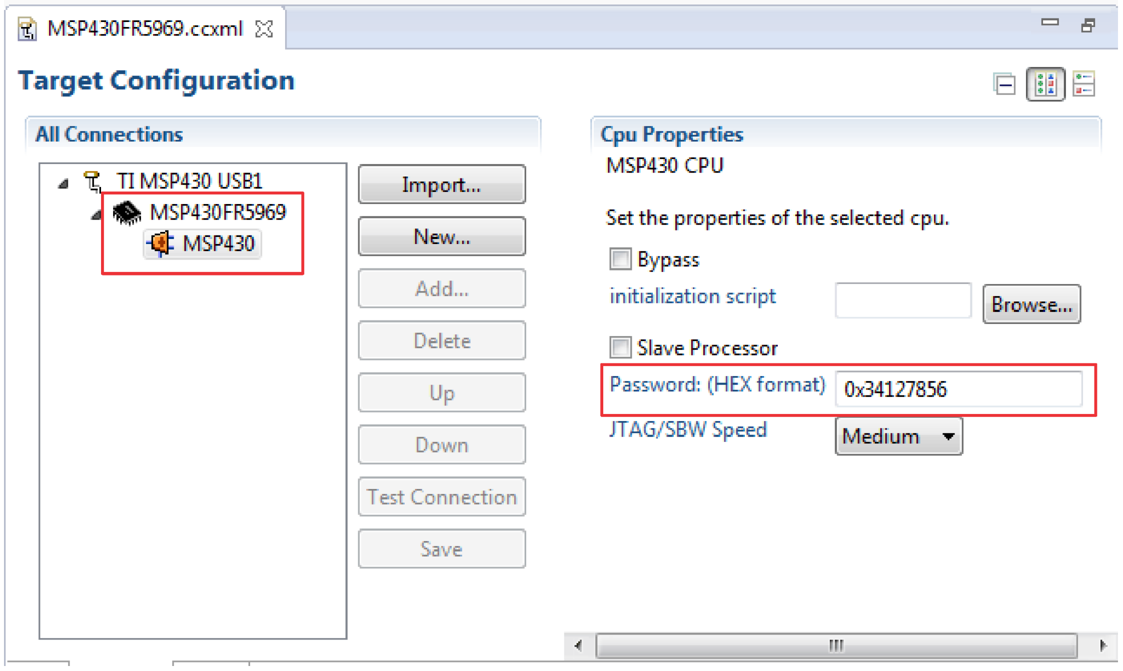
For the MSP430 TI C/C++ compiler, use #pragma DATA_SECTION to place the correct signatures into the linker file section .jtagsignature and to place the password into the section .jtagpassword. Use the #pragma RETAIN to keep the compiler from optimizing out the data (since it is not used by the program). For more information on these pragmas, see the *MSP430 TI C/C++ Compiler guide* (SLAU132).

```
#pragma RETAIN(JTAG_signatures)
#pragma DATA_SECTION(JTAG_signatures, ".jtagsignature")
const uint16_t JTAG_signatures[] = {0xAAAA, 0x0002};

...

#pragma RETAIN(JTAG_password)
#pragma DATA_SECTION(JTAG_password, ".jtagpassword")
const uint8_t JTAG_password[] = {0x12, 0x34, 0x56, 0x78};
```

The following example shows a section of the TI-txt binary file for the device locked with a 2-word password of 12h 34h 56h 78h:

```
@ff80
AA AA 02 00 FF FF FF FF 12 34 56 78 FF FF FF FF
```

After the device has been locked with password and has undergone BOR reset (at which point the settings take effect), the password has to be provided in order to regain access to the device for reprogramming. Provide the password in CCS, in the targetConfigs folder, open the .ccxml file for the device. Under "Advanced Setup" click the link "Target Configuration". Then, click on "MSP430" under the device part number, and in the box "Password: (HEX format)" enter the password to provide when unlocking the part. For the example above, the password is 12h 34h 56h 78h. The password should be written in words, starting with the least significant word, so for this example, enter 0x34127856. An easy way to find the correct order to supply the password to the toolchain is to use the Memory Browser with the view set to "16-Bit Hex – TI Style", and look at the password programmed into the device on first load.



MSP430 CPU

Set the properties of the selected cpu.

Bypass

initialization script

Slave Processor

Password: (HEX format) 0x34127856

JTAG/SBW Speed Medium

Figure 1. Target Configuration

NOTE: When a different user project is loaded, if JTAG lock with password was enabled in the code that was previously loaded in the device, the correct password must be provided in the project settings the first time to get into the part and erase the device. After the part is erased, if JTAG lock with password is not enabled in the new project, then the part is unlocked and can be debugged normally.

2.3.2 Using JTAG Lock With Password in IAR

The code example JTAG_lock_FR5xx_with_password.c from the associated zip file shows how to set the JTAG lock with password on an MSP430FR5969 device in IAR. Note however that some production programming tools may also offer the option to set the password from a GUI interface instead of having to hard-code it into the project.

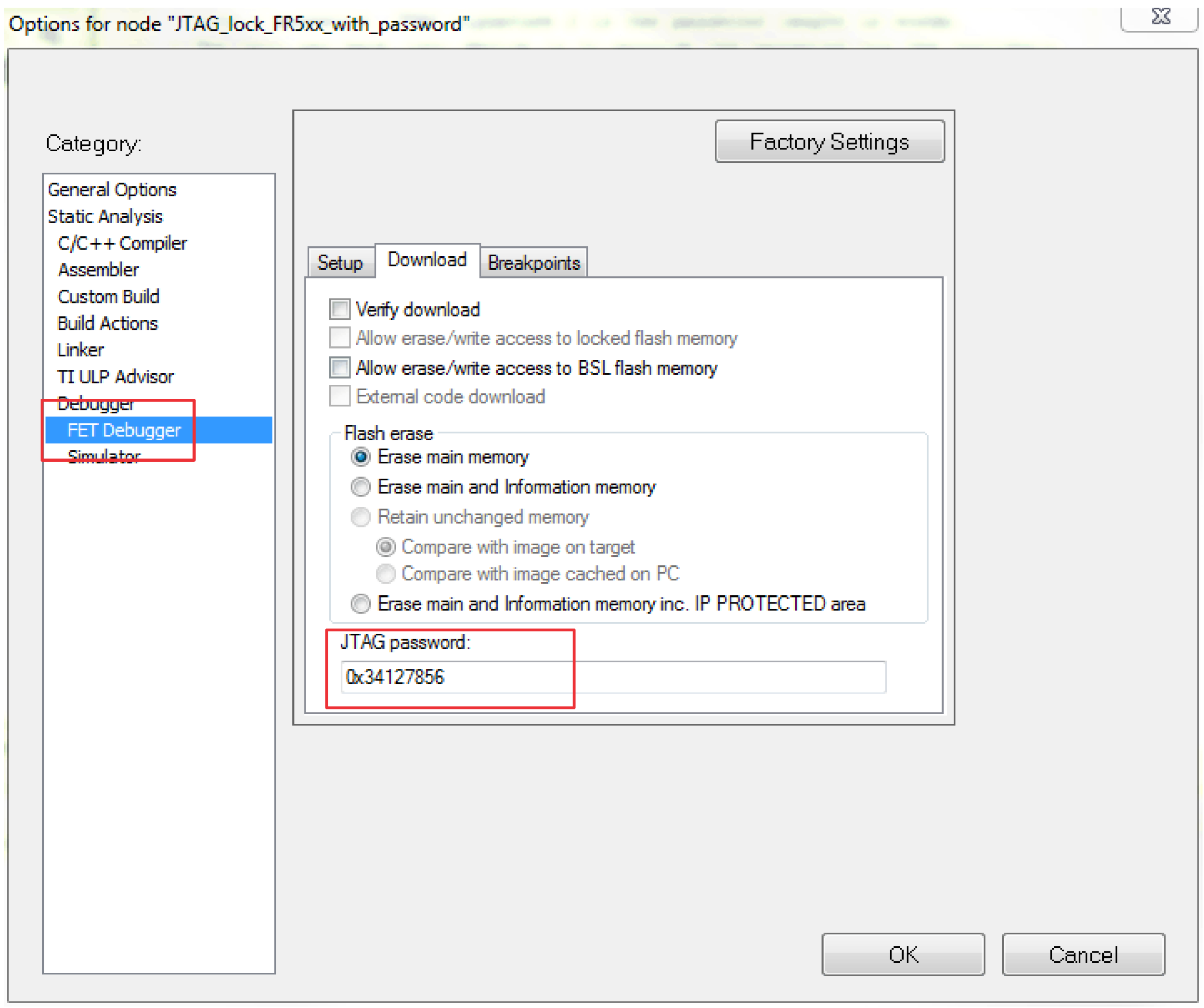
For the IAR C/C++ compiler, use #pragma location to place the correct signatures starting at the address FF80h, and to place the password starting at the address FF88h. Use the __root keyword to keep the compiler from optimizing out the data (since it is not used by the program). For more information on these pragmas, see the IAR C/C++ Compiler User's Guide, found in IAR under the Help menu.

```
#pragma location = 0xFF80
__root const uint16_t JTAG_signatures[] = {0xAAAA, 0x0002};

...

#pragma location = 0xFF88
__root const uint8_t JTAG_password[] = {0x12, 0x34, 0x56, 0x78};
```

The following example shows a section of the TI-txt binary file for a device locked with a 2-word password of 12h 34h 56h 78h:

```
@FF80
AA AA 02 00
@FF88
12 34 56 78
```

To provide the password in IAR, in the Project > Options menu, go to "Debugger" > "FET Debugger", and select the "Download" tab. In the box "JTAG password" enter the password to provide when unlocking the part. For the example above, the password is 12h 34h 56h 78h. The password should be written in words, starting with the least significant word, so for this example, enter 0x34127856. An easy way to find the correct order to supply the password to the toolchain is to use the Memory view with the format set to "2x Units", and look at the password programmed into the device on first load.



Options for node "JTAG_lock_FR5xx_with_password"

Category:
- General Options
- Static Analysis
- C/C++ Compiler
- Assembler
- Custom Build
- Build Actions
- Linker
- TI ULP Advisor
- Debugger
- FET Debugger
- Simulator

Download tab:
- [ ] Verify download
- [ ] Allow erase/write access to locked flash memory
- [ ] Allow erase/write access to BSL flash memory
- [ ] External code download

Flash erase:
- [✓ Erase main memory]
- Erase main and Information memory
- Retain unchanged memory
  - Compare with image on target
  - Compare with image cached on PC
- Erase main and Information memory inc. IP PROTECTED area

JTAG password: 0x34127856

Figure 2. JTAG Password on First Load

NOTE: When a different user project is loaded, if JTAG lock with password was enabled in the code that was previously loaded in the device, the correct password must be provided in the project settings the first time to get into the part and erase the device. After the part is erased, if JTAG lock with password is not enabled in the new project, then the part is unlocked and can be debugged normally.

3 IP Encapsulation (IPE)

IP Encapsulation (IPE) is a feature found on some MSP430 FRAM microcontrollers, like the MSP430FR59xx/69xx family. (Note: For the IP Protection secure zones feature found on MSP432 devices, see the application report *Software IP Protection on MSP432P4xx Microcontrollers* (SLAA660).

IP Encapsulation allows the user to encapsulate and protect an area of the FRAM memory from readout. When IPE is in place, any code or data in the IP Encapsulated area is protected from read or write access from anywhere outside of the IP Encapsulated area, even by JTAG. The way to remove the IPE protection is to perform a special mass erase sequence enabled by the tool chain. No form of code protection is perfect, but this feature adds an additional layer on top of JTAG/SBW or bootloader security, for sensitive data like keys or for proprietary code that is the user's intellectual property (IP).

The IP Encapsulation is only as secure as the code stored in it—poor code security practices can make the code more vulnerable even if it is within the encapsulated area. Code that can read/write to addresses, or that does not follow robust coding practices, is especially concerning. It is also important to clear any hardware modules and peripheral registers used or RAM to their original states at the end of the encapsulated execution if it is desired to hide what the code does with these modules.

For example, the IPE example project includes an example of clearing hardware module registers used by the IPE code section:

```
    TA0CCTL0 = 0;
    TA0CCR0 = 0;
    TA0CCR1 = 0;
    TA0CCR2 = 0;
    TA0CTL = 0;
    TA0R = 0;
    P4DIR &= ~BIT6;
    P4OUT &= ~BIT6;
```

It also includes an example of clearing the general-purpose CPU registers R4–R15:

```
    __asm(" mov.w #0, R4");
    __asm(" mov.w #0, R5");
    __asm(" mov.w #0, R6");
    __asm(" mov.w #0, R7");
    __asm(" mov.w #0, R8");
    __asm(" mov.w #0, R9");
    __asm(" mov.w #0, R10");
    __asm(" mov.w #0, R11");
    __asm(" mov.w #0, R12");
    __asm(" mov.w #0, R13");
    __asm(" mov.w #0, R14");
    __asm(" mov.w #0, R15");
```

NOTE: If passing parameters back to code outside the IPE area, the registers R12-R15 may need to be preserved. See the *MSP430 Optimizing C/C++ Compiler User's Guide* sections on "How a Function Makes a Call" and "How a Called Function Responds" for more information.

More code security measures may be needed for different applications; for example, clearing RAM that was allocated in the course of the function. Another possible precaution would be disabling interrupts (if possible) for functions not used by the IPE while the IPE executes, and reenabling them at the end of the IPE. This would ensure that the registers or RAM are cleared before servicing any ISR outside of the IPE region. These types of advanced precautions are very application-specific, with what is important to hide varying based on need, but in all cases care should be taken.

3.1 IP Encapsulation Using the IPE Tool in CCS

CCS includes a built-in IPE tool, as well as predefined segments in the linker file, to help enable IPE in a project. While the *MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family User's Guide* describes in detail how to set up an IPE initialization structure and manually enable IPE using the MPU registers, the use of the built-in IPE tool in CCS presents an easier and more automated alternative. The next sections discuss how to enable IPE in a project using these tools in CCS. Also see the associated source (http://www.ti.com/lit/zip/slaa685) for an example project in CCS that uses IPE. This code is meant to be run on the MSP-EXP430FR5969 LaunchPad Development Kit. For more information on enabling IPE in CCS using the tool, see the *Code Composer Studio v6.1 for MSP430 User's Guide* (SLAU157).

3.1.1 CCS Linker File Features for IPE

NOTE: These linker file features are current as of CCSv6.1.1. Earlier versions of the CCS linker file do not have all of these features, so it is recommended to use CCSv6.1.1 or later when using this method for enabling IPE.

The CCS linker .cmd files for devices with IPE support include some sections for placing IPE code and data. All that user code has to do is to tell the compiler what variables and functions should be placed within these predefined sections. The following snippet is from the MSP430FR5969 linker cmd file:

```
GROUP(IPENCAPSULATED_MEMORY)
{
   .ipestruct     : {}              /* IPE Data structure               */
   .ipe           : {}              /* IPE                              */
   .ipe_const     : {}              /* IPE Protected constants          */
   .ipe:_isr      : {}              /* IPE ISRs                         */
   .ipe_vars      : type = NOINIT{} /* IPE variables                    */
} PALIGN(0x0400), RUN_START(fram_ipe_start) RUN_END(fram_ipe_end)
RUN_END(fram_rx_start)
```

- **.ipestruct** – This segment is for containing the IPE initialization structure that includes information about the address location of the IP Encapsulated memory boundaries and control settings. There is more information about the IPE initialization structure in the *MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family User's Guide*. The user should not place any data in this section—when using the built-in CCS IPE tool, the tool generates the required values and places them here.
- **.ipe** – This segment is for any code functions the user wants to place in the IP Encapsulated area.
- **.ipe_const** – This segment is for any constant data that needs to be encapsulated. This could be things like encryption keys, or calibration info, encapsulated code version numbers, or any constants used by the encapsulated code that should be kept hidden. Application dependent.
- **.ipe: _isr** – This segment is for any Interrupt Service Routines (ISRs) that are needed as part of the IP encapsulated user code.
- **.ipe_vars** – This segment is for any data variables that need to be encapsulated. Any variables used solely within the IP encapsulated code should be also placed in the IP encapsulated area instead of in RAM for the most security – otherwise observation of changes to RAM could be used to try to reverse-engineer the functionality of the encapsulated code. This is of course application dependent.

Further at the bottom of the linker file, observe there is a section for calculating the IPE initialization values for the IPE structure:

```
/****************************************************************************/
/* MPU/IPE Specific memory segment definitions                              */
/****************************************************************************/
```

```
#ifdef _IPE_ENABLE
   #define IPE_MPUIPLOCK 0x0080
   #define IPE_MPUIPENA 0x0040
   #define IPE_MPUIPPUC 0x0020

   // Evaluate settings for the control setting of IP Encapsulation
   #if defined(_IPE_ASSERTPUC1)
        #if defined(_IPE_LOCK ) && (_IPE_ASSERTPUC1 == 0x08))
         fram_ipe_enable_value = (IPE_MPUIPENA | IPE_MPUIPPUC |IPE_MPUIPLOCK);
        #elif defined(_IPE_LOCK )
         fram_ipe_enable_value = (IPE_MPUIPENA | IPE_MPUIPLOCK);
      #elif (_IPE_ASSERTPUC1 == 0x08)
         fram_ipe_enable_value = (IPE_MPUIPENA | IPE_MPUIPPUC);
      #else
         fram_ipe_enable_value = (IPE_MPUIPENA);
      #endif
   #else
      #if defined(_IPE_LOCK )
         fram_ipe_enable_value = (IPE_MPUIPENA | IPE_MPUIPLOCK);
      #else
         fram_ipe_enable_value = (IPE_MPUIPENA);
      #endif
   #endif

   // Segment definitions
   #ifdef _IPE_MANUAL                  // For custom sizes selected in the GUI
      fram_ipe_border1 = (_IPE_SEGB1>>4);
      fram_ipe_border2 = (_IPE_SEGB2>>4);
   #else                           // Automated sizes generated by the Linker
      fram_ipe_border2 = fram_ipe_end >> 4;
      fram_ipe_border1 = fram_ipe_start >> 4;
   #endif

   fram_ipe_settings_struct_address = Ipe_settingsStruct >> 4;
   fram_ipe_checksum = ~((fram_ipe_enable_value & fram_ipe_border2 &
fram_ipe_border1) | (fram_ipe_enable_value & ~fram_ipe_border2 & ~fram_ipe_border1)
| (~fram_ipe_enable_value & fram_ipe_border2 & ~fram_ipe_border1) |
(~fram_ipe_enable_value & ~fram_ipe_border2 & fram_ipe_border1));
#endif
```

The user does not have to modify anything with these settings, but this demonstrates how the linker file is working with the CCS IPE tool settings, to determine the values to go in the IPE initialization structure.

3.1.2 Placing Code and Data in the IPE Sections in CCS

The user must indicate for the compiler which code and data should be placed in the IPE sections defined in the linker file. For the MSP430 TI C/C++ compiler, this is enabled through the use of two pragmas, CODE_SECTION and DATA_SECTION. For more information on these pragmas, see the *MSP430 Optimizing C/C++ Compiler User's Guide* (SLAU132). For more information on enabling IPE in CCS using the tool, see the *Code Composer Studio v6.1 for MSP430 User's Guide* (SLAU157).

Code that should be placed in the IPE area must be contained in a function or set of functions. These functions should all be placed in the .ipe section (see the following example).

```
#pragma CODE_SECTION(IPE_encapsulatedInit, ".ipe")
void IPE_encapsulatedInit (void)
{
        ...
        ...

}

#pragma CODE_SECTION(IPE_encapsulatedBlink, ".ipe")
void IPE_encapsulatedBlink (void)
{
        ...
        ...
}
```

Any interrupt service routines (ISRs) used by the IP encapsulated code should also be placed in the IPE area so that they are also encapsulated. These ISRs should be placed in the .ipe:_isr section (see the following example).

```
#pragma CODE_SECTION(TIMER0_A0_ISR, ".ipe:_isr")
#pragma vector=TIMER0_A0_VECTOR
__interrupt
void TIMER0_A0_ISR(void)
{
        ...
        ...

}
```

Some applications may have constants like encryption keys that are recommended to be stored in the IPE area. The CCS linker file does not allow constants in the same section as variables or code. These constant values should be placed in the .ipe_const section (see the following example).

```
#pragma DATA_SECTION(IPE_encapsulatedKeys, ".ipe_const")
const uint16_t IPE_encapsulatedKeys[] = {0x0123, 0x4567, 0x89AB, 0xCDEF,
                                         0xAAAA, 0xBBBB, 0xCCCC, 0xDDDD};
```

Finally, any variables used by the IPE code, should also be placed in the IPE area rather than RAM so that other parts of the code cannot read or access them.

```
#pragma DATA_SECTION(IPE_encapsulatedCount, ".ipe_vars")
unsigned char IPE_encapsulatedCount;
```

```
...

#pragma DATA_SECTION(IPE_i, ".ipe_vars")
uint8_t IPE_i;
```

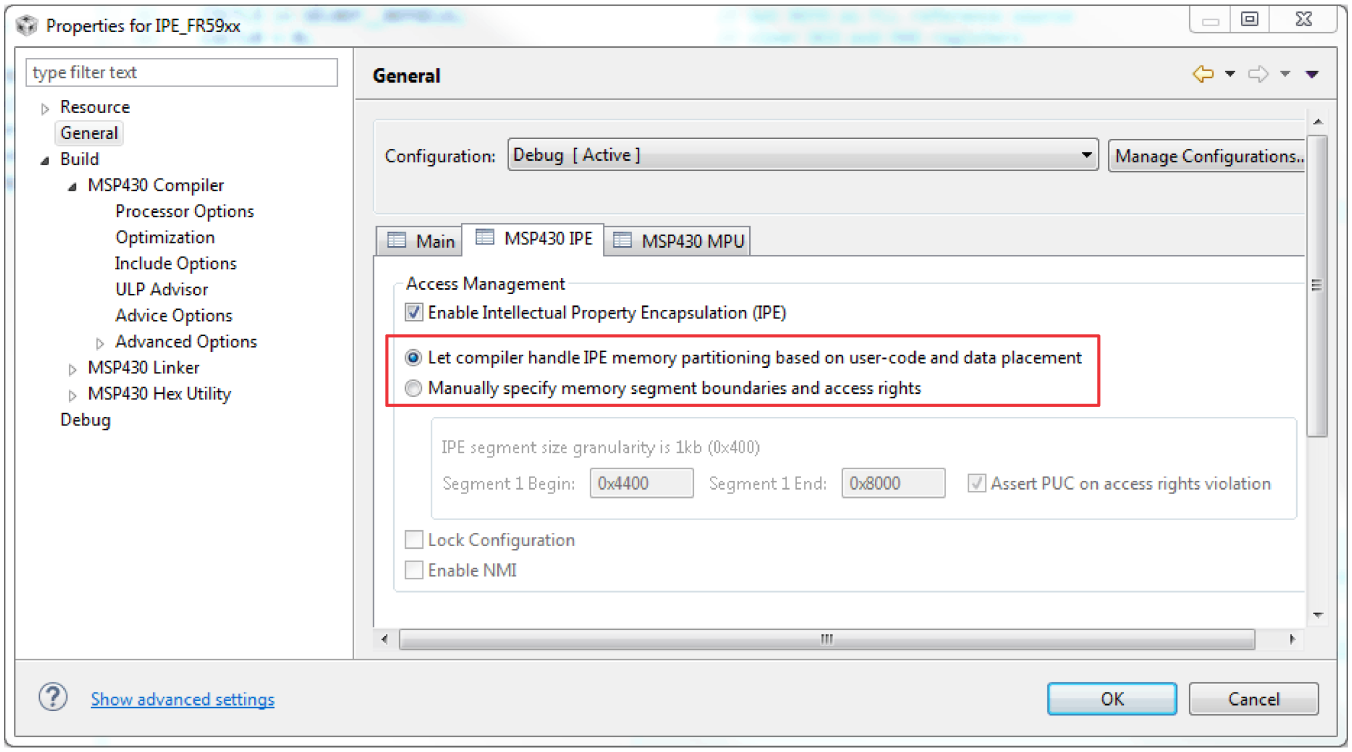
The .ipe_vars section is set as NOINIT in the linker file to indicate that any variables placed here should not have any initialization value and should not be zero-initialized by the start-up code. This is because the C start-up code and .cinit initialization tables are not located in the IPE area, and therefore cannot write to and set the variables in the IPE section because they do not have permission. For the user, this means that any variables placed in the .ipe_vars section, should be set in a custom init function that is placed in the .ipe section, so that the variables are initialized by code running inside the IPE area. This init function should then be called by the user's main code at start-up to initialize the variables before they are used, for example after halting the watchdog timer.

```
#pragma DATA_SECTION(IPE_encapsulatedCount, ".ipe_vars")
unsigned char IPE_encapsulatedCount;

/*
 * main.c
 */
void main(void) {
    WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer

    IPE_encapsulatedInit();
        ...
        ...
}

#pragma CODE_SECTION(IPE_encapsulatedInit, ".ipe")
void IPE_encapsulatedInit (void)
{
        IPE_encapsulatedCount = 1;
}
```

3.1.3 Enabling IPE in CCS Project Options

Enabling IPE in the CCS Project is simply achieved by going to Project > Properties > General, and then using the MSP430 IPE tab.

For automated IPE handling, select "Enable Intellectual Property Encapsulation (IPE)" and "Let compiler handle IPE memory partitioning based on user-code and data placement". Click OK, and now the IPE is set up (see Figure 3).


Properties for IPE_FR59xx

General

Configuration: Debug [Active]

Access Management

[✓] Enable Intellectual Property Encapsulation (IPE)

(●) Let compiler handle IPE memory partitioning based on user-code and data placement

( ) Manually specify memory segment boundaries and access rights

IPE segment size granularity is 1kb (0x400)

Segment 1 Begin: 0x4400 Segment 1 End: 0x8000 [✓] Assert PUC on access rights violation

[ ] Lock Configuration

[ ] Enable NMI

Figure 3. IPE Tool in CCS

3.1.4 Viewing the Compiler-Selected IPE partitioning in CCS

After building the project in CCS with IPE enabled, if desired, users can use the .map file found in the Debug folder to view what IPE boundaries were set and where the different segments like .ipe and the functions within have been placed. Additionally, the values of things like fram_ipe_border1, fram_ipe_settings_struct_address, and others can also be found.

```
SECTION ALLOCATION MAP

 output                                  attributes/
section   page    origin      length       input sections
--------  ----  ----------  ----------   ----------------
...
...

.ipestruct
*           0    00004800    00000008
                  00004800    00000008
MSPIPE_INIT_LIB_CCS_msp430_large_code_restricted_data.lib : ipe_init.o
(.ipestruct:retain)

.ipe        0    00004808    000000d2
                  00004808    000000cc     IPE_FR59xx.obj
(.ipe:IPE_encapsulatedBlink)
                  000048d4    00000006     IPE_FR59xx.obj (.ipe:IPE_encapsulatedInit)

.ipe_const
*           0    000048da    00000010
                  000048da    00000010     IPE_FR59xx.obj (.ipe_const)
```

```
.ipe:_isr
*          0    000048ea    0000000a
                  000048ea    0000000a     IPE_FR59xx.obj (.ipe:_isr:TIMER0_A0_ISR)

.ipe_vars
*          0    000048f4    00000002     UNINITIALIZED
                  000048f4    00000002     IPE_FR59xx.obj (.ipe_vars)...

GLOBAL SYMBOLS: SORTED ALPHABETICALLY BY Name

address   name
-------   ----
...
...
00004808  IPE_encapsulatedBlink
000048f4  IPE_encapsulatedCount
000048d4  IPE_encapsulatedInit
000048da  IPE_encapsulatedKeys
000048f5  IPE_i
0000ff88  Ipe_enableSignature
00004800  Ipe_settingsStruct
0000ff8a  Ipe_structureAddress
...
...
00000480  fram_ipe_border1
000004c0  fram_ipe_border2
ffffffff  fram_ipe_checksum
00000040  fram_ipe_enable_value
00004c00  fram_ipe_end
00000480  fram_ipe_settings_struct_address
00004800  fram_ipe_start
00004400  fram_rw_start
00004c00  fram_rx_start
...
...
```

The IPE struct is built into the binary image for the code (.txt or .hex) to allow it to be found by the device boot code at first start-up as described in the *MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family User's Guide* (SLAU367). This binary can also be viewed by going to Project > Properties > Build > MSP430 Hex Utility. Select "Enable MSP430 Hex Utility", and under "Output Format Options" choose "Output TI-TXT hex format (--ti_txt)". Build, and a .txt file should appear in the Debug folder.

When viewing the txt file to view the IPE initialization struct, verify these items:

1. Check the value at 0xFF8A. This is the portion of the IPE signature that contains the address of the IPE struct. Remember that the value has been shifted right by 4 bits. In the example, the value at 0xFF8A is 0x0480 which points to address 0x4800 for the ipe struct. Note how this matches the .map file.

   ```
   @ff80
   FF FF FF FF FF FF FF FF AA AA 80 04 FF FF FF FF
   ```

2. Go to the address of the IPE initialization struct (which we found in step 1, in this case 0x4800), and see the values in the structure. It will be in the format from the user's guide.

Table 7. IPE Initialization Structure

Field Name	Address Offset	Length	Description
MPUIPC0	0h	word	Control setting for IP Encapsulation. Value is written to MPUIPC0
MPUIPB2	2h	word	Upper border of IP Encapsulation segment. Value is written to MPUIPSEGB2.
MPUIPB1	4h	word	Lower border of IP Encapsulation segment. Value is written to MPUIPSEGB1.
MPUCHECK	6h	word	Odd bit interleaved parity

In the following example, you can see:

- MPUIPC0 = 0x0040 → MPUIPENA = 1 = IPE enable
- MPUIPB2 = 0x04C0 → address 0x4C00 is IPE end
- MPUIPB1 = 0x0480 → address 0x4800 is IPE start
- MPUCHECK = 0xFFFF → checksum of the previous data. Calculated where checksum lower byte = INV(Byte 0 XOR Byte 2 XOR Byte 4) = INV(40 XOR C0 XOR 80) = 0xFF, and checksum upper byte = INV(Byte 1 XOR Byte 3 XOR Byte 5) = INV(00 XOR 04 XOR 04) = 0xFF

```
@4800
40 00 C0 04 80 04 FF FF
```

3.1.5 Running and Testing IPE Code in CCS

This code is meant to be run on the MSP-EXP430FR5969 LaunchPad Development Kit.

1. Download the example (http://www.ti.com/lit/zip/slaa685)
2. Import the project into CCSv6 or greater:
   - Project > Import CCS Projects…
   - Browse… and select the location where you have downloaded and unzipped the code. Make sure that "Copy projects into workspace" is checked, and click OK.
3. Build the project.

3.1.5.1 Debug Settings With IPE in CCS

As mentioned in the *MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family User's Guide* (SLAU367), the IPE settings do not take effect until the part has reset so that the boot code can run. To test the IPE preventing JTAG access of protected areas, two different debug configurations have been provided with the example project to simplify the processes of loading the part and testing the IPE feature:

- IPE_FR59xx_load – Programs the device with new code, including IPE code.
- IPE_FR59xx_test – Debugs the device without reprogramming the device. Used for IPE protection testing.

In CCS, select the drop-down arrow next to the debug bug icon, and select Debug configurations (see Figure 4).

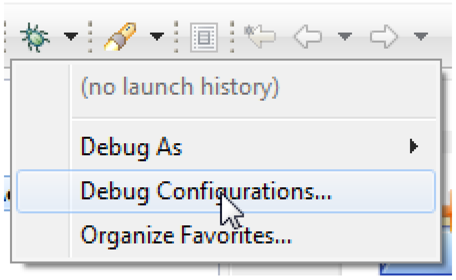
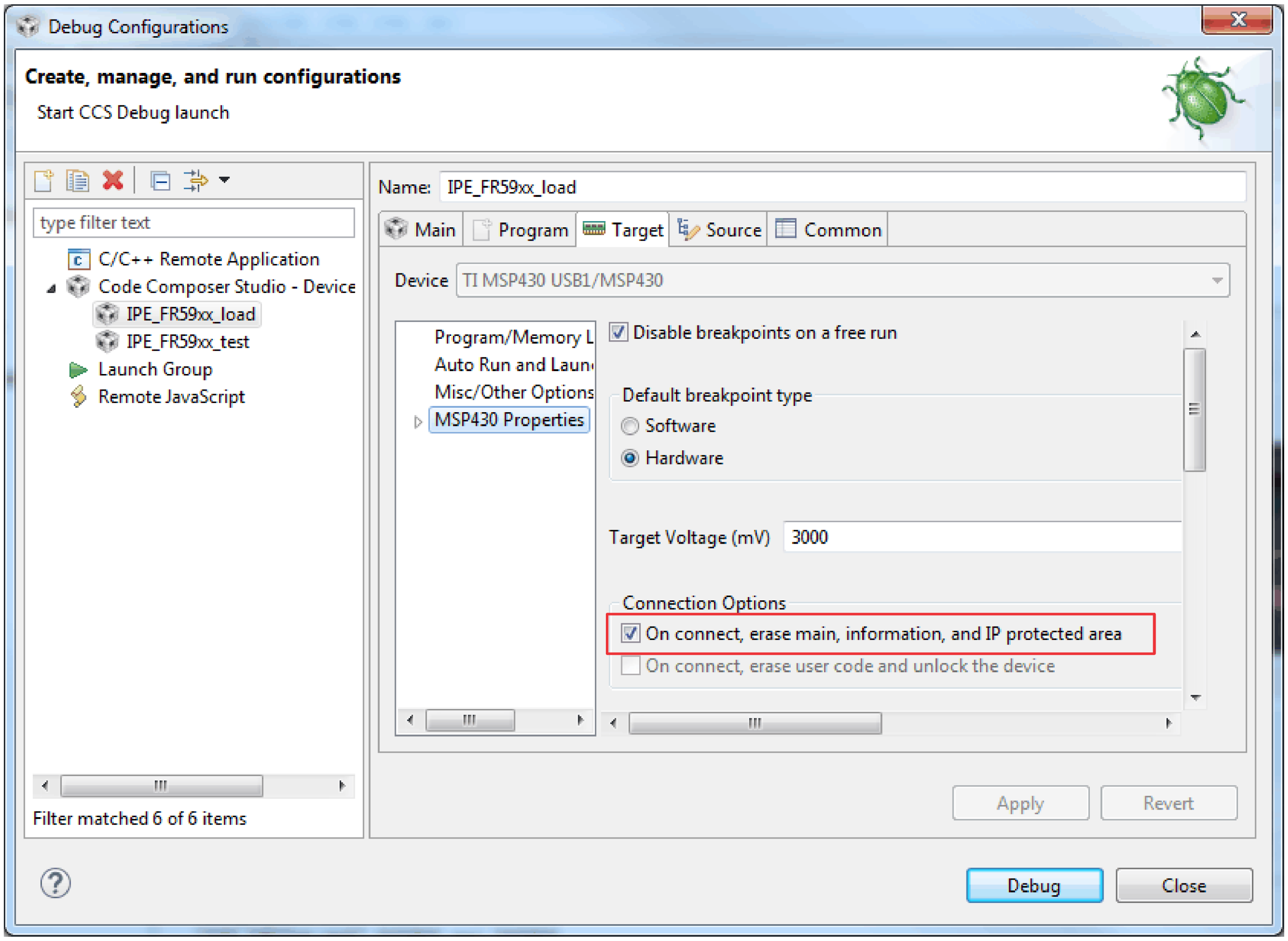


Figure 4. Debug Configurations in CCS

In the new window (see Figure 5), there are two debug configurations for this project, IPE_FR59xx_load and IPE_FR59xx_test. Each of these configurations includes some changes from the default debug configuration to better facilitate debug with IPE enabled.

IPE_FR59xx_load configures the device to erase the IP protected area before trying to program the part. Otherwise, the program load fails if IPE code is present, because IPE prevents programming access from JTAG. However a special erase can be performed by the toolchain to mass erase the part and also erase the secured nonvolatile system data area that stores the saved IPE structure pointer [see "Trapdoor Mechanism for IP Structure Pointer Transfer" in the *MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family User's Guide* (SLAU367)]. This is set in the debug configuration on the Target tab under MSP430 Properties > Connection Options, by selecting "On connect, erase main, information, and IP protected area".


Debug Configurations

Create, manage, and run configurations

Start CCS Debug launch

Name: IPE_FR59xx_load

Device: TI MSP430 USB1/MSP430

- C/C++ Remote Application
- Code Composer Studio - Device
  - IPE_FR59xx_load
  - IPE_FR59xx_test
- Launch Group
- Remote JavaScript

Filter matched 6 of 6 items

Tabs: Main, Program, Target, Source, Common

Program/Memory L
Auto Run and Laun
Misc/Other Options
MSP430 Properties

☑ Disable breakpoints on a free run

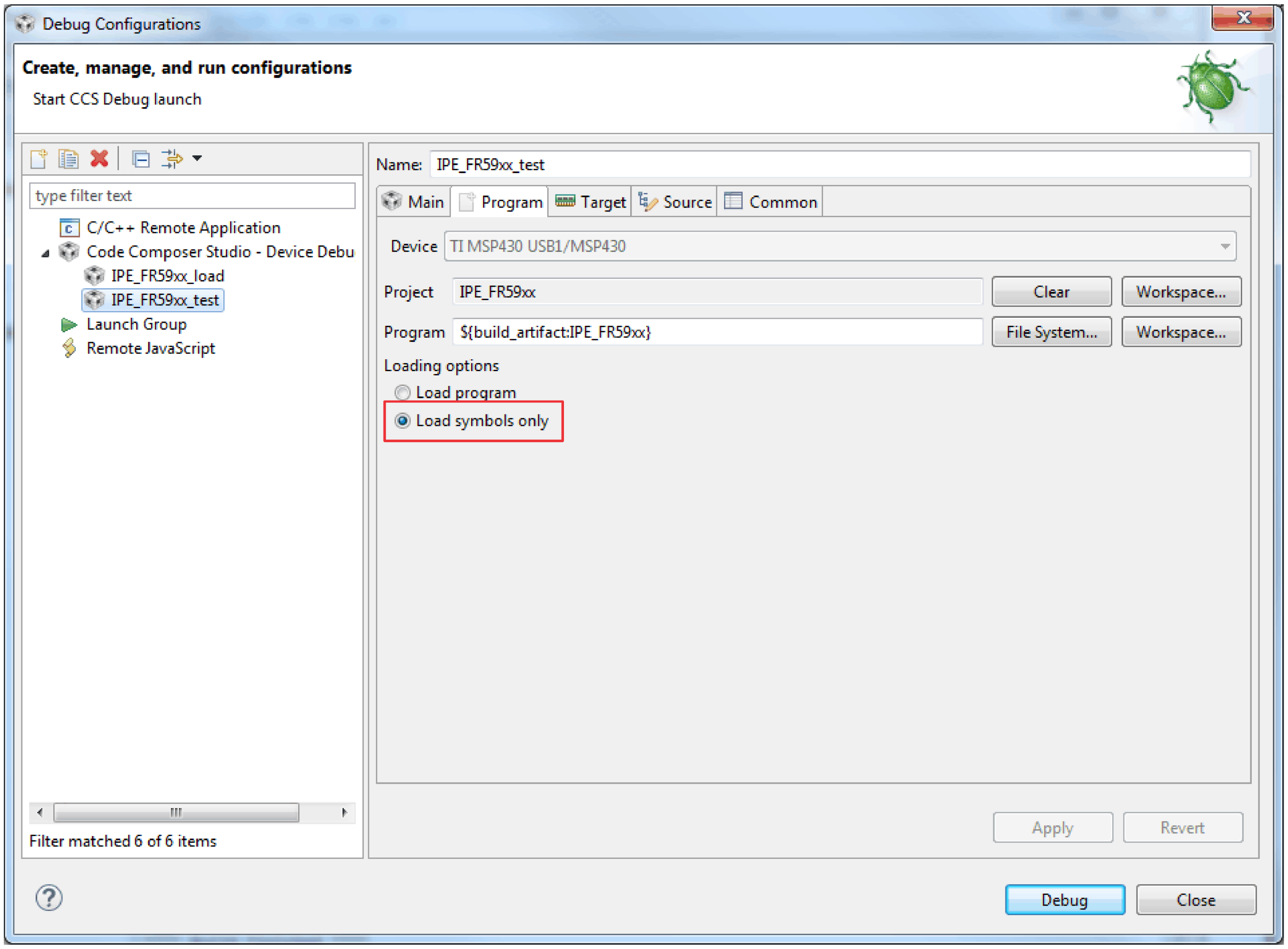
Default breakpoint type
○ Software
● Hardware

Target Voltage (mV) 3000

Connection Options
☑ On connect, erase main, information, and IP protected area
☐ On connect, erase user code and unlock the device

Figure 5. IPE Load Configuration in CCS

IPE_FR59xx_test on the other hand, makes sure that this erase-on-connect does not occur, so that the user can observe the IPE behavior (see Figure 6). This build configuration leaves the On connect, erase main, information, and IP protected area setting deselected. Rather, the difference between IPE_FR59xx_test and the default configuration is that IPE_FR59xx_test does not try to load the program (which would fail with IPE enabled). The user in this case needs to debug without downloading. This is set in the debug configuration on the Program tab, under Loading options, by selecting "Load symbols only".


Debug Configurations

Create, manage, and run configurations

Start CCS Debug launch

type filter text

- C/C++ Remote Application
- Code Composer Studio - Device Debu
  - IPE_FR59xx_load
  - IPE_FR59xx_test
- Launch Group
- Remote JavaScript

Filter matched 6 of 6 items

Name: IPE_FR59xx_test

Main | Program | Target | Source | Common

Device: TI MSP430 USB1/MSP430

Project: IPE_FR59xx

Program: ${build_artifact:IPE_FR59xx}

Loading options

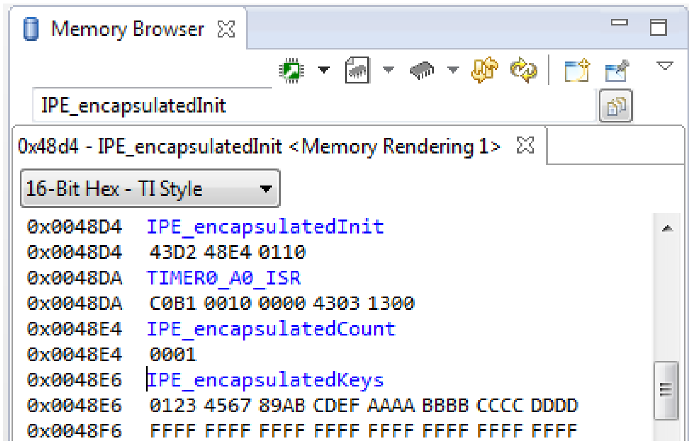
Load program [✓ Load symbols only]

Figure 6. IPE Test Configuration in CCS

For ease of development with IPE for any CCS project where IPE is enabled, TI recommends making two build configurations with the settings demonstrated here.

3.1.5.2 Testing IPE in CCS

1. Click the drop-down next to the debug icon, go to Debug Configurations, and select the IPE_FR59xx_load debug configuration. Debug.
2. The program loads into the device. Run the code. The LEDs on the LaunchPad should blink.
3. Pause the code.
   1. Select View > Memory Browser to open the Memory view. In the Memory view, enter the name of IPE protected variables or functions (IPE_encapsulatedInit, IPE_encapsulatedBlink, and so on).
   2. Observe that the IP encapsulated variables and functions are still viewable. This is because the boot code must run and load the IPE structure pointer into the internal secured nonvolatile system data area the first time, and then load the IPE registers from the IPE structure before IPE will take effect (requires a reset so bootcode run).


Memory Browser

IPE_encapsulatedInit

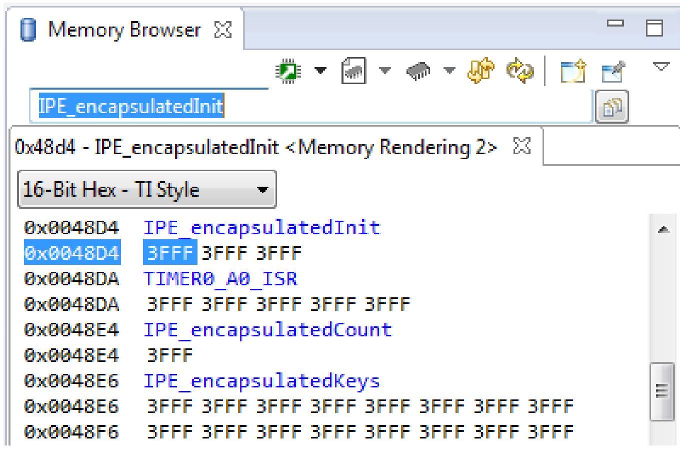
0x48d4 - IPE_encapsulatedInit <Memory Rendering 1>

16-Bit Hex - TI Style

```
0x0048D4  IPE_encapsulatedInit
0x0048D4  43D2 48E4 0110
0x0048DA  TIMER0_A0_ISR
0x0048DA  C0B1 0010 0000 4303 1300
0x0048E4  IPE_encapsulatedCount
0x0048E4  0001
0x0048E6  IPE_encapsulatedKeys
0x0048E6  0123 4567 89AB CDEF AAAA BBBB CCCC DDDD
0x0048F6  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
```

Figure 7. IPE Memory View in CCS - Unprotected

4. End the debug session.
5. Power cycle the device. The IPE should now be active and preventing readout and write access to the IPE areas of the device.
6. Click the drop-down next to the debug icon, go to Debug Configurations, and select the IPE_FR59xx_test debug configuration. Debug.
7. No program is loaded this time, only the debugging symbols. Run the code. LEDs on the LaunchPad should blink.
8. Pause the code.
   - Select View > Memory Browser to open the Memory view. In the Memory view, enter the name of IPE protected variables or functions (IPE_encapsulatedInit, IPE_encapsulatedBlink, and so on).
   - Observe that the IPE encapsulated variables and functions are no longer viewable. 0x3FFF (JMP$) is all that the tool can see in these areas. IP encapsulation is active and preventing readout of the memory area.


Memory Browser

IPE_encapsulatedInit

0x48d4 - IPE_encapsulatedInit <Memory Rendering 2>

16-Bit Hex - TI Style

```
0x0048D4  IPE_encapsulatedInit
0x0048D4  3FFF 3FFF 3FFF
0x0048DA  TIMER0_A0_ISR
0x0048DA  3FFF 3FFF 3FFF 3FFF 3FFF
0x0048E4  IPE_encapsulatedCount
0x0048E4  3FFF
0x0048E6  IPE_encapsulatedKeys
0x0048E6  3FFF 3FFF 3FFF 3FFF 3FFF 3FFF 3FFF 3FFF
0x0048F6  3FFF 3FFF 3FFF 3FFF 3FFF 3FFF 3FFF 3FFF
```

Figure 8. IPE Memory View in CCS - Protected

NOTE: When a different user project is loaded, if IPE was enabled in the code that was previously loaded in the device, the IPE area will cause the new project not to be able to load correctly and give an error in CCS (because the IPE area is protected from erase or write). To load the new project the first time, select "On connect, erase main, information, and IP protected area" in the debug settings for the new project for the first time you load it, to erase the IPE code and settings.

3.2 IP Encapsulation Using the IPE Tool in IAR

IAR includes a built-in IPE tool, as well as predefined segments in the linker file, to help enable IPE in a project. While the MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family User's Guide describes in detail how to set up an IPE initialization structure and manually enable IPE using the MPU registers, the use of the built-in IPE tool in IAR presents an easier more automated alternative. The next sections discuss how to enable IPE in a project using these tools in IAR. Also see the zip file (http://www.ti.com/lit/zip/slaa685) for an example project in IAR that uses IPE. This code is meant to be run on the MSP-EXP430FR5969 LaunchPad Development Kit. For more information on enabling IPE in IAR using the tool, see the IAR C/C++ Compiler User Guide found in the Help menu of the IAR Embedded Workbench™ IDE.

3.2.1 IAR Linker File Features for IPE

NOTE: These linker file features are current as of IAR v6.40.1. Earlier versions of the IAR linker file do not have all of these features, so it is recommended to use IAR v6.40.1 or greater when using this method for enabling IPE.

The IAR linker .xcl files for devices with IPE support include some sections for placing IPE code and data. All that user code has to do is to tell the compiler what variables and functions should be placed within these predefined sections. The following snippet is from the MSP430FR5969 linker xcl file:

```
// ---------------------------
// Intellectual Property Encapsulation (IPE)
//

-Z(CONST)IPE_B1=4400-FF7F
-Z(DATA)IPEDATA16_N
-Z(CODE)IPECODE16
-Z(CONST)IPEDATA16_C,IPE_B2
```

- **(CONST)IPE_B1** – This segment is the beginning of the IPE area, and also contains the IPE initialization structure that includes information about the address location of the IP Encapsulated memory boundaries and control settings. There is more information about the IPE initialization structure in the *MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family User's Guide* (SLAU367). The user should not place any data in this section. When using the built-in IAR IPE tool, the tool generates the required values and places them here.
- **(DATA)IPEDATA16_N** – This segment is for any data variables that need to be encapsulated (these variables should be declared with the __no_init option). Any variables used solely within the IP encapsulated code should be also placed in the IP encapsulated area instead of in RAM for the most security. Otherwise, observation of changes to RAM could be used to try to reverse-engineer the functionality of the encapsulated code. This is of course application dependent.
- **(CODE)IPECODE16** – This segment is for any code functions the user wants to place in the IP Encapsulated area. Any Interrupt Service Routines (ISRs) that are needed as part of the IP encapsulated user code are placed here as well.
- **(CONST)IPEDATA16_C** – This segment is for any constant data that needs to be encapsulated. This could include things like encryption keys or passwords, or other sensitive data.
- **(CONST)IPE_B2** – This is the boundary marking the end of the IPE area.

3.2.2 Placing Code and Data in the IPE Sections in IAR

The user must indicate for the compiler which code and data should be placed in the IPE sections defined in the linker file. For the IAR C/C++ compiler, this is enabled through the use of #pragma location. For more information on this pragma and also on enabling IPE in IAR, see the IAR C/C++ Compiler User Guide, found under the Help menu in IAR.

Code that should be placed in the IPE area must be contained in a function or set of functions. These functions should all be placed in the IPECODE16 section (see the following example).

```
#pragma location = "IPECODE16"
void IPE_encapsulatedInit(void)
{
        ...
        ...

}

#pragma location = "IPECODE16"
void IPE_encapsulatedBlink (void)
{
        ...
        ...
}
```

Any interrupt service routines (ISRs) used by the IP encapsulated code should also be placed in the IPE area so that they are also encapsulated. These ISRs should also be placed in the IPECODE16 section (see the following example).

```
#pragma location = "IPECODE16"
#pragma vector=TIMER0_A0_VECTOR
__interrupt
void TIMER0_A0_ISR(void)
{
        ...
        ...
}
```

Some applications may have constants like encryption keys that are recommended to be stored in the IPE area. The IAR linker file does not allow constants in the same section as variables or code. These constant values should be placed in the IPEDATA16_C section (see the following example).

```
#pragma location = "IPEDATA16_C"
const uint16_t IPE_encapsulatedKeys[] = {0x0123, 0x4567, 0x89AB, 0xCDEF,
                                         0xAAAA, 0xBBBB, 0xCCCC, 0xDDDD};
```

Finally, any variables used by the IPE code, should also be placed in the IPEDATA16_N section rather than RAM so that other parts of the code cannot read or access them. Note that these variables should be marked with the __no_init keyword.

```
#pragma location = "IPEDATA16_N"
__no_init unsigned char IPE_encapsulatedCount;
```

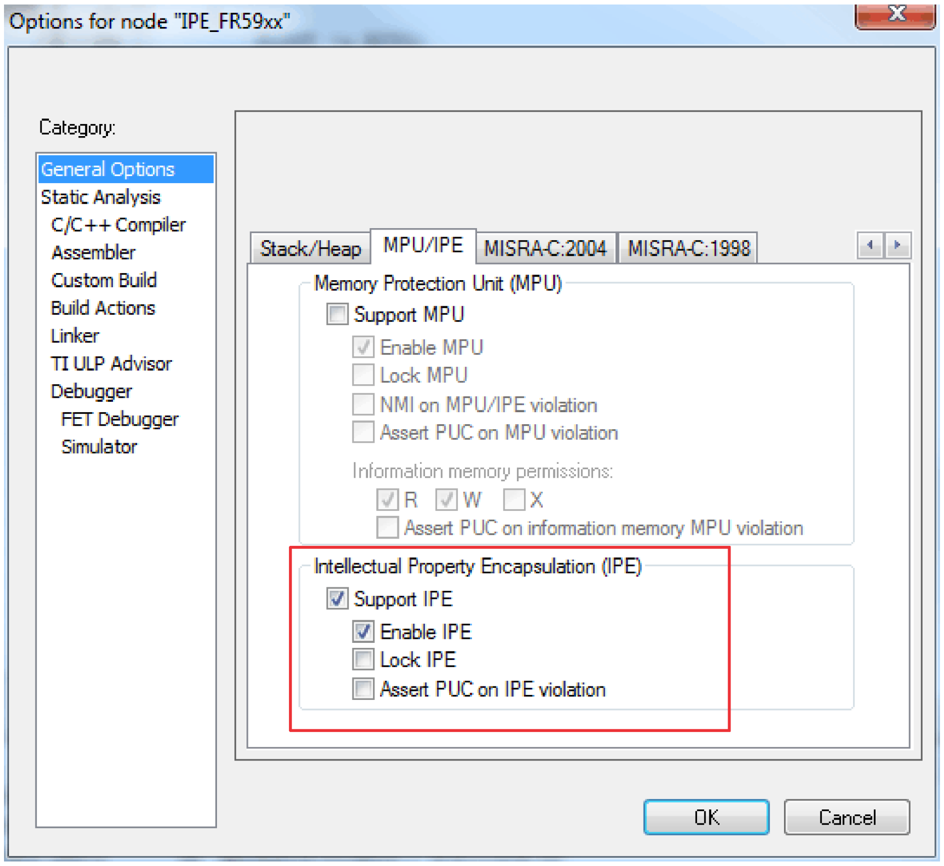
The variables should be marked with the __no_init keyword to indicate that any variables placed here should not have any initialization value and should not be zero-initialized by the start-up code. This is because the C start-up code and .cinit initialization tables are not located in the IPE area, and therefore cannot write to and set the variables in the IPE section because they do not have permission. For the user, this means that any variables placed in the IPEDATA16_N section, should be set in a custom init function that is placed in the IPECODE16 section, so that the variables are initialized by code running inside the IPE area. This init function should then be called by the user's main code at start-up to initialize the variables before they are used; for example, after halting the watchdog timer.

```
#pragma location = "IPEDATA16_N"
__no_init unsigned char IPE_encapsulatedCount;

/*
 * main.c
 */
void main(void) {
    WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer

    IPE_encapsulatedInit();
        ...
        ...
}

#pragma location = "IPECODE16"
void IPE_encapsulatedInit(void)
{
        IPE_encapsulatedCount = 1;
}
```

3.2.3 Enabling IPE in IAR Project Options

Enabling IPE in the IAR Project is simply achieved by going to Project > Options > General Options, and then using the MPU/IPE tab.

For automated IPE handling, select "Support IPE" and "Enable IPE". Click OK, and now the IPE is set up.



Memory Protection Unit (MPU)
- [ ] Support MPU
  - [✓] Enable MPU
  - [ ] Lock MPU
  - [ ] NMI on MPU/IPE violation
  - [ ] Assert PUC on MPU violation

Information memory permissions:
- [✓] R [✓] W [ ] X
- [ ] Assert PUC on information memory MPU violation

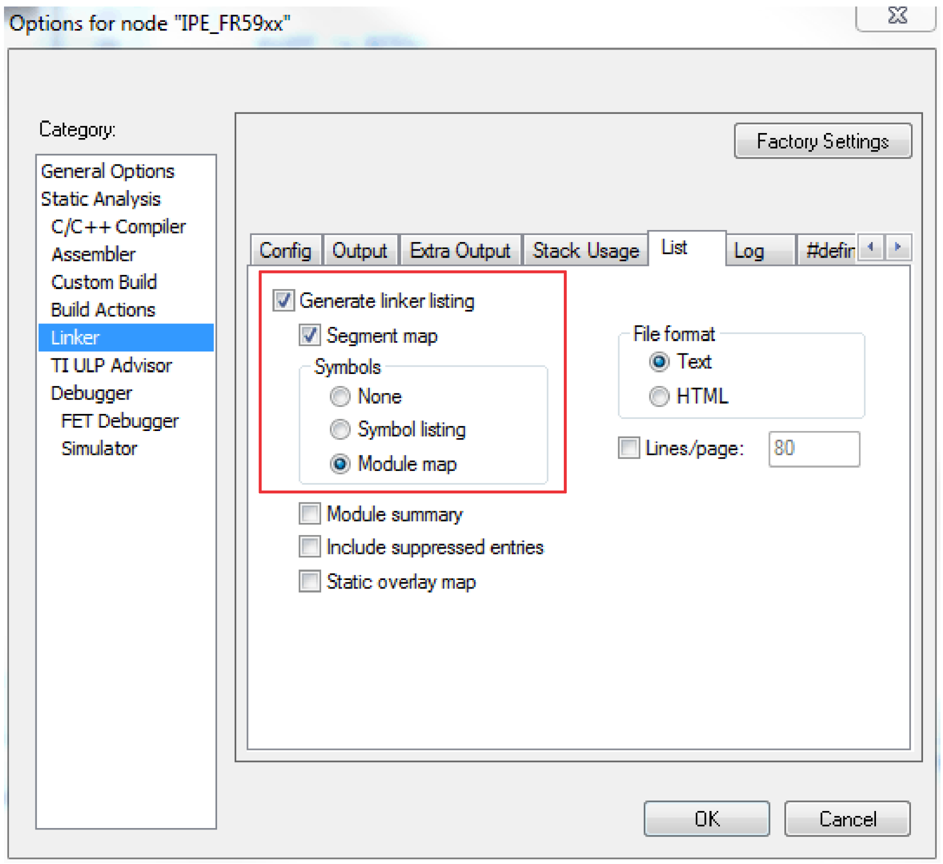
Intellectual Property Encapsulation (IPE)
- [✓] Support IPE
  - [✓] Enable IPE
  - [ ] Lock IPE
  - [ ] Assert PUC on IPE violation

Figure 9. IPE Tool in IAR

3.2.4 Viewing the Compiler-Selected IPE Partitioning in IAR

After building the project in IAR with IPE enabled, if desired, users can use the .map file found in the Output folder to view what IPE boundaries were set and where the different segments like IPECODE16 and the functions within have been placed. Additionally, the values of things like __iar_430_MPUIPC0_value, which shows the IPE register setup value, IPE_B1 and IPEB2 boundaries, and so on can also be found.

To generate the .map file, go to Project > Options > Linker, and on the List tab select "Generate linker listing". "Segment map" should already be checked as well. Build, and a .map file should appear in the Output folder.

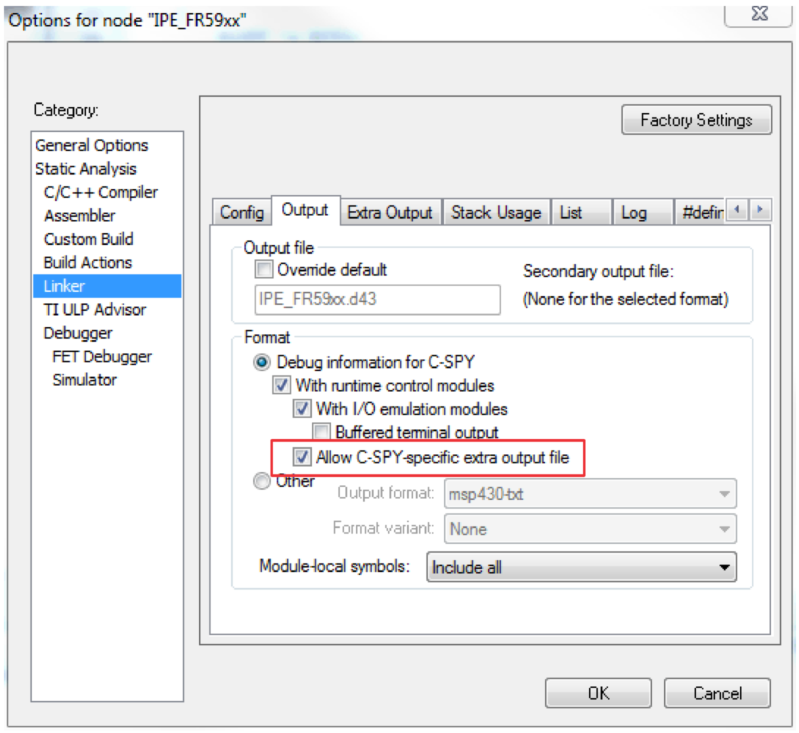


[✓] Generate linker listing
[✓] Segment map
Symbols: None / Symbol listing / [✓ Module map]
[ ] Module summary
[ ] Include suppressed entries
[ ] Static overlay map
File format: [✓ Text] / HTML
[ ] Lines/page: 80

Figure 10. Generating .map File in IAR

```
                 ****************************************
                 *                                      *
                 *              MODULE MAP              *
                 *                                      *
                 ****************************************
...
...

    -------------------------------------------------------------------------
IPEDATA16_N
  Relative segment, address: 4808 - 4808 (0x1 bytes), align: 0
  Segment part 17.            Intra module refs:   IPE_encapsulatedBlink
                                                   IPE_encapsulatedInit
           ENTRY                    ADDRESS         REF BY
           =====                    =======         ======
           IPE_encapsulatedCount    4808
    -------------------------------------------------------------------------
IPEDATA16_N
  Relative segment, address: 4809 - 4809 (0x1 bytes), align: 0
  Segment part 18.            Intra module refs:   IPE_encapsulatedBlink
           ENTRY                    ADDRESS         REF BY
           =====                    =======         ======
           IPE_i                    4809
    -------------------------------------------------------------------------
IPEDATA16_C
  Relative segment, address: 48DE - 48ED (0x10 bytes), align: 1
  Segment part 19.            Intra module refs:   IPE_encapsulatedBlink
           ENTRY                   ADDRESS         REF BY
           =====                   =======         ======
           IPE_encapsulatedKeys    48DE
    -------------------------------------------------------------------------...
...
...
    -------------------------------------------------------------------------
IPECODE16
  Relative segment, address: 480A - 480F (0x6 bytes), align: 1
  Segment part 25.            Intra module refs:   main
           ENTRY                   ADDRESS         REF BY
           =====                   =======         ======
           IPE_encapsulatedInit    480A
    -------------------------------------------------------------------------
IPECODE16
  Relative segment, address: 4810 - 48D3 (0xc4 bytes), align: 1
  Segment part 24.            Intra module refs:   main
           ENTRY                   ADDRESS         REF BY
           =====                   =======         ======
           IPE_encapsulatedBlink   4810
    -------------------------------------------------------------------------
IPECODE16
  Relative segment, address: 48D4 - 48DD (0xa bytes), align: 1
  Segment part 23.            Intra module refs:   TIMER0_A0_ISR::??INTVEC 90
           ENTRY                   ADDRESS         REF BY
           =====                   =======         ======
           TIMER0_A0_ISR           48D4
               interrupt function
    -------------------------------------------------------------------------...
...
...
                ****************************************
                *                                      *
                *      SEGMENTS IN ADDRESS ORDER       *
                *                                      *
                ****************************************


SEGMENT                SPACE    START ADDRESS   END ADDRESS     SIZE  TYPE  ALIGN
=======                =====    =============   ===========     ====  ====  =====
...
...
IPE_B1                                  4800 - 4807                8   rel   10
IPEDATA16_N                             4808 - 4809                2   rel    0
IPECODE16                               480A - 48DD               D4   rel    1
IPEDATA16_C                             48DE - 48ED               10   rel    1
IPE_B2                                     4C00                        rel   10
```

The IPE struct is built into the binary image for the code (.txt or .hex) to allow it to be found by the device boot code at first start-up as described in the *MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family User's Guide* (SLAU367). This binary can also be viewed by going to Project > Options > Linker, and on the Output tab select "Allow C-SPY-specific extra output file".


Options for node "IPE_FR59xx"

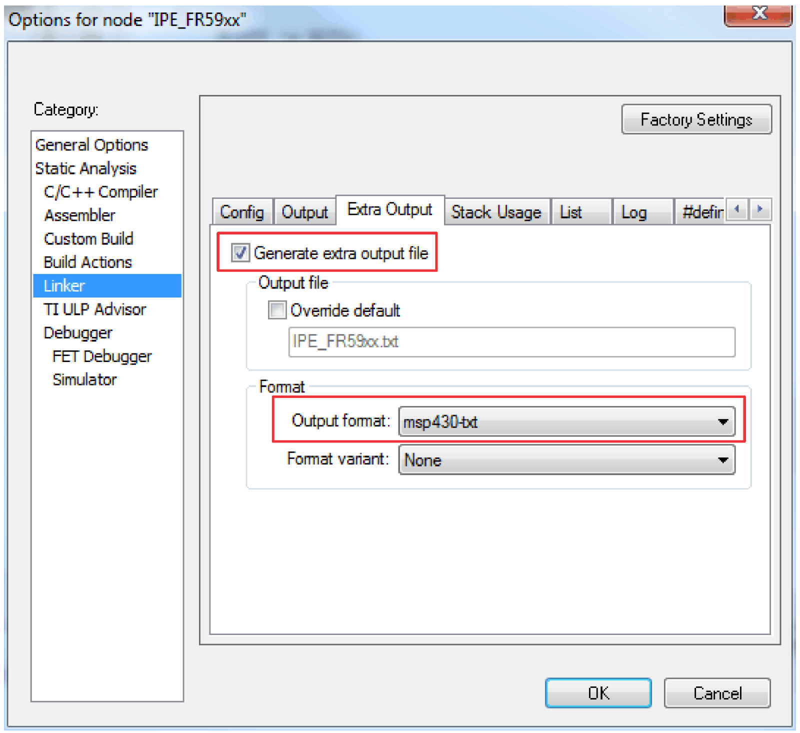
Category:
General Options
Static Analysis
C/C++ Compiler
Assembler
Custom Build
Build Actions
Linker
TI ULP Advisor
Debugger
FET Debugger
Simulator

Output file
[ ] Override default
IPE_FR59xx.d43
Secondary output file:
(None for the selected format)

Format
[● Debug information for C-SPY]
[✓] With runtime control modules
[✓] With I/O emulation modules
[ ] Buffered terminal output
[✓] Allow C-SPY-specific extra output file
○ Other
Output format: msp430-txt
Format variant: None
Module-local symbols: Include all

Figure 11. Options for IPE_FR59xx

On the Extra Output tab, select "Generate extra output file" and set the Output format to "msp430-txt". Build, and a .txt file should appear in the Output folder.


Options for node "IPE_FR59xx"

Category:
General Options
Static Analysis
C/C++ Compiler
Assembler
Custom Build
Build Actions
Linker
TI ULP Advisor
Debugger
FET Debugger
Simulator

[✓] Generate extra output file
Output file
[ ] Override default
IPE_FR59xx.txt
Format
Output format: msp430-txt
Format variant: None

Figure 12. Generating .txt Binary File in IAR

When viewing the txt file to view the IPE initialization struct, verify these items:

1. Check the value at 0xFF8A. This is the portion of the IPE signature that contains the address of the IPE struct. Remember that the value has been shifted right by 4 bits. In the example, the value at 0xFF8A is 0x0480, which points to address 0x4800 for the ipe struct.

```
@FF88
AA AA 80 04
```

2. Go to the address of the IPE initialization struct (found in step 1, in this case 0x4800), and see the values in the structure. It is in the format from the user's guide.

Table 8. IPE Initialization Structure

Field Name	Address Offset	Length	Description
MPUIPC0	0h	word	Control setting for IP Encapsulation. Value is written to MPUIPC0
MPUIPB2	2h	word	Upper border of IP Encapsulation segment. Value is written to MPUIPSEGB2.
MPUIPB1	4h	word	Lower border of IP Encapsulation segment. Value is written to MPUIPSEGB1.
MPUCHECK	6h	word	Odd bit interleaved parity

In the following example, you can see:

- MPUIPC0 = 0x0040 – MPUIPENA = 1 = IPE enable
- MPUIPB2 = 0x04C0 → address 0x4C00 is IPE end
- MPUIPB1 = 0x0480 → address 0x4800 is IPE start
- MPUCHECK = 0xFFFF → checksum of the previous data. Calculated where checksum lower byte = INV(Byte 0 XOR Byte 2 XOR Byte 4) = INV(40 XOR C0 XOR 80) = 0xFF, and checksum upper byte = INV(Byte 1 XOR Byte 3 XOR Byte 5) = INV(00 XOR 04 XOR 04) = 0xFF

```
@4800
40 00 C0 04 80 04 FF FF
```

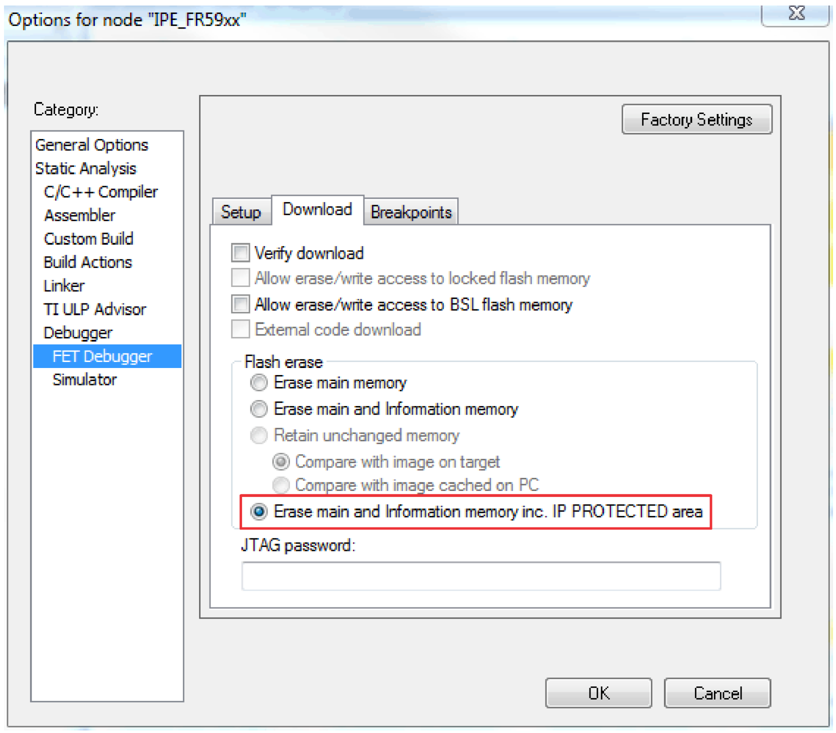
3.2.5 Running and Testing IPE Code in IAR

This code is meant to be run on the MSP-EXP430FR5969 LaunchPad Development Kit.

1. Download the example (http://www.ti.com/lit/zip/slaa685).
2. Open the workspace IPE_FR59xx.eww in IAR EW430 6.30.2 or greater.
3. Build the project.

3.2.5.1 Debug Settings With IPE in IAR

The project has to configure the device to erase the IP protected area before trying to program the part. Otherwise, the program load fails if IPE code is present, because IPE prevents programming access from JTAG. However, a special erase can be performed by the toolchain to mass erase the part and also erase the secured nonvolatile system data area that stores the saved IPE structure pointer [see "Trapdoor Mechanism for IP Structure Pointer Transfer" in the *MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family User's Guide* (SLAU367)]. This is set in Project > Options > Debugger > FET Debugger on the Download tab. Select "Erase main and Information memory inc. IP PROTECTED area".



Download tab settings:

- [ ] Verify download
- [ ] Allow erase/write access to locked flash memory
- [ ] Allow erase/write access to BSL flash memory
- [ ] External code download

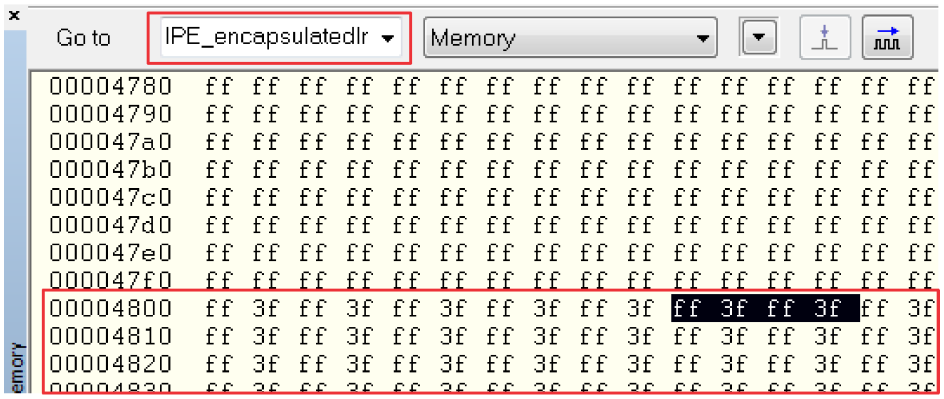
Flash erase
- ○ Erase main memory
- ○ Erase main and Information memory
- ○ Retain unchanged memory
  - ● Compare with image on target
  - ○ Compare with image cached on PC
- [✓ Erase main and Information memory inc. IP PROTECTED area]

JTAG password:

Figure 13. IPE Debug Configurations in IAR

3.2.5.2 Testing IPE in IAR

1. Click the Download and Debug icon.
2. The program loads into the device, and runs the code. The LEDs on the LaunchPad should blink.
3. Pause the code.
   1. Select View > Memory to open the Memory view. In the Memory view, enter the name of IPE protected variables or functions (IPE_encapsulatedInit, IPE_encapsulatedBlink, and so on), or addresses like 0x4800.
   2. Observe that the IPE encapsulated variables and functions are not viewable. 0x3FFF (JMP$) is all that the tool can see in these areas. IP encapsulation is active and preventing readout of the memory area.



```
00004780  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
00004790  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
000047a0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
000047b0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
000047c0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
000047d0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
000047e0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
000047f0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
00004800  ff 3f ff 3f ff 3f ff 3f ff 3f ff 3f ff 3f ff 3f
00004810  ff 3f ff 3f ff 3f ff 3f ff 3f ff 3f ff 3f ff 3f
00004820  ff 3f ff 3f ff 3f ff 3f ff 3f ff 3f ff 3f ff 3f
00004830  ff 3f ff 3f ff 3f ff 3f ff 3f ff 3f ff 3f ff 3f
```

Figure 14. IPE Memory View in IAR - Protected

NOTE: When a different user project is loaded, if IPE was enabled in the code that was previously loaded in the device, the IPE area causes the new project not to load correctly and can give an error in IAR (because the IPE area is protected from erase or write). To load the new project the first time, select "Erase main and Information memory inc. IP PROTECTED area" in the debug settings for the new project for the first time you load it, to erase the IPE code and settings.

4 Bootloader (BSL) Security Features

The MSP430 bootloader allows a way for users to do field firmware updates, even when JTAG/SBW access has been locked or disabled. It can also be a valuable tool for users trying to read out memory or run tests on field-returned units from their end customers, where the JTAG/SBW access was locked or disabled at production. The BSL provides a number of features to help prevent misuse of this BSL access feature. For more information about the BSL implementation on a particular device, see the appropriate BSL user's guide for that device [*MSP430 Programming With the Bootloader (BSL)* (SLAU319) or *MSP430FR57xx, FR58xx, FR59xx, FR68xx, and FR69xx Bootloader (BSL) User's Guide* (SLAU550)]. Also see the device data sheet to determine if the device has a built-in default BSL. Most MSP430 devices implement BSL, but some devices (like MSP430G2xx1/G2xx2 and MSP430i2040) do not support a built-in BSL.

NOTE: For information on securing an MSP432 device or using MSP432 device BSL, see the application report *Configuring Security and Bootloader (BSL) on MSP432P4xx* (SLAA659).

4.1 Password Protection

All MSP430 bootloaders provide password protection on some commands. Typically, any commands that allow readout of the device, like TX_DATA_BLOCK, or control of the device, like Load PC or RX_DATA_BLOCK, are protected and do not execute unless the correct password was already provided. Usually, the only commands not protected are RX_PASSWORD to receive the BSL password and unlock the other commands, CHANGE_BAUD_RATE for changing the BSL baud rate to facilitate BSL communication, and MASS_ERASE for totally wiping the device memory.

The password consists of a number of bytes, typically 32 bytes, located at the end of the interrupt vector table up to memory address 0xFFFF (see the BSL user's guide for more details on specific addresses for the device). Almost all MSP430 BSLs have the password being made up of the data at the addresses 0xFFE0-0xFFFF (excluding only MSP430F54xx non-A devices that have only a 16-byte password).

Because the BSL password consists of values from the device interrupt vector table (IVT), on a programmed device the IVT always has a value other than 0xFFFF at least for the reset vector at address 0xFFFE (in addition to any other interrupt vectors used in the application), there is always some BSL password set simply by loading a program into the device.

In CCS, if any interrupt vectors have no ISRs defined by the user, the TI C/C++ compiler automatically adds a trap ISR to catch any of these unused vectors. Because of this, the unused vectors have some value other than 0xFFFF, but it is the same value for all unused vectors. This adds some more values to the password, but if the user code is not using many interrupts it still might not have very much variation to it.

```
@FFE0
1A 4C 1A 4C 1A 4C 1A 4C 1A 4C 9C 48 1A 4C 1A 4C
1A 4C 1A 4C 1A 4C 1A 4C 1A 4C 1A 4C 1A 4C 00 4C
```

In the TI-txt file snippet above, observe the IVT area that contains the BSL password – 32 bytes between 0xFFE0-0xFFFF. The values 489Ch and 4C00h are highlighted in green text (these are used interrupt vectors) in this example, for one of the Timer ISRs and for the Reset vector. Observe how the reset vector at FFFEh is indicating that the start-up code begins at 4C00h. All other IVT entries, which are unused, have been filled with 4C1Ah – this is the address of the trap ISR. Looking in the .map file, the address of the trap ISR is also found, marked __TI_ISR_TRAP. Other ISR addresses are also found under the _isr section.

```
.text:_isr
*          0    00004c00    00000020
                  00004c00    0000001a     rts430x_lc_rd_eabi.lib : boot_special.obj
(.text:_isr:_c_int00_noargs_noexit)
                  00004c1a    00000006                            : isr_trap.obj
(.text:_isr:__TI_ISR_TRAP)
```

If it is desired to customize the BSL password, the unused entries of the IVT could be modified to other values for the password. However, the danger in this case is that if an interrupt comes in, the IVT entry is used as the address for code execution to jump to, so putting random values here could cause code to run wild or reset the part especially if an erroneous interrupt occurs (for example if an interrupt had been enabled unintentionally). This may not be a very likely scenario but still does pose some risk, and therefore, the user must decide based on their system and application. Alternately, the user can define separate ISRs for all unused interrupts with IVT entries in the BSL password area FFE0h–FFFFh, and use linker file modification and pragmas to place these at desired addresses to create the desired BSL password.

4.2 Mass Erase on Incorrect Password

By default, on most MSP430 bootloaders, an incorrect BSL password results in mass erase of the device. The mass erase causes all code to be erased. This feature adds another layer of security by helping to prevent someone from doing a "brute force" guessing of the password by trying every combination of 32 bytes. If the password is not correct on the first attempt, the part is mass erased and now there is no code to be read out. See the corresponding device user's guide to determine if a particular device supports this feature. A BSL mass erase does not erase the IPE regions and IPE security settings are retained, if enabled.

The mass erase also erases the IVT, so after one bad password attempt the firmware is gone and the BSL password then returns to the default blank value of FFh for all 32 bytes. This means that after one bad password the default password can then be used to access the device – this helps for cases where perhaps a bad load or corrupted firmware was present in the device, where the BSL password may not be correct, by still allowing for a load of new firmware. This should be considered however as a possibility if there is a concern about unauthorized sources being able to load new firmware into the unit – in that case, other measures may need to be taken.

Bootloaders that the support mass erase on incorrect password feature also provide a way to disable this feature if desired. This is done by setting a BSL signature in the device memory to indicate that mass erase should not occur on an incorrect BSL password. More information on disabling this feature can be found in the BSL user's guide or device user's guide. Note that in this case the device may be vulnerable to more "brute force" type of attempts to guess the password.

Table 9. BSL Signature Functionality on MSP430FR5xx/FR6xx Devices

Name	Addresses	Value	Device Security
BSL Password	FFE0h–FFFFh	User Defined + Vector Table Configuration	This password must be provided by BSL host before the device is accessible by the BSL.
BSL Signature	FF84h–FF87h	5555_5555h	BSL is disabled.
		AAAA_AAAAh	BSL is password-protected. Mass erase on wrong BSL password feature is disabled.
		Any other value	BSL is password-protected. Mass erase on wrong BSL password.

Table 10. BSL Signature Functionality on MSP430F1xx/F2xx/F4xx Devices

Name	Addresses	Value	Device Security
BSL Password	FFE0h–FFFFh	User Defined + Vector Table Configuration	This password must be provided by BSL host before the device is accessible by the BSL.
BSL Signature	Data word beneath the IVT[(1)]	AA55h	BSL is disabled.
		0000h	BSL is password-protected. Mass erase on wrong BSL password feature is disabled.
		Any other value	BSL is password-protected. Mass erase on wrong BSL password.

[(1)] The start of the IVT varies depending on the device. Refer to the "Memory Organization" section of the device-specific data sheet. For example, the MSP430F2131 IVT starts at FFE0h. Therefore, the BSL signature on that device is located at FFDEh–FFDFh.

4.3 Disabling Bootloader (BSL)

If the bootloader is not used in the application, or if there is a concern about an unauthorized user obtaining access through the BSL despite the password feature mentioned above, BSL can also be completely disabled. On devices with a Flash-based BSL, the BSL area of Flash can simply be erased when programming the device at production. On devices with a ROM-based BSL, the BSL can be completely disabled by setting a BSL signature in the device main memory as shown in Table 9 and Table 10. See the appropriate BSL user's guide for more information:

MSP430 Programming With the Bootloader (BSL) (SLAU319)

MSP430FR57xx, FR58xx, FR59xx, FR68xx, and FR69xx Bootloader (BSL) User's Guide (SLAU550)

NOTE: If the bootloader is disabled completely and JTAG/SBW access is also disabled, not only is there no possibility for field firmware updates or patches, but there is also no way to recover the part by loading new code if an issue like memory corruption occurs. It also means that diagnosing any issues encountered in the field may not be possible since it is not possible to read out or load new code for diagnostic testing. This needs to be taken into careful consideration when making the decision to disable BSL rather than using password protection and other features – whether the benefits in the particular situation outweigh the risk of not being able to do updates or diagnose code issues in the field.

5 References

1. *MSP430 FRAM Technology – How To and Best Practices* (SLAA628)
2. *Code Composer Studio v6.1 for MSP430 User's Guide* (SLAU157)
3. *MSP430 Optimizing C/C++ Compiler v4.4 User's Guide* (SLAU132)
4. *MSP430 Programming Via the JTAG Interface* (SLAU320)
5. *MSP430 Programming With the Bootloader (BSL)* (SLAU319)
6. *MSP430FR57xx, MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Bootstrap Loader (BSL)* (SLAU550)
7. *MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family User's Guide* (SLAU367)
8. *MSP430x5xx and MSP430x6xx Family User's Guide* (SLAU208)
9. *MSP430x2xx Family User's Guide* (SLAU144)
10. *MSP430i2xx Family User's Guide* (SLAU335)