User's Guide

Code Composer Studio™ IDE v10.x for MSP430™ MCUs

ABSTRACT

This manual describes the use of TI Code Composer Studio™ IDE v10.x (CCS v10.x) with the MSP430™ ultra-low-power microcontrollers. This document applies only for the Windows version of the Code Composer Studio IDE. The Linux® version is similar and, therefore, is not described separately.

Table of Contents

List of Figures

List of Tables

Trademarks

1 Read This First

1.1 How to Use This Manual

Read and follow the instructions in Section 2. This section includes instructions on installing the software and describes how to run the demonstration programs. After you see how quick and easy it is to use the development tools, TI recommends that you read all of this manual.

This manual describes only the setup and basic operation of the software development environment but does not fully describe the MSP430 microcontrollers or the complete development software and hardware systems. For details on these items, see the appropriate TI documents listed in Section 2.3, *Important MSP430 Documents on the Web*, and in Section 1.3.

This manual applies to the use of CCS with the TI MSP-FET, MSP-FET430UIF, eZ-FET, and eZ430 development tools series.

These tools contain the most up-to-date materials available at the time of packaging. For the latest materials (including data sheets, user's guides, software, and application information), visit the TI MSP430 website at www.ti.com/msp430 or contact your local TI sales office.

1.2 Information About Cautions and Warnings

This document may contain cautions and warnings.

CAUTION

This is an example of a caution statement.

A caution statement describes a situation that could potentially damage your software or equipment.

Warning

This is an example of a warning statement.

A warning statement describes a situation that could potentially cause harm to you.

The information in a caution or a warning is provided for your protection. Read each caution and warning carefully.

1.3 Related Documentation From Texas Instruments

CCS documentation

MSP430™ Assembly Language Tools User's Guide

MSP430™ Optimizing C/C++ Compiler User's Guide

MSP430 development tools documentation

MSP Debuggers User's Guide

MSP430™ Hardware Tools User's Guide

eZ430-F2013 Development Tool User's Guide

eZ430-RF2480 User's Guide

eZ430-RF2500 Development Tool User's Guide

eZ430-RF2500-SEH Development Tool User's Guide

eZ430-Chronos™ Development Tool User's Guide

MSP-EXP430G2 LaunchPad™ Experimenter Board User's Guide

Advanced Debugging Using the Enhanced Emulation Module (EEM) With Code Composer Studio IDE

MSP430 device Family User's Guides

MSP430F1xx Family User's Guide

MSP430F2xx Family User's Guide

MSP430F3xx Family User's Guide

MSP430F4xx Family User's Guide

MSP430F5xx and MSP430x6xx Family User's Guide

MSP430FR4xx and MSP430FR2xx Family User's Guide

MSP430FR57xx Family User's Guide

MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide

CC430 device Family User's Guide

CC430 Family User's Guide

1.4 If You Need Assistance

Support for the MSP430 microcontrollers and the FET development tools is provided by the TI Product Information Center (PIC). Contact information for the PIC can be found on the TI website at www.ti.com/support. A Code Composer Studio specific Wiki page (FAQ) is available, and the TI E2E™ support forums for the MSP430 microcontrollers and the Code Composer Studio IDE provide open interaction with peer engineers, TI engineers, and other experts. Additional device-specific information can be found on the MSP430 website.

2 Get Started Now!

This section provides instructions on installing the software, and shows how to run the demonstration programs.

2.1 Software Installation

To install Code Composer Studio™ IDE (CCS), download the corresponding CCS package to the HostOS platform, and extract the full zip archive before running ccs_setup_x.x.x.x. The user can select to use the offline or online installer (TI recommends the offline installer for slow and unreliable connections). Follow the instructions shown on the screen. The hardware drivers for the USB JTAG emulators (MSP-FET, MSPFET430UIF, eZ-FET, and eZ430 series) are installed automatically when installing CCS. The parallel-port FET (MSP-FET430PIF) legacy debug interface is no longer supported in this version of CCS.

Note

The legacy MSP-FET430PIF (parallel port emulator) is not supported by this version of CCS.

Note

Fully extract the zip archive (CCSx.x.x.x_y.zip) before running ccs_setup_x.x.x.x.

Note

If the MSP-FET or eZ-FET debugger driver install fails:

Under certain conditions (depending on the hardware and operating system that is used), the MSP-FET or eZ-FET driver install may fail on the first attempt. This can lead to unresponsive behavior on IDEs. To resolve this issue, disconnect the MSP-FET or eZ-FET and then plug it again, or plug it in to a different USB port, and restart the IDE.

Table 2-1. System Requirements

	Recommended System Requirements	Minimum System Requirements
Processor	Dual Core x86 compatible processor	1.0-GHz x86 compatible processor
RAM	6GB	2GB
Free Disk Space	2GB average (1 or 2 device families); 3.5GB all features	900MB (depends on features selected during installation)
Operating System	• Windows®: Windows 7 (SP1 or later), Windows 8.x and Windows 10 • Linux: Details on the Linux distributions supported is available here: http://software-dl.ti.com/ccs/esd/documents/ccsv10_linux_host_support.html • Mac: The most current and the previous versions are supported at the time of CCS release.	

2.2 Flashing the LED

This section demonstrates on the FET the equivalent of the C-language "Hello world!" introductory program. CCS includes C code template files that allow flashing the LED in no time. To get started:

1. Start Code Composer Studio by clicking **Start** → **All Programs** → **Texas Instruments** → **Code Composer Studio** → **Code Composer Studio**.
2. Create a new Project by clicking **File** → **New** → **CCS Project**.
3. Enter a project name.
4. Set the Device Family to MSP430 and select the Device Variant to use (for example, MSP430F2274).
5. Select "Blink The LED" in the "Project templates and example" section.
6. Click Finish.

Note

The predefined examples work with most MSP430 boards. Specific examples are automatically selected for MSP430G221x, MSP430L092, and MSP430FR59xx devices. Certain MSP430F4xx boards use Port P5.0 for the LED connection, which must be changed manually in the code.

7. To compile the code and download the application to the target device, click **Run** → **Debug (F11)**.

CAUTION

Never disconnect the JTAG or emulator USB cable during an active debug session. Always terminate a running debug session properly (by clicking on the "Terminate" icon) before disconnecting the target device.

8. To start the application, click **Run** → **Resume (F8)** or click the Play button on the toolbar.

See FAQ 1 if the CCS debugger is unable to communicate with the device.

Congratulations, you have just built and tested an MSP430 application!

2.3 Important MSP430™ Documents

The primary sources of MSP430 and CCS information are the device-specific data sheets and user's guides. The MSP430 website (www.ti.com/msp430) contains the latest version of these documents.

Documents describing the Code Composer Studio tools (Code Composer Studio IDE, assembler, C compiler, linker, and librarian) can be found at www.ti.com/tool/ccstudio. A Code Composer Studio specific Wiki page (FAQ) is available at processors.wiki.ti.com/index.php/Category:CCS, and the TI E2E support forums at e2e.ti.com provide additional help. Documentation for third party tools, such as IAR Embedded Workbench for MSP430, can usually be found on the respective third-party website.

3 Development Flow

This section describes how to use Code Composer Studio IDE (CCS) to develop application software and how to debug that software.

3.1 Using Code Composer Studio™ IDE (CCS)

The following sections are an overview of how to use CCS. For a full description of the software development flow with CCS in assembly or C, see the *MSP430 Assembly Language Tools User's Guide* and the *MSP430 Optimizing C/C++ Compiler User's Guide*.

3.1.1 Creating a Project From Scratch

This section presents step-by-step instructions to create an assembly or C project from scratch and to download and run an application on the MSP430 (see Section 3.1.2). Also, the MSP430 Code Composer Studio Help presents a more comprehensive overview of the process.

1. Start CCS (**Start** → **All Programs** → **Texas Instruments** → **Code Composer Studio** → **Code Composer Studio**).
2. Create new project (**File** → **New** → **CCS Project**). Enter the name for the project, click next and set Device Family to MSP430.
3. Select the appropriate device variant. For assembly only projects, select **Empty Assembly-only Project** in the **Project template and examples** section.
4. If using a USB Flash Emulation Tool such as the MSP-FET, MSP-FET430UIF, eZ-FET, or the eZ430 Development Tool, they should be already configured by default.
5. For C projects the setup is complete now, main.c is shown, and code can be entered. For Assembly only projects the main.asm is shown. If, instead, you want to use an existing source file for your project, click **Project** → **Add Files...** and browse to the file of interest. Single click on the file and click Open or double-click on the file name to complete the addition of it into the project folder.
6. Click Finish.
7. Enter the program text into the file.

Note

Use MSP430 headers (*.h files) to simplify code development.

CCS is supplied with files for each device that define the device registers and the bit names. TI recommends using these files, which and can greatly simplify the task of developing a program. To include the .h file corresponding to the target device, add the line **#include <msp430xyyy.h>** for C and **.cdecls C,LIST,"msp430xyyy.h"** for assembly code, where xyyy specifies the MSP430 part number.

8. Build the project (**Project** → **Build Project**).
9. Debug the application (**Run** → **Debug (F11)**). This starts the debugger, which gains control of the target, erases the target memory, programs the target memory with the application, and resets the target.

   See FAQ 1 if the debugger is unable to communicate with the device.
10. Click **Run** → **Resume (F8)** to start the application.
11. Click **Run** → **Terminate** to stop the application and to exit the debugger. CCS returns to the C/C++ view (code editor) automatically.

CAUTION

Never disconnect the JTAG or emulator USB cable during an active debug session. Always terminate a running debug session properly (by clicking on the "Terminate" icon) before disconnecting the target device.

12. Click **File** → **Exit** to exit CCS.

3.1.2 Project Settings

The settings required to configure the CCS are numerous and detailed. Most projects can be compiled and debugged with default factory settings. The project settings are accessed by clicking **Project → Properties** for the active project. The following project settings are recommended or required:

- Specify the target device for debug session (**Project → Properties → General → Device → Variant**). The corresponding Linker Command File and Runtime Support Library are selected automatically.
- To more easily debug a C project, disable optimization (**Project → Properties → Build → MSP430 Compiler → Optimization → Optimization level**).
- Specify the search path for the C preprocessor (**Project → Properties → Build → MSP430 Compiler → Include Options**).
- Specify the search path for any libraries being used (**Project → Properties → Build → MSP430 Compiler → File Search Path**).
- Specify the debugger interface (**Project → Properties → General → Device → Connection**). Select TI MSP430 LPTx for the parallel FET interface or TI MSP430 USBx for the USB interface.
- Enable the erasure of the Main and Information memories before object code download (**Project → Properties → Debug → MSP430 Properties → Download Options → Erase Main and Information Memory**).
- To ensure proper stand-alone operation, select Hardware Breakpoints (**Project → Properties → Debug → MSP430 Properties**). If Software Breakpoints are enabled (**Project → Properties → Debug → Misc/Other Options → Allow software breakpoints to be used**), ensure proper termination of each debug session while the target is connected; otherwise, the target may not be operational stand-alone as the application on the device still contains the software breakpoint instructions.

3.1.3 Using Math Library for MSP430 (MSPMathlib) in CCS v5.5 and Newer

TI's MSPMathlib is part of CCSv5.5 and newer releases. This optimized library provides up to 26x better performance in applications that use floating point scalar math. For details, see the MSPMathlib web page (www.ti.com/tool/mspmathlib).

MSPMathlib is active by default in CCSv5.5+ for all new projects on all supported devices. For imported projects, it is used only if the project already uses MSPMathlib or if it has been manually enabled.

To disable MSPMathlib: Remove libmath.a under **Project → Properties → Build → MSP430 Linker → File Search Path** in the "Include library file or command file as input (--library, -l)" field.

To enable MSPMathlib: Add libmath.a under **Project → Properties → Build → MSP430 Linker → File Search Path** in the "Include library file or command file as input (--library, -l)" field. Important: Put libmath.a before other libraries that may be listed here.

3.1.4 Using an Existing CCE v2.x, CCE v3.x, CCS v4.x, CCS v5.x, CCS v6.x, CCS v7.x, CCS v8.x, or CCS v9.x Project

CCS v10.x supports the conversion of workspaces and projects created in version CCE v2.x, CCE v3.x, CCS v4.x, CCS v5.x, CCS v6.x, CCS v7.x, CCS v8.x, or CCS v9.x to the CCS v10.x format (**File → Import → General → Existing Projects into Workspace → Next**). Browse to legacy CCE or CCS workspace that contains the project to be imported. The Import Wizard lists all of the projects in the given workspace. Specific Projects can then be selected and converted. CCEv2 and CCEv3 projects may require manual changes to the target configuration file (*.ccxml) after import.

CCS may return a warning that an imported project was built with another version of Code Generation Tools (CGT) depending on the previous CGT version.

While the support for assembly projects has not changed, the header files for C code have been modified slightly to improve compatibility with the IAR Embedded Workbench® IDE (interrupt vector definitions). The definitions used in CCE 2.x are still given but have been commented out in all header files. To support CCE 2.x C code,

remove the "//" in front of the #define statements that are located at the end of each .h file in the section "Interrupt Vectors".

3.1.5 Stack Management

The reserved stack size can be configured through the project options (**Project → Properties → Build → MSP430 Linker → Basic Options → Set C System Stack Size**). Stack size is defined to extend from the last location of RAM for 50 to 80 bytes (that is, the stack extends downwards through RAM for 50 to 80 bytes, depending on the RAM size of the selected device).

The stack can overflow due to small size or application errors. See Section 3.2.2.1 for a method of tracking the stack size.

3.1.6 How to Generate Binary Format Files (TI-TXT and INTEL-HEX)

The CCS installation includes the hex430.exe conversion tool. It can be configured to generate output objects in TI-TXT format for use with the MSP-GANG as well as INTEL-HEX format files for TI factory device programming. The tool can be used either stand-alone in a command line (located in `<Installation Root>\ccsv6\tools\compiler\ti-cgt-msp430_x.x.x\bin`) or directly within CCS. To generate the file automatically after every build, use the MSP430 Hex Utility menu (**Project → Properties → Build → MSP430 Hex Utility**) and select the options there for generating the binary files. The generated file is stored in the <Workspace>\<Project>\Debug\ directory.

3.2 Using the Integrated Debugger

See Section 10 for a description of FET-specific menus within CCS.

3.2.1 Breakpoint Types

The debugger breakpoint mechanism uses a limited number of on-chip debugging resources (specifically, N breakpoint registers, see Table 3-1). When N or fewer breakpoints are set, the application runs at full device speed (or "realtime"). When greater than N breakpoints are set and Use Software Breakpoints is enabled (**Project → Properties → Debug → Misc/Other Options → Allow software breakpoints to be used**), an unlimited number of software breakpoints can be set while still meeting realtime constraints.

Note

A software breakpoint replaces the instruction at the breakpoint address with a call to interrupt the code execution. Therefore, there is a small delay when setting a software breakpoint. In addition, the use of software breakpoints always requires proper termination of each debug session; otherwise, the application may not be operational stand-alone, because the application on the device would still contain the software breakpoint instructions.

Both address (code) and data (value) breakpoints are supported. Data breakpoints and range breakpoints each require two MSP430 hardware breakpoints.

Table 3-1. Device Architecture, Breakpoints, and Other Emulation Features

Device	MSP430 Architecture	4-Wire JTAG	2-Wire JTAG[1]	Break-points (N)	Range Break-points	Clock Control	State Sequencer	Trace Buffer	LPMx.5 Debugging Support
CC430F512x	MSP430Xv2	✓	✓	3	✓	✓			
CC430F513x	MSP430Xv2	✓	✓	3	✓	✓			
CC430F514x	MSP430Xv2	✓	✓	3	✓	✓			
CC430F612x	MSP430Xv2	✓	✓	3	✓	✓			
CC430F613x	MSP430Xv2	✓	✓	3	✓	✓			
CC430F614x	MSP430Xv2	✓	✓	3	✓	✓			
MSP430AFE2xx	MSP430	✓	✓	2		✓			
MSP430BT5190	MSP430Xv2	✓	✓	8	✓	✓	✓	✓	
MSP430F11x1	MSP430	✓		2					
MSP430F11x2	MSP430	✓		2					
MSP430F12x	MSP430	✓		2					
MSP430F12x2	MSP430	✓		2					
MSP430F13x	MSP430	✓		3	✓				
MSP430F14x	MSP430	✓		3	✓				
MSP430F15x	MSP430	✓		8	✓	✓	✓	✓	
MSP430F161x	MSP430	✓		8	✓	✓	✓	✓	
MSP430F16x	MSP430	✓		8	✓	✓	✓	✓	
MSP430F20xx	MSP430	✓	✓	2		✓			
MSP430F21x1	MSP430	✓		2		✓			
MSP430F21x2	MSP430	✓	✓	2		✓			
MSP430F22x2	MSP430	✓	✓	2		✓			
MSP430F22x4	MSP430	✓	✓	2		✓			
MSP430F23x	MSP430	✓		3	✓	✓			
MSP430F23x0	MSP430	✓		2		✓			
MSP430F2410	MSP430	✓		3	✓	✓			
MSP430F241x	MSP430X	✓		8	✓	✓	✓	✓	
MSP430F24x	MSP430	✓		3	✓	✓			
MSP430F261x	MSP430X	✓		8	✓	✓	✓	✓	
MSP430F41x	MSP430	✓		2		✓			
MSP430F41x2	MSP430	✓	✓	2		✓			
MSP430F42x	MSP430	✓		2		✓			
MSP430F42x0	MSP430	✓		2		✓			
MSP430F43x	MSP430	✓		8	✓	✓	✓	✓	
MSP430F43x1	MSP430	✓		2		✓			
MSP430F44x	MSP430	✓		8	✓	✓	✓	✓	
MSP430F44x1	MSP430	✓		8	✓	✓	✓	✓	
MSP430F461x	MSP430X	✓		8	✓	✓	✓	✓	
MSP430F461x1	MSP430X	✓		8	✓	✓	✓	✓	
MSP430F471xx	MSP430X	✓		8	✓	✓	✓	✓	
MSP430F47x	MSP430	✓		2		✓			
MSP430F47x3	MSP430	✓		2		✓			
MSP430F47x4	MSP430	✓		2		✓			
MSP430F51x1	MSP430Xv2	✓	✓	3	✓	✓			

Device	MSP430 Architecture	4-Wire JTAG	2-Wire JTAG[1]	Break-points (N)	Range Break-points	Clock Control	State Sequencer	Trace Buffer	LPMx.5 Debugging Support
MSP430F51x2	MSP430Xv2	✓	✓	3	✓	✓			
MSP430F52xx	MSP430Xv2	✓	✓	3	✓	✓			
MSP430F530x	MSP430Xv2	✓	✓	3	✓	✓			
MSP430F5310	MSP430Xv2	✓	✓	3	✓	✓			
MSP430F532x	MSP430Xv2	✓	✓	8	✓	✓	✓	✓	
MSP430F533x	MSP430Xv2	✓	✓	8	✓	✓	✓	✓	
MSP430F534x	MSP430Xv2	✓	✓	8	✓	✓	✓	✓	
MSP430F535x	MSP430Xv2	✓	✓	8	✓	✓	✓	✓	
MSP430F54xx	MSP430Xv2	✓	✓	8	✓	✓	✓	✓	
MSP430F54xxA	MSP430Xv2	✓	✓	8	✓	✓	✓	✓	
MSP430F550x	MSP430Xv2	✓	✓	3	✓	✓			
MSP430F5510	MSP430Xv2	✓	✓	3	✓	✓			
MSP430F552x	MSP430Xv2	✓	✓	8	✓	✓	✓	✓	
MSP430F563x	MSP430Xv2	✓	✓	8	✓	✓	✓	✓	
MSP430F565x	MSP430Xv2	✓	✓	8	✓	✓	✓	✓	
MSP430F643x	MSP430Xv2	✓	✓	8	✓	✓	✓	✓	
MSP430F645x	MSP430Xv2	✓	✓	8	✓	✓	✓	✓	
MSP430F663x	MSP430Xv2	✓	✓	8	✓	✓	✓	✓	
MSP430F665x	MSP430Xv2	✓	✓	8	✓	✓	✓	✓	
MSP430F67xx	MSP430Xv2	✓	✓	3	✓	✓			
MSP430F67xx1	MSP430Xv2	✓	✓	3	✓	✓			
MSP430F67xx1A	MSP430Xv2	✓	✓	3	✓	✓			
MSP430F67xxA	MSP430Xv2	✓	✓	3	✓	✓			
MSP430FE42x	MSP430	✓		2		✓			
MSP430FE42x2	MSP430	✓		2		✓			
MSP430FG42x0	MSP430	✓		2		✓			
MSP430FG43x	MSP430	✓		2		✓			
MSP430FG461x	MSP430X	✓		8	✓	✓	✓	✓	
MSP430FG47x	MSP430	✓		2		✓			
MSP430FG642x	MSP430Xv2	✓	✓	8	✓	✓	✓	✓	
MSP430FG662x	MSP430Xv2	✓	✓	8	✓	✓	✓	✓	
MSP430FR20xx	MSP430Xv2	✓	✓	3	✓	✓			
MSP430FR21xx	MSP430Xv2	✓	✓	3	✓	✓			
MSP430FR23xx	MSP430Xv2	✓	✓	3	✓	✓			
MSP430FR24xx	MSP430Xv2	✓	✓	3	✓	✓			
MSP430FR25xx	MSP430Xv2	✓	✓	3	✓	✓			
MSP430FR26xx	MSP430Xv2	✓	✓	3	✓	✓			
MSP430FR41xx	MSP430Xv2	✓	✓	3	✓	✓			
MSP430FR57xx	MSP430Xv2	✓	✓	3	✓	✓			
MSP430FR58xx	MSP430Xv2	✓	✓	3	✓	✓			✓
MSP430FR59xx	MSP430Xv2	✓	✓	3	✓	✓			✓
MSP430FR60xx	MSP430Xv2	✓	✓	3	✓	✓			✓
MSP430FR68xx	MSP430Xv2	✓	✓	3	✓	✓			✓
MSP430FR69xx	MSP430Xv2	✓	✓	3	✓	✓			✓

Device	MSP430 Architecture	4-Wire JTAG	2-Wire JTAG(1)	Break-points (N)	Range Break-points	Clock Control	State Sequencer	Trace Buffer	LPMx.5 Debugging Support
MSP430FW42x	MSP430	✓		2		✓			
MSP430G2xxx	MSP430	✓	✓	2		✓			
MSP430i20xx	MSP430	✓	✓	2		✓			
MSP430L092	MSP430Xv2	✓		2		✓			
MSP430SL54xxA	MSP430Xv2	✓	✓	8	✓	✓	✓	✓	
MSP430TCH5E	MSP430	✓	✓	2		✓			
RF430FRL15xH	MSP430Xv2	✓	✓	2		✓			

(1) The 2-wire JTAG debug interface is also referred to as Spy-Bi-Wire (SBW) interface. This interface is supported only by the USB emulators (eZ430-xxxx, eZ-FET, and MSP-FET430UIF USB JTAG emulator) and the MSP-GANG430 production programming tool.

(2) Support is limited to Spy-Bi-Wire (SBW) on MSP-FET430UIF. No limitations on MSP-FET.

3.2.2 Using Breakpoints

If the debugger is started with greater than N breakpoints set and software breakpoints are disabled (**Project → Properties → Debug → Misc/Other Options → Allow software breakpoints to be used** option is unchecked), a message is shown that informs the user that not all breakpoints can be enabled. CCS permits any number of breakpoints to be set, regardless of the Use Software Breakpoints setting of CCS. If software breakpoints are disabled, a maximum of N breakpoints can be set within the debugger.

Resetting a program requires a breakpoint, which is set on the address defined in **Project → Properties → Debug → Auto Run and Launch Options → Auto Run Options → Run to symbol**.

The Run To Cursor operation temporarily requires a breakpoint.

Console I/O (CIO) functions, such as printf, require the use of a breakpoint. If these functions are compiled in, but you do not wish to use a breakpoint, disable CIO functionality by changing the option in **Project → Properties → Debug → Program/Memory Load Options → Program/Memory Load Options → Enable CIO function use** (requires setting a breakpoint).

Note

Do not set a breakpoint on a RETI instruction if the previous instruction modifies the stack pointer. Program execution will not work properly after reaching the break point.

3.2.2.1 Breakpoints in CCS

CCS supports a number of predefined breakpoint types that can be selected by opening a menu found next to the Breakpoints icon in the Breakpoint window (**Window → Show View → Breakpoints**). In addition to traditional breakpoints, CCS allows setting watchpoints to break on a data address access instead of an address access. The properties of breakpoints and watchpoints can be changed in the debugger by right clicking on the breakpoint and selecting Properties (see Figure 3-1 and Figure 3-2).

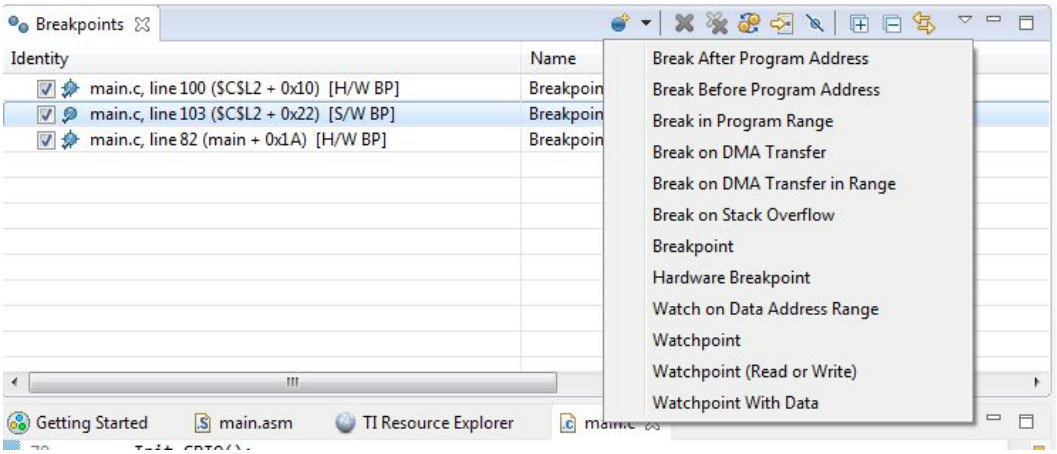

Figure 3-1. Breakpoints

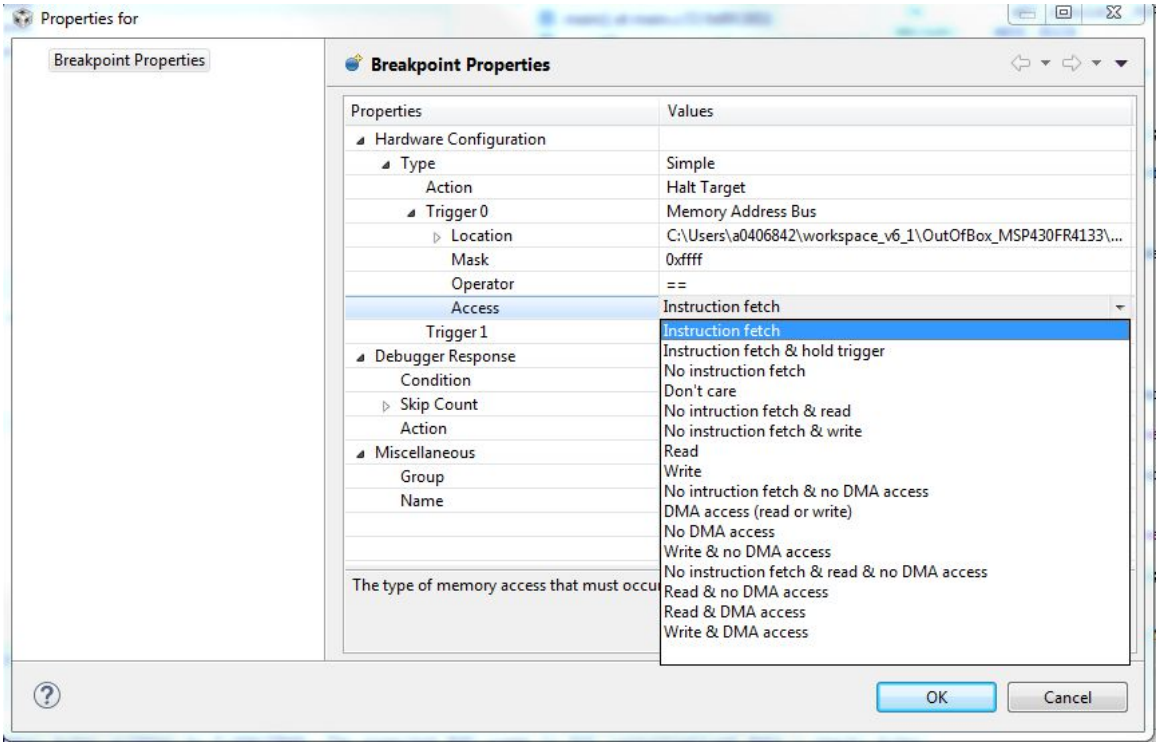

Figure 3-2. Breakpoint Properties

- **Break after program address**

  Stops code execution when the program attempts to execute code after a specific address.
- **Break before program address**

  Stops code execution when the program attempts to execute code before a specific address.
- **Break in program range**

  Stops code execution when the program attempts to execute code in a specific range.
- **Break on DMA transfer**
- **Break on DMA transfer in range**

  Breaks when a DMA access within a specified address range occurs.
- **Break on stack overflow**

  It is possible to debug the applications that caused the stack overflow. Set Break on Stack Overflow (right click in Breakpoints window and then select "Break on Stack Overflow" in the context menu). The program execution stops on the instruction that caused the stack overflow. The size of the stack can be adjusted in Project → Properties → C/C++ Build → MSP430 Linker → Basic Options.
- **Breakpoint**

  Sets a breakpoint.
- **Hardware breakpoint**

  Forces a hardware breakpoint if software breakpoints are not disabled.
- **Watch on data address range**

  Stops code execution when data access to an address in a specific range occurs.
- **Watchpoint**

  Stops code execution if a specific data access to a specific address is made.
- **Watchpoint (Read or Write)**

  Stops code execution if a read or write data access to a specific address is made.
- **Watchpoint with data**

  Stops code execution if a specific data access to a specific address is made with a specific value.

  Restriction 1: Watchpoints are applicable to global variables and non-register local variables. In the latter case, set a breakpoint (BP) to halt execution in the function where observation of the variable is desired (set code breakpoint there). Then set the watchpoint and delete (or disable) the code breakpoint in the function and run or restart the application.

  Restriction 2: Watchpoints are applicable to variables 8 bits and 16 bits wide.

Note

Not all options are available on every MSP430 derivative (see Table 3-1). Therefore, the number of predefined breakpoint types in the breakpoint menu varies depending on the selected device.

For more information on advanced debugging with CCS, see Advanced Debugging Using the Enhanced Emulation Module (EEM) With Code Composer Studio IDE.

3.2.3 Download Options for MSP430 Devices

By default, CCS Debugger downloads the application to RAM or flash when a debug session starts. The Download options (see Figure 3-3) let you modify the behavior of the download.

- **Copy application to external SPI memory after program load**

  Saves user code to external SPI memory.
- **Allow Read/Write/Erase access to BSL memory**

  Enables erase and write access to BSL flash memory.
- **Erase main memory only**

  Erases only the main flash memory before download. The Information memory is not erased.
- **Erase main and Information memory**

  Erases the main and Information flash memories before download.
- **Erase main, information and protected information memory**

  Erases the main and Information flash memories, including the IP protected area, before download.
- **Erase and download necessary segments only (Differential Download)**

  Keeps track of changes in the program image and only writes the portions that have changed between program loads. Theoretically, this should improve load performance for small changes. It can make performance worse for larger changes. It relies on the compiler and linker to radically change the binary image for a small source code change.
- **Replace written memory locations, retain unwritten memory locations**

  Writes only the flash segments that are being written to. This does not track differences in the loaded image and always writes a segment included in the loaded image.
- **By Address Range (specify below)**

  Erases only the flash memory specified segments before download.

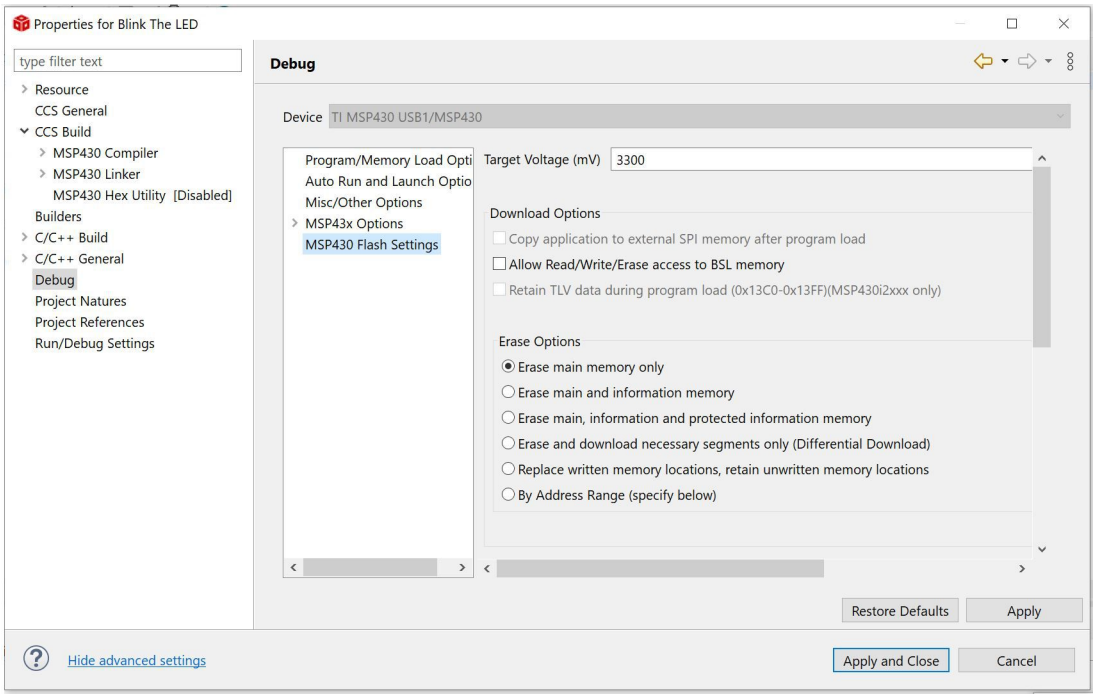

Figure 3-3. Download Options

4 EnergyTrace™ Technology

4.1 Introduction

EnergyTrace™ Technology is an energy-based code analysis tool that measures and displays the application's energy profile and helps to optimize it for ultra-low power consumption.

MSP430 devices with built-in **EnergyTrace+[CPU State]+[Peripheral States]** (or in short **EnergyTrace++**) technology allow real-time monitoring of many internal device states while user program code executes. EnergyTrace++ technology is supported on selected MSP430 devices and debuggers.

EnergyTrace mode (without the "++") is the base of **EnergyTrace Technology** and enables analog energy measurement to determine the energy consumption of an application but does not correlate it to internal device information. The EnergyTrace mode is available for all MSP430 devices with selected debuggers, including CCS.

4.2 Energy Measurement

Debuggers with EnergyTrace Technology support include a new and unique way of continuously measuring the energy supplied to a target microcontroller that differs considerably from the well-known method of amplifying and sampling the voltage drop over a shunt resistor at discrete times. A software-controlled dc-dc converter is used to generate the target power supply. The time density of the dc-dc converter charge pulses equals the energy consumption of the target microcontroller. A built-in on-the-fly calibration circuit defines the energy equivalent of a single dc-dc charge pulse.

Figure 4-1 shows the energy measurement principle. Periods with a small number of charge pulses per time unit indicate low energy consumption and thus low current flow. Periods with a high number of charge pulses per time unit indicate high energy consumption and also a high current consumption. Each charge pulse leads to a rise of the output voltage V_{OUT}, which results in an unavoidable voltage ripple common to all dc-dc converters.

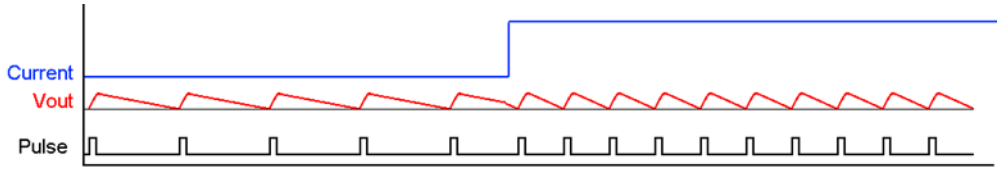

Figure 4-1. Pulse Density and Current Flow

The benefit of sampling continuously is evident: even the shortest device activity that consumes energy contributes to the overall recorded energy. No shunt-based measurement system can achieve this.

4.3 Code Composer Studio™ Integration

EnergyTrace Technology is available as part of Texas Instrument's Code Composer Studio IDE for MSP430 microcontrollers. Additional controls and windows are available if the hardware supports EnergyTrace Technology.

EnergyTrace can be used while debugging an application (debug session) or only to measure the current consumption of standalone running application (without a debug session). Using EnergyTrace without a debug session measures the current consumption of the running application without changing the code content or the CPU states.

During a debugging session, the EnergyTrace and EnergyTrace++ modes are available, depending on the supported hardware features on the target device. Only EnergyTrace mode is available with stand-alone running applications (see Table 4-1).

Table 4-1. Availability of EnergyTrace and EnergyTrace++ Modes

	EnergyTrace	EnergyTrace++
Debugging session	x	x
Stand-alone application	x	

To use EnergyTrace Technology without debugging session

1. Connect your target board embedding the firmware
2. Press the EnergyTrace Technology button in the toolbar menu (see Figure 4-2). You don't need to build or start debug session.

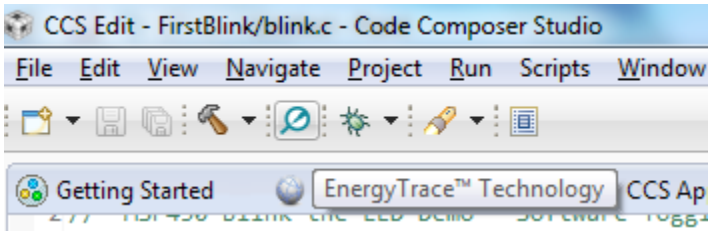

Figure 4-2. EnergyTrace Button in the Toolbar Menu

3. Click the start trace collection button (▶) to start the EnergyTrace Technology measurement.
4. Click the stop trace collection button (●) to stop the EnergyTrace Technology measurement.
5. Refer to Section 4.3.4 for more details.

To exit this mode, click the ⊠ in the EnergyTrace™ Technology window (see Figure 4-3).

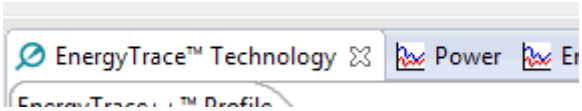

Figure 4-3. Exit EnergyTrace Mode

To use EnergyTrace mode or EnergyTrace++ mode in a debugging session

1. Connect your target board.
2. Select and build your project
3. Start the debug session.
4. The EnergyTrace window is displayed if it has been already selected. If not, click the EnergyTrace button in the toolbar menu.
5. Refer to Section 4.3.3 and Section 4.3.4 for more details.

4.3.1 EnergyTrace Technology Settings

EnergyTrace settings are available in the Code Composer Studio Preferences. Go to **Window → Preferences → Code Composer Studio → Advanced Tools → EnergyTrace™ Technology** (see Figure 4-4).

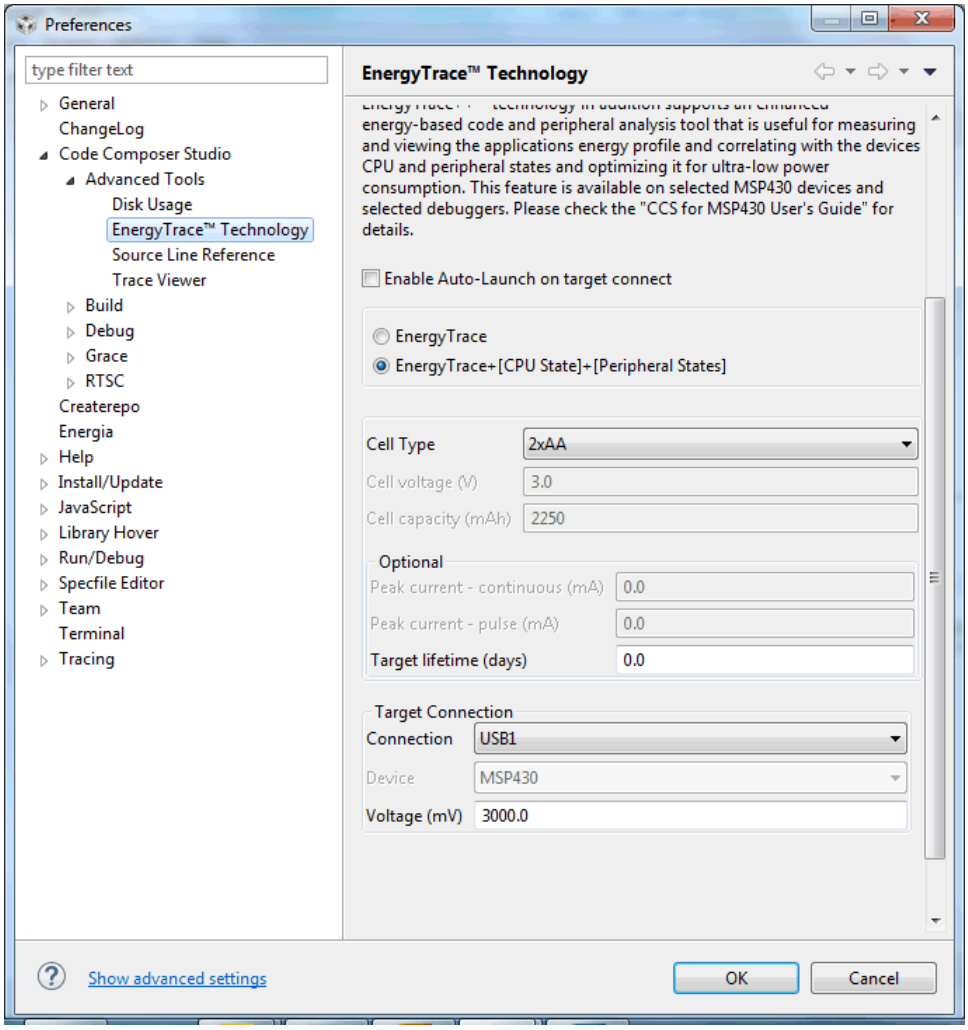

Figure 4-4. EnergyTrace™ Technology Preferences

- **Enable Auto-Launch on target connect**: check this box to enable the EnergyTrace modes when entering a debug session.
- **Two capture modes are supported**:
  - The full-featured **EnergyTrace+[CPU State]+[Peripheral States]** mode that delivers real-time device state information together with energy measurement data
  - The **EnergyTrace** mode that delivers only energy measurement data
  - Use the radio button to select the mode to enable when a debug session is launched. If an MSP430 device does not support device state capturing, the selection is ignored and Code Composer Studio starts in the EnergyTrace mode.

While a debug session is active, click the  icon in the Profile window to switch between the modes.

To use the **EnergyTrace+[CPU State]+[Peripheral States]** mode to capture real-time device state information while an application is executing, the default Debug Properties of the project must also be modified. Right click on the active project in the Project Explorer and click on Properties (see Figure 4-5).

In the Debug section, enable the **Enable Ultra Low Power debug / Debug LPMx.5** option in the Low Power Mode Settings (see Figure 4-6). If this option is not enabled, the **EnergyTrace+[CPU State]+[Peripheral States]** mode cannot capture data from the device.

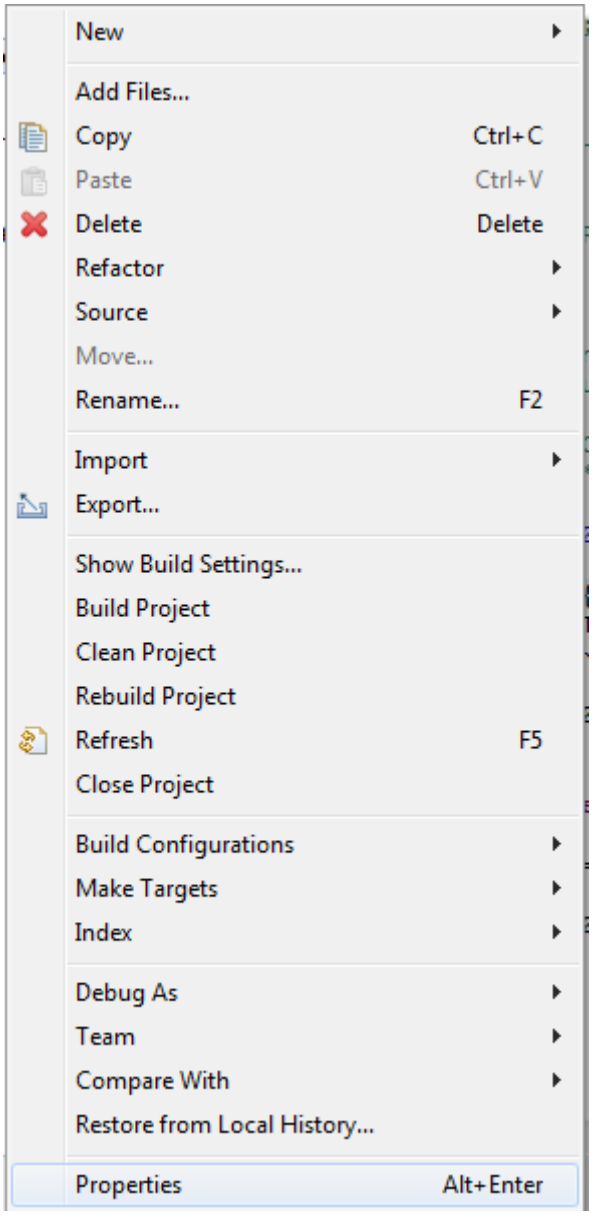

Figure 4-5. Project Properties

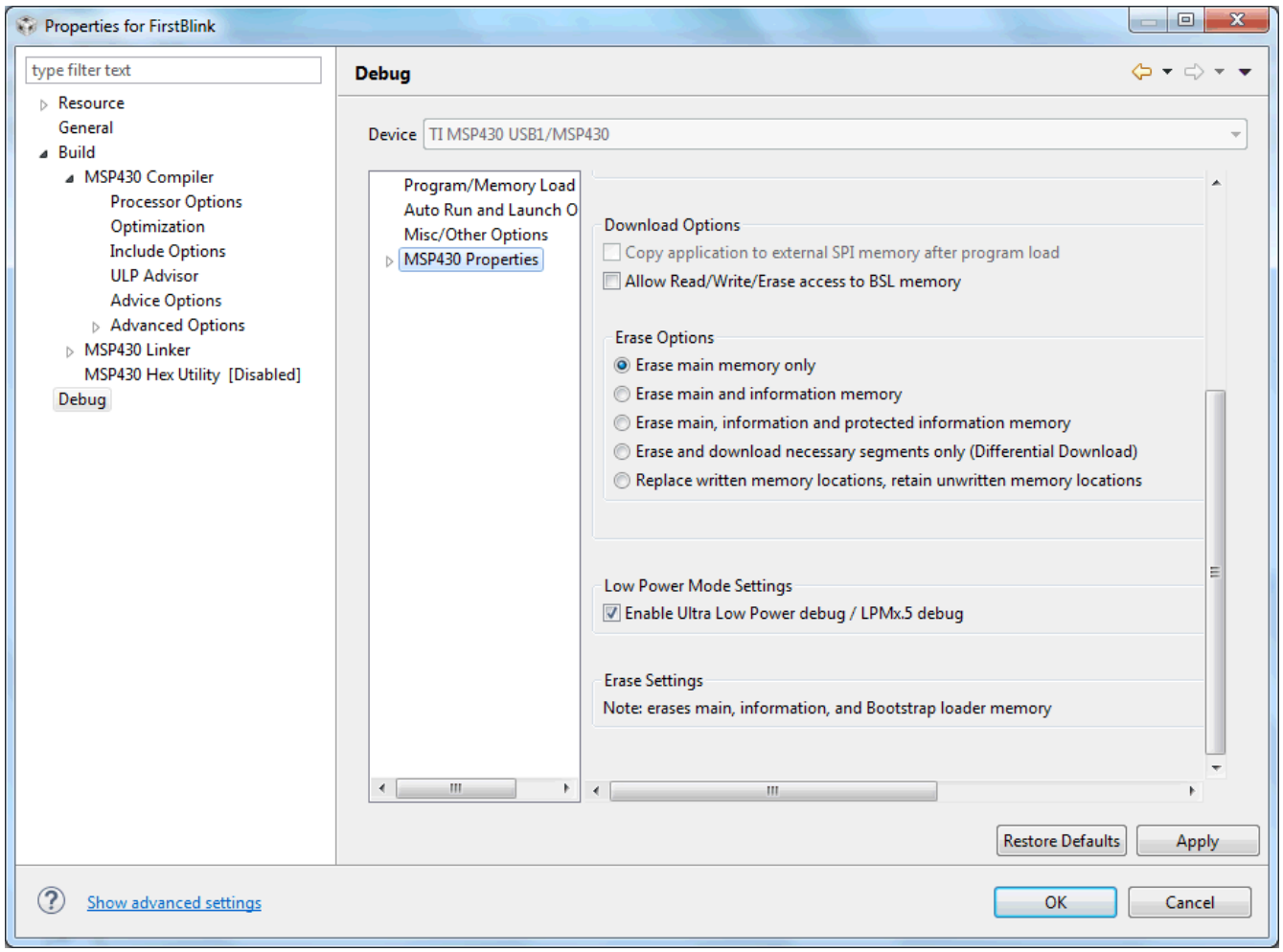

Figure 4-6. Debug Properties

Note

If the EnergyTrace Technology windows are not opened when a debug session starts, verify the following items:

- Does the hardware (debugger and device) support EnergyTrace Technology? To determine if your selected device supports EnergyTrace technology, refer to the device-specific data sheet, the *MSP430 Hardware Tools User's Guide*, or the user guide that came with the evaluation board.
- Is EnergyTrace Technology globally enabled in **Window → Preferences → Code Composer Studio → Advanced Tools → EnergyTrace™ Technology**?
- Is the "Enable Ultra Low Power debug / Debug LPMx.5" option enabled in **Project → Properties → Debug → Low Power Mode Settings** (required only when selecting EnergyTrace mode)?

- **Battery Selection** (see Figure 4-7): The window is used to select one of the available standard batteries or define a customized battery. The EnergyTrace will use the battery characteristics to calculate the estimated selected battery lifetime for the current application depending on the measured current consumption. Available standard batteries are CR2032, 2xAAA or 2xAA.

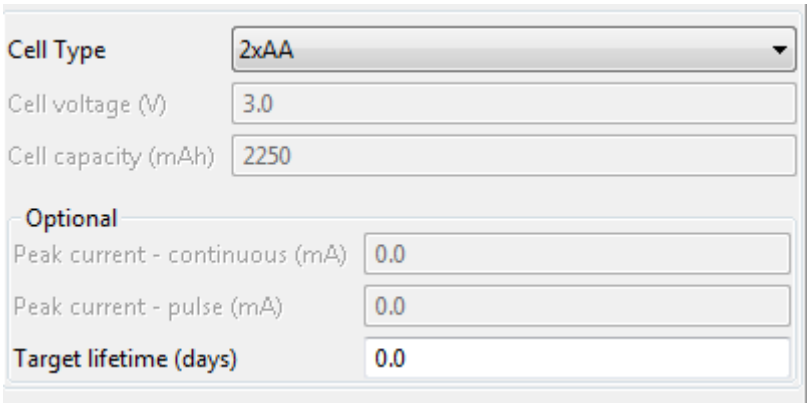

Figure 4-7. Battery Selection

A custom battery can also be selected, and its characteristics can be entered (see Figure 4-8).

- Cell voltage (V)
- Cell capacity (mAh)
- Peak current - continuous (mA)
- Peak current - pulse (mA)
- Target lifetime (days)

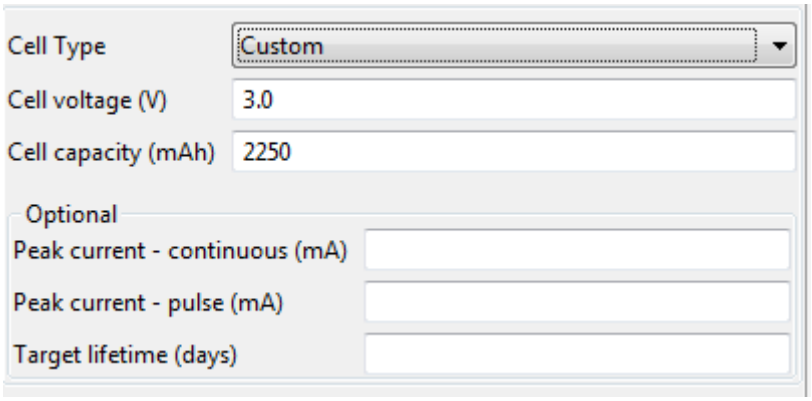

Figure 4-8. Custom Battery Type

- **Target Connection** (see Figure 4-9): The menu is used to select which debug probe is used for EnergyTrace measurement. The voltage can be also adjusted.

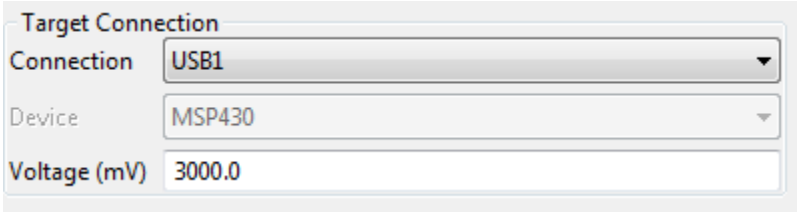

Figure 4-9. Target Connection

4.3.2 Controlling EnergyTrace Technology

EnergyTrace Technology can be controlled using the control bar icons in the **Profile** window (see Figure 4-10). Table 4-2 describes the function of each of these buttons.

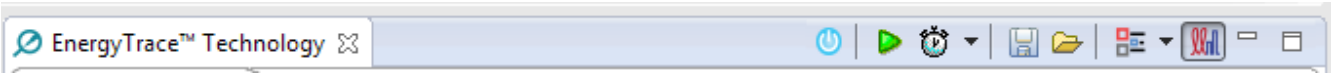

Figure 4-10. EnergyTrace™ Technology Control Bar

Table 4-2. EnergyTrace™ Technology Control Bar Icons

	Enable or disable EnergyTrace Technology. When disabled, icon turns gray.
	Starts trace collection.
	Stops trace collection.
	Set capture period: 5 sec, 10 sec, 30 sec, 1 min, or 5 min. Data collection stops after time has elapsed. However, the program continues to execute until the Pause button in the debug control window is clicked.
	Save profile to project directory. When saving an EnergyTrace++ profile, the default filename will start with "MSP430_D" followed by a timestamp. When saving an EnergyTrace profile, the default filename will start with "MSP430" followed by a timestamp.
	Load previously saved profile for comparison.
	Restore graphs or open Preferences window.
	Switch between **EnergyTrace++** mode and **EnergyTrace** mode

4.3.3 EnergyTrace++ Mode

When debugging devices with built-in EnergyTrace++ support, the **EnergyTrace++ mode** gives information about both energy consumption and the internal state of the target microcontroller. The following windows are opened during the start-up of a debug session (also see Figure 4-11):

- Profile
- States
- Power
- Energy

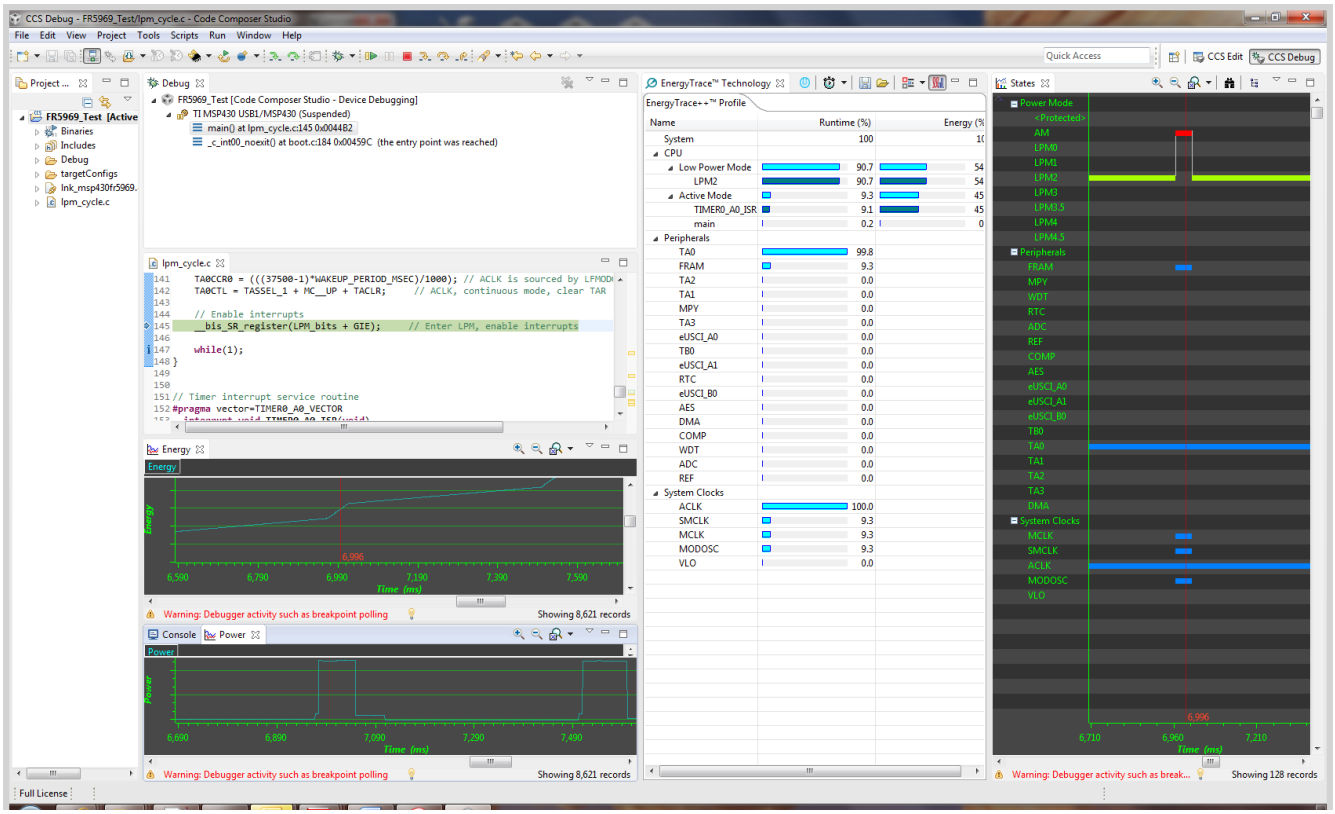

Figure 4-11. Debug Session With EnergyTrace++ Graphs

The **Profile** window (see Figure 4-12) is the control interface for EnergyTrace++. It can be used to set the capturing time or to save the captured data for later reference. The **Profile** window also displays a compressed view of the captured data and allows comparison with previous data.

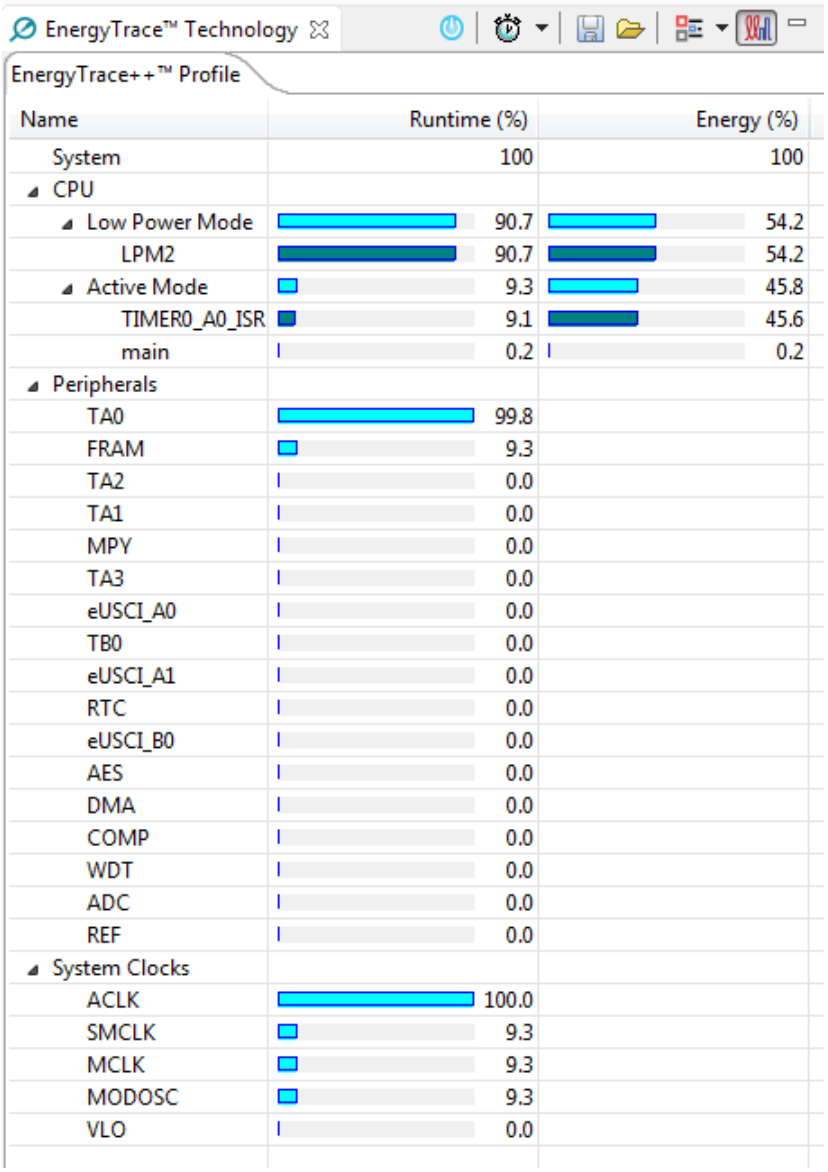

Figure 4-12. Profile Window

The **Profile** window enables a quick overview of the resource use of the profiled application. The resources are split into three categories:

- CPU: Shows information about program execution
  - Low Power Mode: Shows a summary of low-power mode use. Valid low-power modes are LPM0, LPM1, LPM2, LPM3, LPM4, LPM3.5, and LPM4.5. If the low-power mode cannot be properly determined, a line labeled as <Undetermined> is displayed to indicate the time spent in that mode.
  - Active Mode: Shows which functions have been executed during active mode. Functions in the run-time library are listed separately under the _RTS_ subcategory. If the device supports IP Encapsulation, a line labeled as <Protected> is displayed to indicate the time executing out of IP encapsulated memory.
- Peripherals: Shows relative on time of the device peripherals
- System Clocks: Shows relative on time of the system clocks

The **States** window (see Figure 4-13) shows the real-time trace of the target microcontroller's internal states during the captured session. State information includes the Power Modes, on and off state of peripheral modules and the state of the system clocks.

Figure 4-13 shows a device wakeup from low-power mode LPM2 to Active Mode, with the FRAM memory enabled during the active period. It can be clearly seen that the device high-speed clocks MCLK and SMCLK, as well as the MODOSC, are only active while the device is in active mode. The **States** window allows a direct verification of whether or not the application exhibits the expected behavior; for example, that a peripheral is disabled after a certain activity.

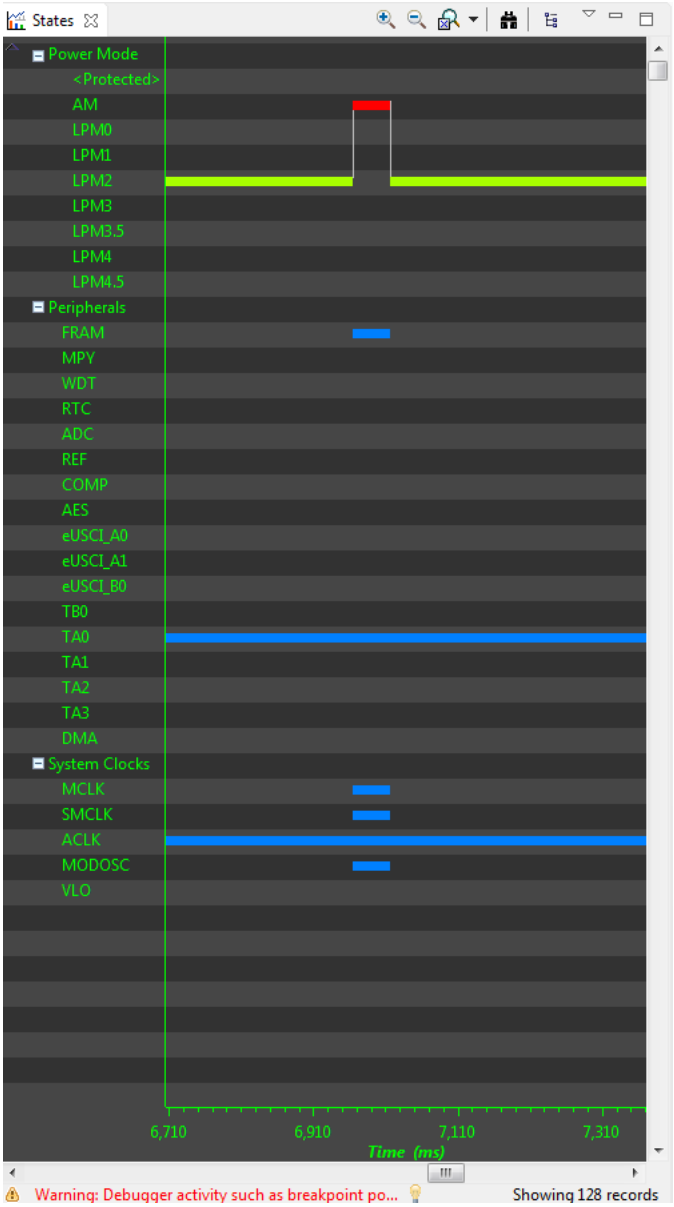

Figure 4-13. States Window

The **Power** window (see Figure 4-14) shows the dynamic power consumption of the target over time. The current profile is plotted in light blue color, while a previously recorded profile that has been reloaded for comparison is plotted in yellow color.

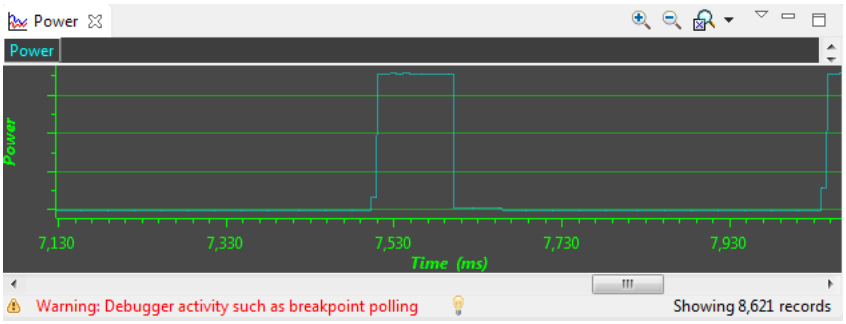

Figure 4-14. Power Window

The **Energy** window (see Figure 4-15) shows the accumulated energy consumption of the target over time. The current profile is plotted in light blue color, while a previously recorded profile that has been reloaded for comparison is plotted in yellow color.

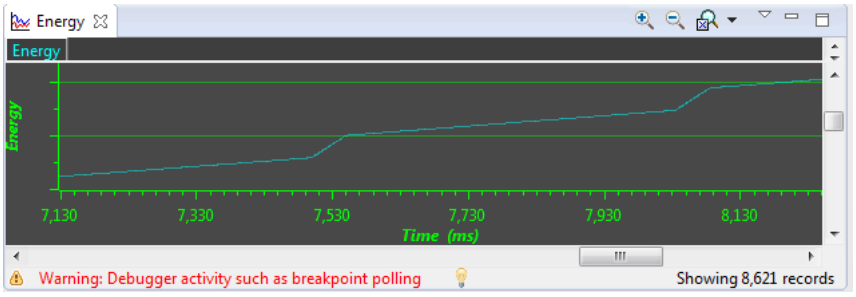

Figure 4-15. Energy Window

Note

During the capture of the internal states, the target microcontroller is constantly accessed by the JTAG or Spy-Bi-Wire debug logic. These debug accesses consume energy; therefore, no absolute power numbers are shown on the Power and Energy graph vertical axis. To see absolute power numbers of the application, TI recommends using the **EnergyTrace mode** in combination with the **Free Run** option. In this mode, the debug logic of the target microcontroller is not accessed while measuring energy consumption.

4.3.4 EnergyTrace Mode

This mode allows a stand-alone use of the energy measurement feature with MSP430 microcontrollers that do not have built-in EnergyTrace++ support. It can also be used to verify the energy consumption of the application without debugger activity. If the **EnergyTrace** mode is selected in the Preferences window, the following windows open when a debug session starts (also see Figure 4-16):

- Profile
- Power
- Energy

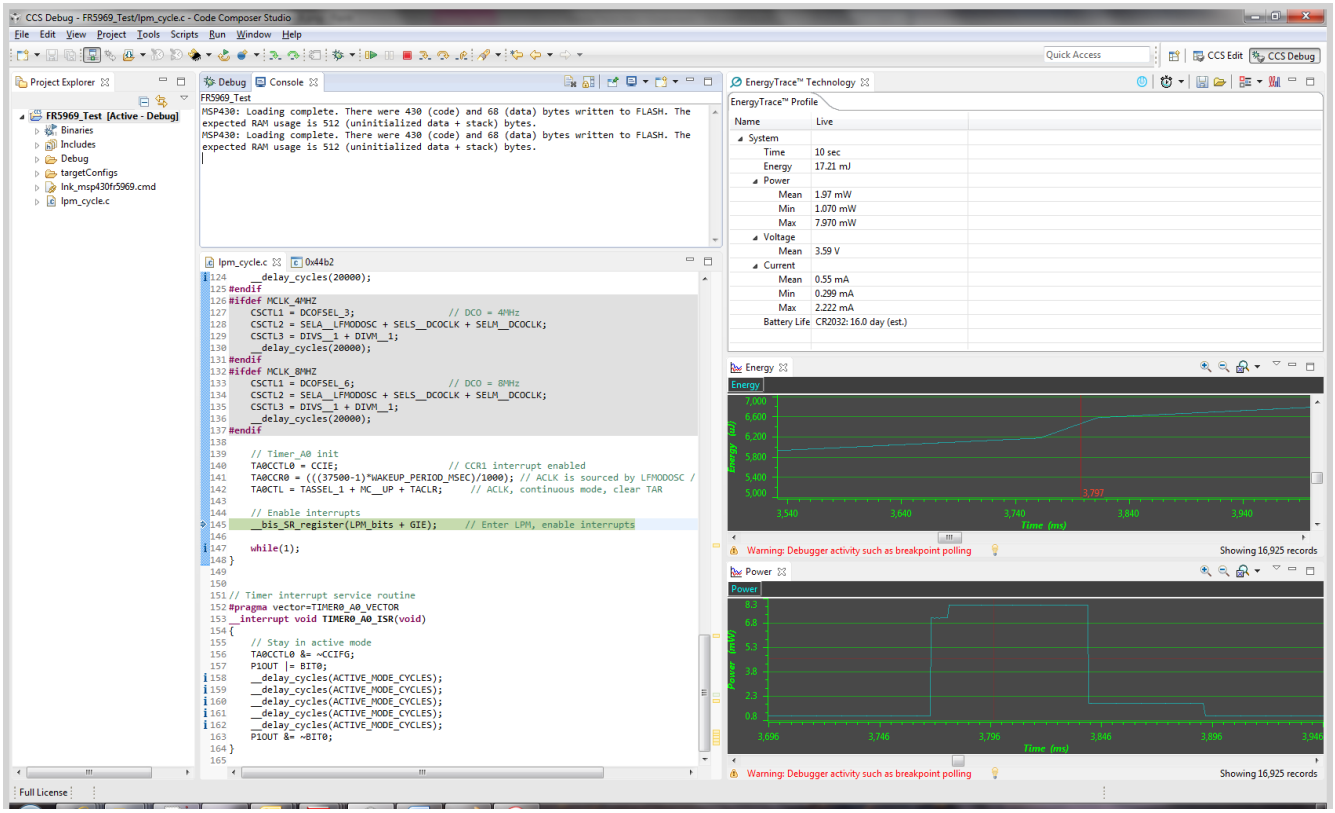

Figure 4-16. Debug Session With EnergyTrace Graphs

In the **EnergyTrace** mode, the **Profile** window shows statistical data about the application that has been profiled (see Figure 4-17). The following parameters are shown:

- Captured time
- Total energy consumed by the application (in mJ)
- Minimum, mean, and maximum power (in mW)
- Mean voltage (in V)
- Minimum, mean, and maximum current (in mA)
- Estimated life time of the selected battery (in days) for the captured energy profile

Note

The formula to calculate the battery life time assumes an ideal 3-V battery and does not account for temperature, aging, peak current, and other factors that could negatively affect battery capacity. It should also be noted that changing the target voltage (for example, from 3.6 V to 3 V) might cause the analog circuitry to behave differently and operate in a more or less efficient state, hence reducing or increasing energy consumption. The value shown in the Profile window cannot substitute measurements on real hardware.

EnergyTrace™ Profile

Name	Live
System	
Time	10 sec
Energy	14.61 mJ
Power	
Mean	1.75 mW
Min	1.068 mW
Max	7.943 mW
Voltage	
Mean	3.59 V
Current	
Mean	0.49 mA
Min	0.298 mA
Max	2.213 mA
Battery Life	CR2032: 18.8 day (est.)

Figure 4-17. EnergyTrace Profile Window

The **Power** window (see Figure 4-18) shows the dynamic power consumption of the target over time. The current profile is plotted in light blue color, while a previously recorded profile that has been reloaded for comparison is plotted in yellow color.

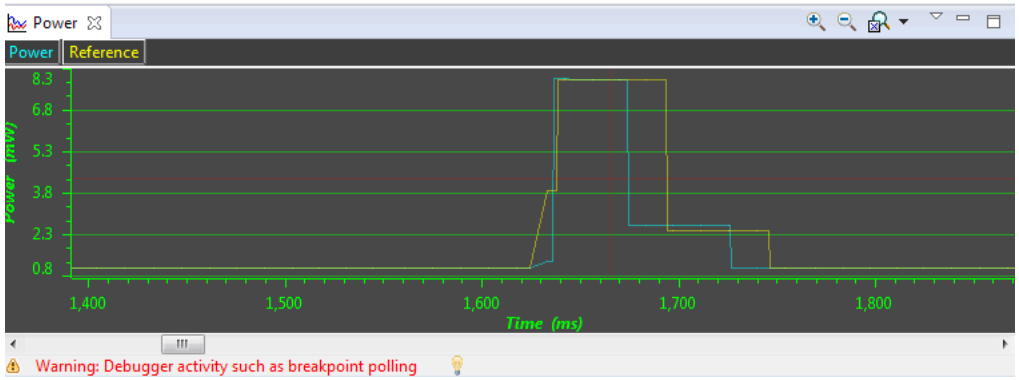

Figure 4-18. Zoom Into Power Window

The **Energy** window (see Figure 4-19) shows the accumulated energy consumption of the target over time. The current profile is plotted in light blue color, while a previously recorded profile that has been reloaded for comparison is plotted in yellow color.

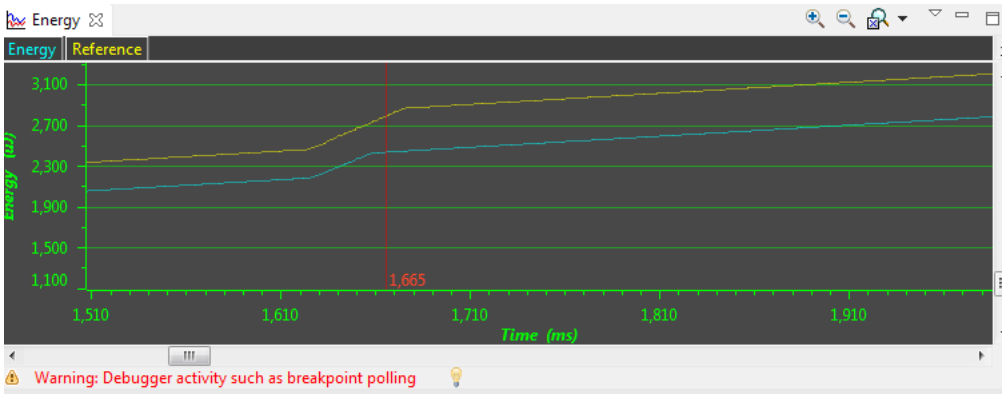

Figure 4-19. Zoom Into Energy Window

Note

During program execution through the debugger's view **Resume** button, the target microcontroller is constantly accessed by the JTAG or Spy-Bi-Wire protocol to detect when a breakpoint has been hit. Inevitably, these debug accesses consume energy in the target domain and change the result shown in both Energy and Power graphs. To see the absolute power consumption of an application, TI recommends using the **Free Run** mode. In **Free Run** mode, the debug logic of the target microcontroller is not accessed. See Figure 4-20 for an example of the effect of energy consumption coming from debug accesses. The yellow profile was recorded in **Resume** mode, and the green profile was recorded in **Free Run** mode.

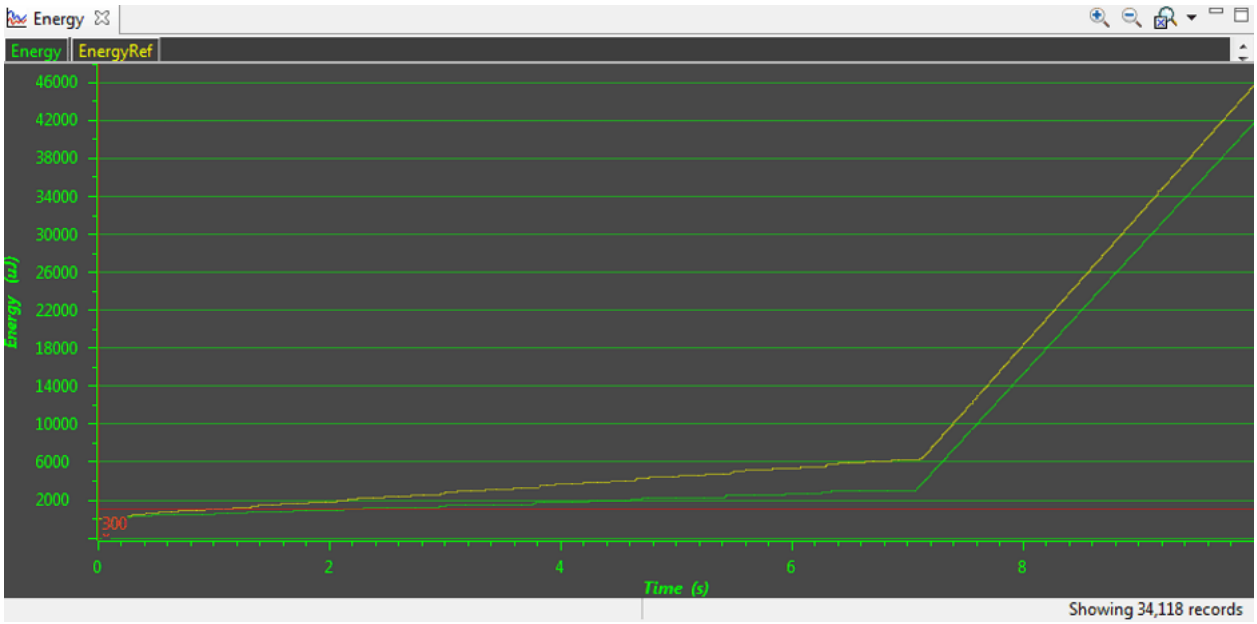

Figure 4-20. Energy Profile of the Same Program in Resume (Yellow Line) and Free Run (Green Line)

4.3.5 Comparing Captured Data With Reference Data

The EnergyTrace Technology can be used in various ways. One is to check the device's internal states over time against the expected behavior and correct any misbehavior; for example, due to a peripheral not being disabled after periodic usage. Another way is to compare the captured data against previously captured data. The previously captured data is called the *reference data* in the following discussion.

After the reference data has been loaded, a yellow reference graph is plotted in the Power and Energy windows. The Power window shows the power profiles of both data sets over time and is useful to determine any changes in static power consumption; for example, due to use of a deeper low-power mode or disabling of unused peripherals. It also shows how the dynamic power consumption has changed from one measurement to the other; for example, due to ULP Advisor hints being implemented. The Energy window shows the accumulated energy consumption over time and gives an indication which profile is more energy efficient.

In the **EnergyTrace++** mode, the condensed view of both captured and reference data is displayed in the Profile window (see Figure 4-21). You can quickly see how the overall energy consumption and use of power modes, peripherals, and clocks changed between both capture sessions. In general, parameters that have become better are shown with a green bar, and parameters that have become worse are shown with a red bar. For example, time spent in Active Mode is generally seen as negative. Hence, if a code change makes the application spend less time in active mode, the negative delta is shown as a green bar, and the additional time spent in a low-power mode is shown as a green bar.

EnergyTrace™ Technology

Reference: MSP430_D_2014_02_27_115349

EnergyTrace++™ Profile

Name	Runtime (%)	Energy (%)	Delta Runtime (%)	Delta Energy (%)	Reference Runtime (%)	Reference Energy (%)
System	100	83.5	0	-16.5	100	100
CPU						
Low Power Mode	94.3	67.2	3.5	12.9	90.8	54.3
LPM2	94.3	67.2	3.5	12.9	90.8	54.3
LPM0	0.0	0.0	-0.0	-0.0	0.0	0.0
Active Mode	5.7	32.8	-3.5	-12.9	9.2	45.7
TIMER0_A0_ISR	5.5	32.6	-3.6	-13.0	9.1	45.7
main	0.2	0.2	0.1	0.1	0.1	0.1
Peripherals						
TA0	99.8		-0.1		99.9	
FRAM	5.7		-3.5		9.2	
TA2	0.0		0.0		0.0	
TA1	0.0		0.0		0.0	
MPY	0.0		0.0		0.0	
TA3	0.0		0.0		0.0	
eUSCI_A0	0.0		0.0		0.0	
TB0	0.0		0.0		0.0	
eUSCI_A1	0.0		0.0		0.0	
RTC	0.0		0.0		0.0	
eUSCI_B0	0.0		0.0		0.0	
AES	0.0		0.0		0.0	
DMA	0.0		0.0		0.0	
COMP	0.0		0.0		0.0	
WDT	0.0		0.0		0.0	
ADC	0.0		0.0		0.0	
REF	0.0		0.0		0.0	
System Clocks						
ACLK	100.0		-0.0		100.0	
SMCLK	5.7		-3.5		9.2	
MCLK	5.7		-3.5		9.2	
MODOSC	5.7		-3.5		9.2	
VLO	0.0		0.0		0.0	

Figure 4-21. Comparing Profiles in EnergyTrace++ Mode

In the **EnergyTrace mode**, no States information is available to generate an exhaustive report. However, the overall energy consumed during the measurement is compared and, with it, the Min, Mean, and Max values of power and current. Parameters that have become better are shown with a green bar, and parameters that have become worse are shown with a red bar (see Figure 4-22).

EnergyTrace™

Reference: MSP430_2014_01_21_103959

EnergyMonitor Profile

Name	Live	Delta (%)	Reference
System			
Time	10 sec		10 sec
Energy	7.84 mJ	-11.5	8.86 mJ
Power			
Mean	0.79 mW	-10.8	0.89 mW
Min	0.582 mW	-27.3	0.800 mW
Max	1.938 mW	-0.2	1.942 mW
Voltage			
Mean	2.99 V	0.0	2.99 V
Current			
Mean	0.26 mA	-10.8	0.30 mA
Min	0.195 mA	-27.0	0.268 mA
Max	0.646 mA	-0.4	0.648 mA
Battery Life	CR2032: 35.1 day (est.)	13.0	CR2032: 31.0 day (est.)

Figure 4-22. Comparing Profiles in EnergyTrace Mode

The delta bars are drawn linearly from 0% to 50%. Deltas larger than 50% do not result in a larger delta bar.

4.4 EnergyTrace Technology FAQs

Q: What is the sampling frequency of EnergyTrace++ technology?

A: The sampling frequency depends on the debugger and the selected debug protocol and its speed setting. It typically ranges from 1 kHz (for example, when using the Spy-Bi-Wire interface set to SLOW) up to 3.2 kHz (for example, when using the JTAG interface set to FAST). The debugger polls the state information of EnergyTrace ++ from the device status information. Depending on the sampling frequency, a short or fast duty cycle active peripheral state may not be captured on the State graph. In addition, the higher sampling frequency affects the device energy consumption under EnergyTrace.

Q: What is the sampling frequency of EnergyTrace technology?

A: The sampling frequency to measure the energy consumption is the same independent of which debug protocol or speed and is approximately 4.2 kHz in Free Run mode.

Q: My Power graph seems to include noise. Is my board defective?

A: The power values shown in the Power graph are derived (that is, calculated) from the accumulated energy counted by the measurement system. When the target is consuming little energy, a small number of energy packets over time are supplied to the target, and the software needs to accumulate the dc-dc charge pulses over time before a new current value can be calculated. For currents under 1 µA, this can take up to one second, while for currents in the milliamp range, a current can be calculated every millisecond. Additional filtering is not applied so that detail information is not lost. Another factor that affects the energy (and with it, the current) that is consumed by the target is periodic background debug access during normal code execution, either through capturing of States information or through breakpoint polling. Try recording in Free Run mode to see a much smoother Power graph.

Q: I have a code that repeatedly calls functions that have the same size. I would expect the function profile to show an equal distribution of the run time. In reality, I see some functions having slightly more run time than expected, and some functions slightly less.

A: During program counter trace, various factors affect the number of times a function is detected by the profiler over time. The microcontroller code could benefit from the internal cache, thus executing some functions faster than others. Another influencing factor is memory wait states and CPU pipeline stalls, which add time variance to the code execution. An outside factor is the sampling frequency of the debugger itself, which normally runs asynchronous to the microcontroller's code execution speed, but in some cases shows overlapping behavior, which also results in an unequal function run time distribution.

Q: My power mode profile sometimes shows short periods of power modes that I haven't used anywhere in my code. For example, I'm expecting a transition from active mode to LPM3, but I see a LPM2 during the transition.

A: When capturing in EnergyTrace++ mode, digital information is continuously collected from the target device. One piece of this information is the power mode control signals. Activation of low-power modes requires stepping through a number of intermediate states. Usually this happens too quickly to be captured by the trace function, but sometimes intermediate states can be captured and are displayed for a short period of time as valid low-power modes.

Q: My profile sometimes includes an <Undetermined> low-power mode, and there are gaps in the States graph Power Mode section. Where does the <Undetermined> low-power mode originate from?

A: During transitions from active mode to low-power mode, internal device clocks are switched off, and occasionally the state information is not updated completely. This state is displayed as <Undetermined> in the Profile window, and the States graph shows a gap during the time that the <Undetermined> low-power mode persists. The <Undetermined> state is an indication that your application has entered a low-power mode, but which mode cannot be accurately determined. If your application is frequently entering low-power modes, the <Undetermined> state will probably be shown more often than if your application only rarely uses low-power modes.

Q: When capturing in EnergyTrace mode, the min and max values for power and current show deviation, even though my program is the same. I would expect absolutely the same values.

A: The energy measurement method used on the hardware counts dc-dc charge pulses over time. Energy and power are calculated from the energy over time. Due to statistical sampling effects and charge and discharge effects of the output voltage buffer capacitors, it is possible that minimum and maximum values of currents vary by some percent, even though the program is identical. The captured energy, however, should be almost equal (in the given accuracy range).

Q: What are the influencing factors for the accuracy of the energy measurement?

A: The energy measurement circuit is directly supplied from the USB bus voltage, and thus it is sensitive to USB bus voltage variations. During calibration, the energy equivalent of a single dc-dc charge pulse is defined, and this energy equivalent depends on the USB voltage level. To ensure a good repeatability and accuracy, power the debugger directly from an active USB port, and avoid using bus-powered hubs and long USB cables that can lead to voltage drops, especially when other consumers are connected to the USB hub. Furthermore the LDO and resistors used for reference voltage generation and those in the calibration circuit come with a certain tolerance and ppm rate over temperature, which also influences accuracy of the energy measurement.

Q: I am trying to capture in EnergyTrace++ mode or EnergyTrace mode with a MSP430 device that is externally powered, but there is no data shown in the Profile, Energy, Power and States window.

A: Both EnergyTrace++ mode and EnergyTrace mode require the target to be supplied from the debugger. No data can be captured when the target microcontroller is externally powered.

Q: I cannot measure LPM currents when I am capturing in EnergyTrace++ mode. I am expecting a few microamps but measure more than 150 µA.

A: Reading digital data from the target microcontroller consumes energy in the JTAG domain of the microcontroller. Hence, an average current of approximately 150 µA is measured when connecting an ampere meter to the device power supply pins. If you want to eliminate energy consumption through debug communication, switch to EnergyTrace mode, and let the target microcontroller execute in Free Run mode.

Q: My LPM currents seem to be wrong. I am expecting a few microamps, but measure more, even in Free Run mode or when letting the device execute without debug control from an independent power supply.

A: The most likely cause of this extra current is improper GPIO termination, as floating pins can lead to extra current flow. Also check the JTAG pins again, especially when the debugger is still connected (but idle), as the debugger output signal levels in idle state might not match how the JTAG pins have been configured by the application code. This could also lead to extra current flow.

Q: When I start the EnergyTrace++ windows through View → Other → MSP430-EnergyTrace before launching the debug session, data capture sometimes does not start.

A: Enable EnergyTrace through **Window → Preferences → Code Composer Studio → Advanced Tools → EnergyTrace™ Technology**. When launching a debug session, the EnergyTrace++ windows automatically open, and data capture starts when the device executes. If you accidentally close all EnergyTrace++ windows during a debug session, you can reopen them through **View → Other → MSP430-EnergyTrace**.

5 MSP430 FRAM Memory Protection Mechanisms

The available memory of an FRAM-based microcontroller can be seen as unified memory, which means the memory can be arbitrarily divided between code and data sections. As a consequence, a single FRAM-based microcontroller can be customized for a wide range of application use cases. MSP430 devices support two memory protection methods:

- Memory Protection Unit (MPU) and Intellectual Property Encapsulation (IPE)
- FRAM Memory Write Protection (FRWP). The protection granularity (1k) can be configured on some devices.

See the device-specific data sheet to determine which method a particular device supports. For instructions on the efficient use of this technology, see MSP430™ FRAM Technology – How To and Best Practices.

5.1 Memory Protection Unit (MPU)

To prevent accidental overwrites of the program by application data or other forms of data corruption, the Memory Protection Unit allows partitioning of the available memory and defining access rights for each of the partitions. Thus it is possible to prevent accidental writes to memory sections that contain application code or prevent the microcontroller from executing instructions that are located in the data section of the application.

Figure 5-1 shows the MPU configuration dialog, which is available for FRAM devices that have the MPU feature. To access this dialog, select the menu **Project** → **Properties** → **General** → **MPU**. This dialog lets you enable or disable the MPU and choose between an automatic and manual configuration mode. For the automatic configuration, the compiler tool chain generates two memory segments (read-write memory and executable memory). The segment borders of these two segments and their respective access bits are placed into the according control registers during device start-up. The automatic mode also sets the bit for read access of the MPU Info Memory segment. The MPUSEGxVS bit, which selects if a PUC must be executed on illegal access to a segment, is also set by default for each of the segments.

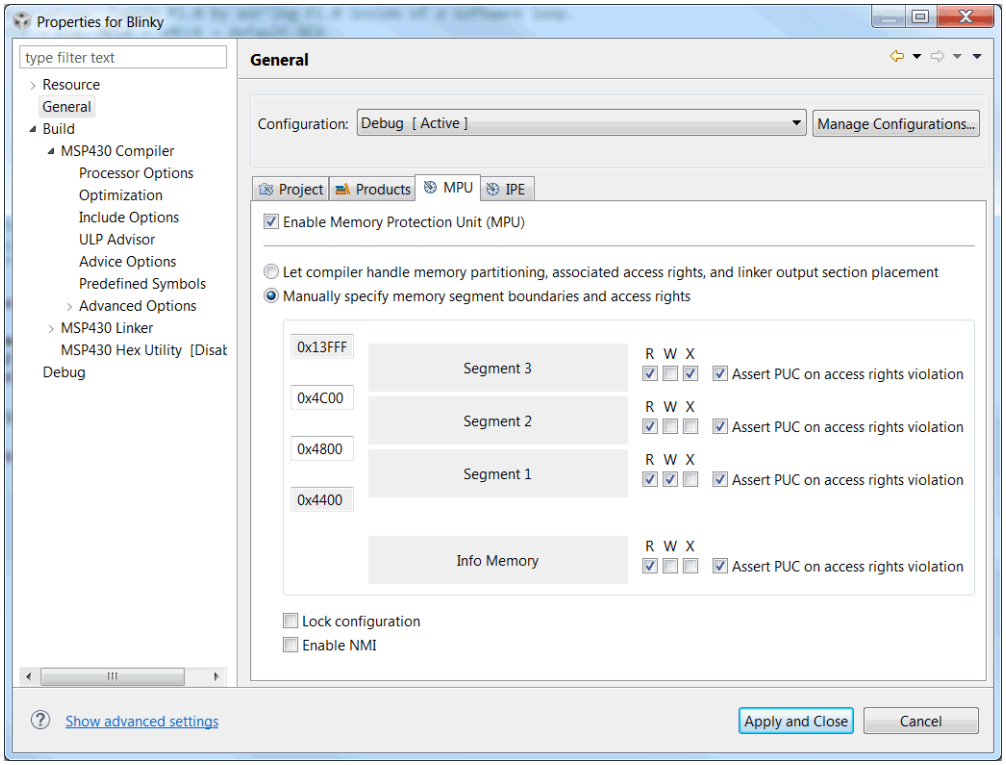

Figure 5-1. MPU Configuration Dialog

As shown in Figure 5-1, the MPU dialog also allows for a complete manual configuration of the Memory Protected Area. As the beginning of Segment 1 is fixed to the start address of FRAM memory and the end of Segment 3 is fixed to the end address of FRAM memory, only the start and end addresses of Segment 2 need to be adjusted. As these addresses are equal to the end address and start addresses of Segment 1 and Segment 3 respectively, these are adjusted automatically by the GUI. The memory and its associated access rights can be configured completely independently in the manual configuration. It is therefore the user's responsibility to place code and data segments into the correct memory locations. Additional configuration of the linking process might be necessary to achieve the correct placement of code and data in the desired memory locations.

5.2 Intellectual Property Encapsulation (IPE)

The memory of many microcontroller applications contains information that should not be accessible by the public. This may include both the application code itself as well as configuration settings for certain peripherals. The IPE module allows the protection of memory that contains this kind of sensitive information. The IPE ensures that only program code that is itself placed in the IPE protected area has access to this memory segment. The access rights are evaluated with each code access, and even JTAG or DMA transfers cannot access the IPE segment. The IPE module is initialized by the boot code before the start of the application code to ensure that the encapsulation is active before any user-controlled access to the memory can be performed.

Figure 5-2 shows the dialog for configuration of IPE memory, which is accessible through the menu **Project** → **Properties** → **General** → **IPE**. The IPE dialog also provides selections for manual and automatic configuration. In the automatic mode, a memory segment ".ipe" is generated by the compiler tool chain and placed in the output file. Placing a variable into this section can be performed directly from the source code:

```
#pragma DATA_SECTION(primeNumbers, ".ipe")
const unsigned int primeNumbers[5] = {2, 3, 5, 7, 11};
```

For a more detailed description on how to allocate space for certain code or data symbols inside sections, see the MSP430™ Optimizing C/C++ Compiler User's Guide.

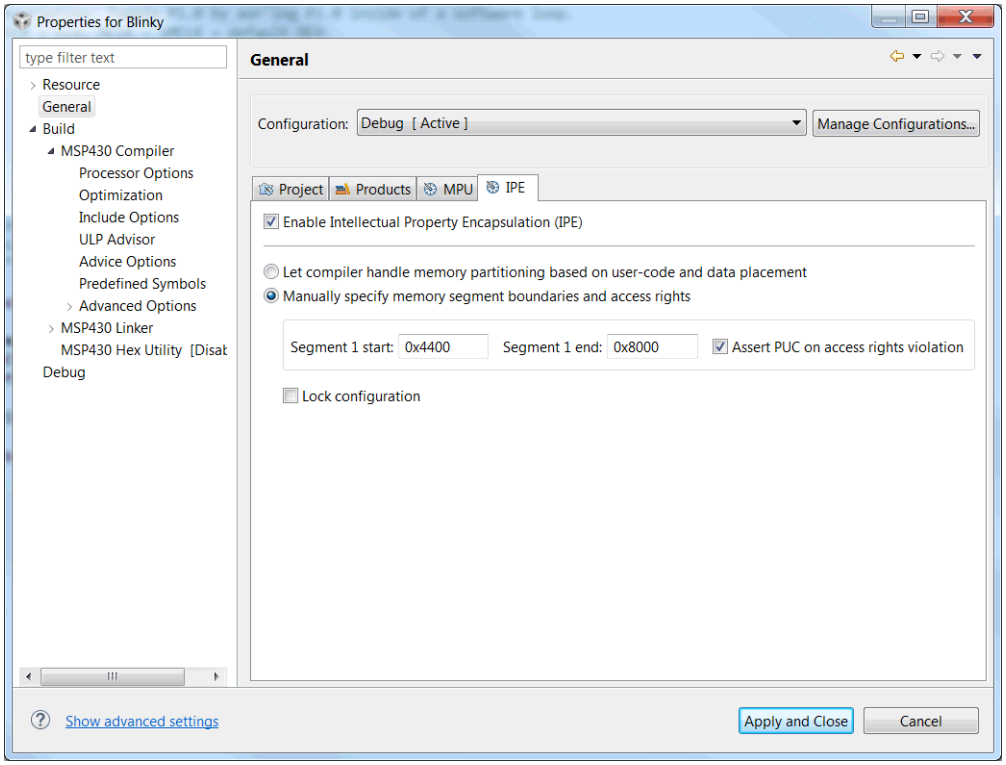

Figure 5-2. IPE Configuration Dialog

The Manual IPE mode lets you configure the IPE segment borders and the control settings. Consequently, additional configuration of the compiler or linker stage may also necessary to achieve the correct placement of code and data in memory. To prevent the IPE from being modified, place the section ".ipestruct" inside the IP encapsulated memory area. This section contains the section borders and control settings that are used to initialize the IPE related registers during device start-up.

5.2.1 IPE Debug Settings

Because it is possible to lock out the debugger from accessing certain memory regions (including downloading new software to the device), it is advisable to enable the option for erasing the IP protected area while the target is under debugger control. The corresponding option can be found under **Project Properties → Debug → MSP430 Properties** (see Figure 5-3).

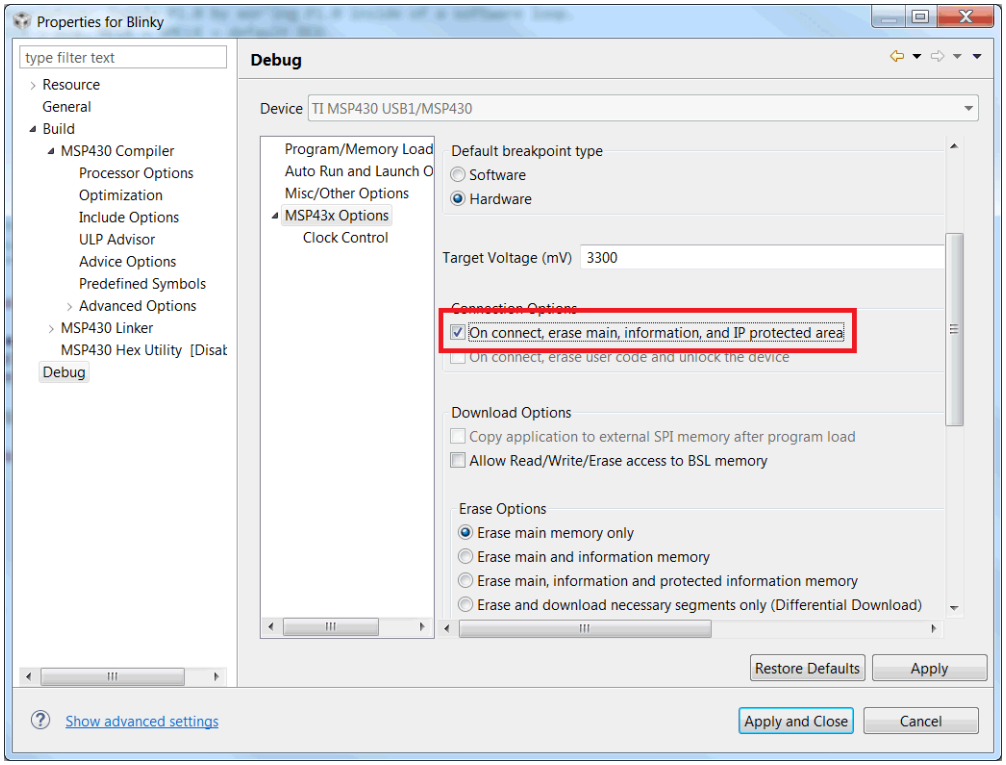

Figure 5-3. IPE Debug Settings

5.3 FRAM Write Protection (FRWP)

The FRWP prevents unintended programming of the FRAM code section. For MSP430FR2xx and MSP430FR4xx MCUs, the FRAM memory is protected by setting the bits control in SYSCFG0 register. Some MSP430 devices can protect and unprotect the whole memory at once, and some devices such as MSP430FR2355, MSP430FR2353, MSP430FR2155 and MSP430FR2153 can unprotect some region and protect the rest of the memory.

CCS 8.1 and newer versions provide a GUI to configure the FRAM write protection. By default, a new CCS project has the Enable FRAM Write Protection (FRWP) option selected.

When the Enable FRAM Write Protection (FRWP) option is selected, you can protect or unprotect the information memory. When the application code uses persistent data type, the size of persistent data is automatically calculated and aligned with 1kB size. These data are then placed in the unprotected program main memory section. The code is placed after the unprotected program main memory.

Figure 5-4 shows the FRWP configuration dialog, which is available for FRAM devices that have the feature. To access this dialog, select the menu **Project** → **Properties** → **General** → **FRWP**.

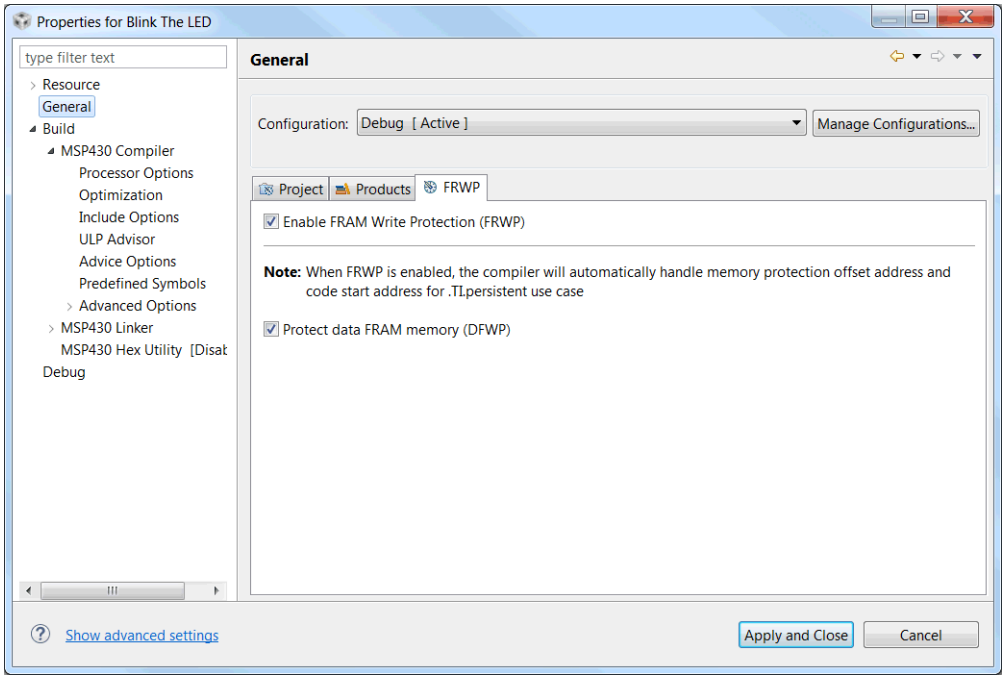

Figure 5-4. FRWP Configuration Dialog

6 Frequently Asked Questions

This appendix presents solutions to frequently asked questions regarding hardware, program development and debugging tools.

6.1 Hardware

For a complete list of hardware related FAQs, see the *MSP430 Hardware Tools User's Guide*.

6.2 Program Development (Assembler, C-Compiler, Linker, IDE)

Note

Consider the CCS Release Notes

For the case of unexpected behavior, see the CCS Release Notes document for known bugs and limitations of the current CCS version. This information can be accessed through the menu item **Start → All Programs → Texas Instruments → Code Composer Studio → Release Notes**.

1. **A common MSP430 "mistake" is to fail to disable the watchdog mechanism.** The watchdog is enabled by default, and it resets the device if not disabled or properly managed by the application. Use `WDTCL = WDTPW + WDTHOLD;` to explicitly disable the Watchdog. This statement is best placed in the _system_pre_init() function that is executed prior to main(). If the Watchdog timer is not disabled, and the Watchdog triggers and resets the device during CSTARTUP, **the source screen goes blank**, as the debugger is not able to locate the source code for CSTARTUP. Be aware that CSTARTUP can take a significant amount of time to execute if a large number of initialized global variables are used.

```
int _system_pre_init(void)
{
/* Insert your low-level initializations here */
WDTCTL = WDTPW + WDTHOLD; // Stop Watchdog timer
/*===================================*/
/* Choose if segment initialization */
/* should be done or not.           */
/* Return: 0 to omit initialization */
/*         1 to run initialization  */
/*===================================*/
return (1);
}
```

2. **Within the C libraries, GIE (Global Interrupt Enable) is disabled before** (and restored after) **the hardware multiplier is used.**
3. **It is possible to mix assembly and C programs within CCS.** See the "Interfacing C/C++ With Assembly Language" chapter of the *MSP430 Optimizing C/C++ Compiler User's Guide*.
4. **Constant definitions (#define) used within the .h files are effectively reserved** and include, for example, C, Z, N, and V. Do not create program variables with these names.
5. **Compiler optimization can remove unused variables and statements that have no effect** and can affect debugging. To prevent this, these variables can be declared `volatile` ; for example:

```
volatile int i;.
```

Note

The Tools Insider blog gives useful tips and tricks about the TI MSP430 compiler and linker. For details, visit the Tools Insider blog and read the *From the Experts* series.

Some useful posts:

From the Experts: Executing code from RAM using TI compilers

From the Experts: Accessing files and libraries from a linker command file (LCF)

6.3 Debugging

The debugger is part of CCS and can be used as a stand-alone application. This section is applicable when using the debugger both stand-alone and from the CCS IDE.

Note

Consider the CCS release notes

In case of unexpected behavior, see the CCS Release Notes document for known bugs and limitations of the current CCS version. To access this information, click **Start → All Programs → Texas Instruments → Code Composer Studio → Release Notes**.

1. **The debugger reports that it cannot communicate with the device.** Possible solutions to this problem include:
   - Make sure that the correct debug interface and corresponding port number have been selected in **Project → Properties → General → Device → Connection**.
   - Make sure that the jumper settings are configured correctly on the target hardware.
   - Make sure that no other software application (for example, a printer driver) has reserved or taken control of the COM or parallel port, which would prevent the debug server from communicating with the device.
   - Open the Device Manager and determine if the driver for the FET tool has been correctly installed and if the COM or parallel port is successfully recognized by the Windows OS. Check the PC BIOS for the parallel port settings (see FAQ 5). For users of IBM or Lenovo ThinkPad® computers, try port setting LPT2 and LPT3, even if operating system reports that the parallel port is located at LPT1.
   - Restart the computer.

   Make sure that the MSP430 device is securely seated in the socket (so that the "fingers" of the socket completely engage the pins of the device), and that its pin 1 (indicated with a circular indentation on the top surface) aligns with the "1" mark on the PCB.

   CAUTION

   Possible Damage To Device

   Always handle MSP430 devices with a vacuum pick-up tool only; do not use your fingers, as you can easily bend the device pins and render the device useless. Also, always observe and follow proper ESD precautions.
2. **The debugger can debug applications that use interrupts and low-power modes.** See FAQ 17).
3. **The debugger cannot access the device registers and memory while the device is running.** The user must stop the device to access device registers and memory.

4. **The debugger reports that the device JTAG security fuse is blown**. With current MSP430-FET430UIF JTAG interface tools, there is a weakness when adapting target boards that are powered externally. This leads to an accidental fuse check in the MSP430 and results in the JTAG security fuse being recognized as blown although it is not.

   Workarounds:
   - Connect the device $\overline{\text{RST}}$/NMI pin to JTAG header (pin 11), MSP-FET430UIF interface tools are able to pull the $\overline{\text{RST}}$ line, this also resets the device internal fuse logic.
   - Do not connect both V_{CC} Tool (pin 2) and V_{CC} Target (pin 4) of the JTAG header. Specify a value for V_{CC} in the debugger that is equal to the external supply voltage.

   Note

   When the V_{CC} voltage is not high enough when trying to erase or write flash memory, the following message displays in the console: "Target device supply voltage is too low for Flash erase/ programming." If this occurs, try to change your supply voltage.

5. **The parallel port designators (LPTx) have the following physical addresses: LPT1 = 378h, LPT2 = 278h, LPT3 = 3BCh.** The configuration of the parallel port (ECP, Compatible, Bidirectional, Normal) is not significant; ECP seems to work well. See FAQ 1 for additional hints on solving communication problems between the debugger and the device.
6. **The debugger asserts $\overline{\text{RST}}$/NMI to reset the device** when the debugger is started and when the device is programmed. The device is also reset by the debugger Reset button, and when the device is manually reprogrammed (using Reload), and when the JTAG is resynchronized (using Resynchronize JTAG). When $\overline{\text{RST}}$/NMI is not asserted (low), the debugger sets the logic driving $\overline{\text{RST}}$/NMI to high impedance, and $\overline{\text{RST}}$/NMI is pulled high through a resistor on the PCB.

   The $\overline{\text{RST}}$/NMI signal is asserted and negated after power is applied when the debugger is started. $\overline{\text{RST}}$/NMI is then asserted and negated a second time after device initialization is complete.
7. **The debugger can debug a device whose program reconfigures the function of the $\overline{\text{RST}}$/NMI pin to NMI.**
8. **The level of the XOUT/TCLK pin is undefined when the debugger resets the device.** The logic driving XOUT/TCLK is set to high impedance at all other times.
9. **When making current measurements of the device, ensure that the JTAG control signals are released**, otherwise the device is powered by the signals on the JTAG pins and the measurements are erroneous. See FAQ 10.
10. **When the debugger has control of the device, the CPU is on** (that is, it is not in low-power mode) regardless of the settings of the low-power mode bits in the status register. Any low-power mode condition is restored prior to STEP or GO. Consequently, do not measure the power consumed by the device while the debugger has control of the device. Instead, run the application using Release JTAG on run.
11. The MEMORY window correctly displays the contents of memory where it is present. However, **the MEMORY window incorrectly displays the contents of memory where there is none present.** Memory should be used only in the address ranges as specified by the device data sheet.
12. The debugger uses the system clock to control the device during debugging. Therefore, **device counters and other components that are clocked by the Main System Clock (MCLK) are affected when the debugger has control of the device.** Special precautions are taken to minimize the effect upon the watchdog timer. The CPU core registers are preserved. All other clock sources (SMCLK and ACLK) and peripherals continue to operate normally during emulation. In other words, **the Flash Emulation Tool is a partially intrusive tool.**

    Devices that support clock control can further minimize these effects by stopping the clock(s) during debugging (**Project → Properties → CCS Debug Settings → Target → Clock Control**).

13. When programming the flash, **do not set a breakpoint on the instruction immediately following the write to flash operation.** A simple work-around to this limitation is to follow the write to flash operation with a NOP and to set a breakpoint on the instruction following the NOP.
14. Multiple internal machine cycles are required to clear and program the flash memory. **When single stepping over instructions that manipulate the flash,** control is given back to the debugger before these operations are complete. Consequently, **the debugger updates its memory window with erroneous information.** A workaround for this behavior is to follow the flash access instruction with a NOP and then step past the NOP before reviewing the effects of the flash access instruction.
15. **Bits that are cleared when read during normal program execution** (that is, interrupt flags) **are cleared when read while being debugged** (that is, memory dump, peripheral registers).

    Using certain MSP430 devices with enhanced emulation logic such as MSP430F43x and MSP430F44x devices, bits do not behave this way (that is, the bits are not cleared by the debugger read operations).
16. **The debugger cannot be used to debug programs that execute in the RAM of F12x and F41x devices.** A workaround for this limitation is to debug programs in flash.
17. **While single stepping with active and enabled interrupts, it can appear that only the interrupt service routine (ISR) is active** (that is, the non-ISR code never appears to execute, and the single step operation stops on the first line of the ISR). However, this behavior is correct because the device processes an active and enabled interrupt before processing non-ISR (that is, mainline) code. A workaround for this behavior is, while within the ISR, to disable the GIE bit on the stack, so that interrupts are disabled after exiting the ISR. This permits the non-ISR code to be debugged (but without interrupts). Interrupts can later be re-enabled by setting GIE in the status register in the Register window.

    On devices with Clock Control, it may be possible to suspend a clock between single steps and delay an interrupt request (**Project** → **Properties** → **CCS Debug Settings** → **Target** → **Clock Control**).
18. On devices equipped with a Data Transfer Controller (DTC), **the completion of a data transfer cycle preempts a single step of a low-power mode instruction.** The device advances beyond the low-power mode instruction only after an interrupt is processed. Until an interrupt is processed, it appears that the single step has no effect. A workaround to this situation is to set a breakpoint on the instruction following the low-power mode instruction, and then execute (Run) to this breakpoint.
19. **The transfer of data by the Data Transfer Controller (DTC) may not stop precisely when the DTC is stopped in response to a single step or a breakpoint.** When the DTC is enabled and a single step is performed, one or more bytes of data can be transferred. When the DTC is enabled and configured for two-block transfer mode, the DTC may not stop precisely on a block boundary when stopped in response to a single step or a breakpoint.
20. **Breakpoints.** CCS supports a number of predefined breakpoint and watchpoint types. See Section 3.2.2 for a detailed overview.

Note

Linux and OS X do not support the MSP-FET430UIF if it has an old firmware image (MSP Debug Stack v2) on it.

Customers who buy a new MSP-FET430UIF will encounter this issue on OS X or Linux, because the MSP Debug Stack v2 is programmed on the debugger during production. To resolve this problem, connect the debugger to a Windows PC and use IAR, CCS, or the MSP430 Flasher to update the firmware on the debugger to the latest version (v3 or newer).

Note

Do not connect through a USB hub when performing a firmware update on the MSP-FET, the MSP-FET430UIF, or a LaunchPad™ development kit.

7 Migration of C Code from IAR 2.x, 3.x, 4.x, 5.x, 6.x or 7.x to CCS

Source code for the TI CCS C compiler and source code for the IAR Embedded Workbench C compiler are not fully compatible. Standard ANSI/ISO C code is portable between these tools, but implementation-specific extensions differ and must be ported. This appendix describes the major differences between the two compilers.

7.1 Interrupt Vector Definition

IAR ISR declarations (using the #pragma vector =) are now fully supported in CCS. However, this is not the case for all other IAR pragma directives.

7.2 Intrinsic Functions

CCS and IAR tools use the same instructions for MSP430 processor-specific intrinsic functions.

7.3 Data and Function Placement

7.3.1 Data Placement at an Absolute Location

The scheme implemented in the IAR compiler using either the @ operator or the #pragma location directive is not supported with the CCS compiler:

```
/* IAR C Code */
__no_init char alpha @ 0x0200;  /* Place 'alpha' at address 0x200 */
#pragma location = 0x0202
const int beta;
```

If absolute data placement is needed, this can be achieved with entries into the linker command file, and then declaring the variables as extern in the C code:

```
/* CCS Linker Command File Entry */
alpha = 0x200;
beta = 0x202;
/* CCS C Code */
extern char alpha;
extern int beta;
```

The absolute RAM locations must be excluded from the RAM segment; otherwise, their content may be overwritten as the linker dynamically allocates addresses. The start address and length of the RAM block must be modified within the linker command file. For the previous example, the RAM start address must be shifted 4 bytes from 0x0200 to 0x0204, which reduces the length from 0x0080 to 0x007C (for an MSP430 device with 128 bytes of RAM):

```
/* CCS Linker Command File Entry */
/************************************************************************/
/* SPECIFY THE SYSTEM MEMORY MAP */
/************************************************************************/
MEMORY /* assuming a device with 128 bytes of RAM */
{
...
RAM   :origin = 0x0204, length = 0x007C  /* was: origin = 0x200, length = 0x0080 */
...
}
```

The definitions of the peripheral register map in the linker command files (lnk_msp430xxxx.cmd) and the device-specific header files (msp430xxxx.h) that are supplied with CCS are an example of placing data at absolute locations.

Note

When a project is created, CCS copies the linker command file corresponding to the selected MSP430 derivative from the include directory (<Installation Root>\ccsv5\ccs_base\tools\compiler \MSP430\include) into the project directory. Therefore, ensure that all linker command file changes are done in the project directory. This allows the use of project-specific linker command files for different projects using the same device.

7.4 Data Placement Into Named Segments

In IAR, it is possible to place variables into named segments using either the @ operator or a #pragma directive:

```
/* IAR C Code */
__no_init int alpha @ "MYSEGMENT";  /* Place 'alpha' into 'MYSEGMENT' */
#pragma location="MYSEGMENT"        /* Place 'beta' into 'MYSEGMENT' */
const int beta;
```

With the CCS compiler, the #pragma DATA_SECTION() directive must be used:

```
/* CCS C Code */
#pragma DATA_SECTION(alpha, "MYSEGMENT")
int alpha;
#pragma DATA_SECTION(beta, "MYSEGMENT")
int beta;
```

See Section 7.7.3 for information on how to translate memory segment names between IAR and CCS.

7.5 Function Placement Into Named Segments

With the IAR compiler, functions can be placed into a named segment using the @ operator or the #pragma location directive:

```
/* IAR C Code */
void g(void) @ "MYSEGMENT"
{
}
#pragma location="MYSEGMENT"
void h(void)
{
}
```

With the CCS compiler, the following scheme with the #pragma CODE_SECTION() directive must be used:

```
/* CCS C Code */
#pragma CODE_SECTION(g, "MYSEGMENT")
void g(void)
{
}
```

See Section 7.7.3 for information on how to translate memory segment names between IAR and CCS.

7.6 C Calling Conventions

The CCS and IAR C-compilers use different calling conventions for passing parameters to functions. When porting a mixed C and assembly project to the TI CCS code generation tools, the assembly functions need to be modified to reflect these changes. For detailed information about the calling conventions, see the *TI MSP430 Optimizing C/C++ Compiler User's Guide* and the *IAR MSP430 C/C++ Compiler Reference Guide*.

The following example is a function that writes the 32-bit word `Data` to a given memory location in big-endian byte order. It can be seen that the parameter `Data` is passed using different CPU registers.

IAR Version:

```
;-------------------------------------------------------------------------------
; void WriteDWBE(unsigned char *Add, unsigned long Data)
;
; Writes a DWORD to the given memory location in big-endian format. The
; memory address MUST be word-aligned.
;
; IN:  R12    Address      (Add)
;      R14    Lower Word   (Data)
;      R15    Upper Word   (Data)
;-------------------------------------------------------------------------------
WriteDWBE
   swpb    R14          ; Swap bytes in lower word
   swpb    R15          ; Swap bytes in upper word
   mov.w   R15,0(R12)   ; Write 1st word to memory
   mov.w   R14,2(R12)   ; Write 2nd word to memory
   ret
```

CCS Version:

```
;-------------------------------------------------------------------------------
; void WriteDWBE(unsigned char *Add, unsigned long Data)
;
; Writes a DWORD to the given memory location in big-endian format. The
; memory address MUST be word-aligned.
;
; IN:  R12    Address     (Add)
;      R13    Lower Word  (Data)
;      R14    Upper Word  (Data)
;-------------------------------------------------------------------------------
WriteDWBE
   swpb    R13          ; Swap bytes in lower word
   swpb    R14          ; Swap bytes in upper word
   mov.w   R14,0(R12)   ; Write 1st word to memory
   mov.w   R13,2(R12)   ; Write 2nd word to memory
   ret
```

7.7 Other Differences

7.7.1 Initializing Static and Global Variables

The ANSI/ISO C standard specifies that static and global (extern) variables without explicit initializations must be pre-initialized to 0 (before the program begins running). This task is typically performed when the program is loaded and is implemented in the IAR compiler:

```
/* IAR, global variable, initialized to 0 upon program start */
int Counter;
```

However, the TI CCS compiler does not pre-initialize these variables; therefore, it is up to the application to fulfill this requirement:

```
/* CCS, global variable, manually zero-initialized */
int Counter = 0;
```

7.7.2 Custom Boot Routine

With the IAR compiler, the C start-up function can be customized, giving the application a chance to perform early initializations such as configuring peripherals, or omit data segment initialization. This is achieved by providing a customized __low_level_init() function:

```
/* IAR C Code */
int __low_level_init(void)
{ =
/* Insert your low-level initializations here */
/*================================== */
/* Choose if segment initialization  */
/* should be done or not.            */
/* Return:  0 to omit initialization */
/*          1 to run initialization  */
/*================================== */
return (1);
}
```

The return value controls whether or not data segments are initialized by the C start-up code. With the CCS C compiler, the custom boot routine name is _system_pre_init(). It is used the same way as in the IAR compiler.

```
/* CCS C Code */
int _system_pre_init(void)
{
/* Insert your low-level initializations here */
/*================================== */
/* Choose if segment initialization  */
/* should be done or not.            */
/* Return:  0 to omit initialization */
/*          1 to run initialization  */
/*================================== */
return (1);
}
```

Omitting segment initialization with both compilers omits both explicit and nonexplicit initialization. The user must ensure that important variables are initialized at run time before they are used.

7.7.3 Predefined Memory Segment Names

Memory segment names for data and function placement are controlled by device-specific linker command files in both CCS and IAR tools. However, different segment names are used. See the linker command files for more detailed information. The following table shows how to convert the most commonly used segment names.

Description	CCS Segment Name	IAR Segment Name
RAM	.bss	DATA16_N DATA16_I DATA16_Z
Stack (RAM)	.stack	CSTACK
Main memory (flash or ROM)	.text	CODE
Information memory (flash or ROM)	.infoA .infoB	INFOA INFOB INFO
Interrupt vectors (flash or ROM)	.int00 .int01 … .int14	INTVEC
Reset vector (flash or ROM)	.reset	RESET

7.7.4 Predefined Macro Names

Both IAR and CCS compilers support a few non ANSI/ISO standard predefined macro names, which help creating code that can be compiled and used on different compiler platforms. Check if a macro name is defined using the #ifdef directive.

Description	CCS Macro Name	IAR Macro Name
Is MSP430 the target and is a particular compiler platform used?	__MSP430__	__ICC430__
Is a particular compiler platform used?	__TI_COMPILER_VERSION__	__IAR_SYSTEMS_ICC__
Is a C header file included from within assembly source code?	__ASM_HEADER__	__ IAR_SYSTEMS_ASM__

8 Migration of Assembler Code from IAR 2.x, 3.x, 4.x, 5.x, 6.x or 7.x to CCS

Source for the TI CCS assembler and source code for the IAR assembler are not 100% compatible. The instruction mnemonics are identical, but the assembler directives are somewhat different. This appendix describes the differences between the CCS assembler directives and the IAR assembler directives.

8.1 Sharing C/C++ Header Files With Assembly Source

The IAR A430 assembler supports certain C/C++ preprocessor directives directly and, thereby, allows direct including of C/C++ header files such as the MSP430 device-specific header files (msp430xxxx.h) into the assembly code:

```
#include "msp430x14x.h" // Include device header file
```

With the CCS Asm430 assembler, a different scheme that uses the .cdecls directive must be used. This directive allows programmers in mixed assembly and C/C++ environments to share C/C++ headers containing declarations and prototypes between the C/C++ and assembly code:

```
.cdecls C,LIST,"msp430x14x.h" ; Include device header file
```

More information on the .cdecls directive can be found in the *MSP430 Assembly Language Tools User's Guide*.

8.2 Segment Control

The CCS Asm430 assembler does not support any of the IAR A430 segment control directives such as ORG, ASEG, RSEG, and COMMON.

Description	Asm430 Directive (CCS)
Reserve space in the .bss uninitialized section	.bss
Reserve space in a named uninitialized section	.usect
Allocate program into the default program section (initialized)	.text
Allocate data into a named initialized section	.sect

To allocate code and data sections to specific addresses with the CCS assembler, it is necessary to create and use memory sections defined in the linker command files. The following example demonstrates interrupt vector assignment in both IAR and CCS assembly to highlight the differences.

```
;-------------------------------------------------------------------------
; Interrupt Vectors Used MSP430x11x1 and 12x(2) - IAR Assembler
;-------------------------------------------------------------------------
   ORG     0FFFEh      ; MSP430 RESET Vector
   DW      RESET       ;
   ORG     0FFF2h      ; Timer_A0 Vector
   DW      TA0_ISR     ;
;------------------------------------------------------------------------- ;
Interrupt Vectors Used MSP430x11x1 and 12x(2) - CCS Assembler
;-------------------------------------------------------------------------
   .sect   ".reset"    ; MSP430 RESET Vector
   .short  RESET       ;
   .sect   ".int09"    ; Timer_A0 Vector
   .short  TA0_ISR     ;
```

Both examples assume that the standard device support files (header files, linker command files) are used. The linker command files are different between IAR and CCS and cannot be reused. See Section 7.7.3 for information on how to translate memory segment names between IAR and CCS.

8.3 Translating A430 Assembler Directives to Asm430 Directives

8.3.1 Introduction

The following sections describe, in general, how to convert assembler directives for the IAR A430 assembler (A430) to TI CCS Asm430 assembler (Asm430) directives. These sections are intended only as a guide for translation. For detailed descriptions of each directive, see either the *MSP430 Assembly Language Tools User's Guide* from TI or the *MSP430 IAR Assembler Reference Guide* from IAR.

Note

Only the assembler *directives* require conversion

Only the assembler directives require conversion, not the assembler instructions. Both assemblers use the same instruction mnemonics, operands, operators, and special symbols such as the section program counter ($) and the comment delimiter (;).

The A430 assembler is not case sensitive by default. These sections show the A430 directives written in uppercase to distinguish them from the Asm430 directives, which are shown in lower case.

8.3.2 Character Strings

In addition to using different directives, each assembler uses different syntax for character strings. A430 uses C syntax for character strings: A quote is represented using the backslash character as an escape character together with quote (\") and the backslash itself is represented by two consecutive backslashes (\\). In Asm430 syntax, a quote is represented by two consecutive quotes (""); see examples:

Character String	Asm430 Syntax (CCS)	A430 Syntax (IAR)
PLAN "C"	"PLAN ""C"""	"PLAN \"C\""
\dos\command.com	"\dos\command.com"	"\\dos\\command.com"
Concatenated string (for example, Error 41)	-	"Error " "41"

8.3.3 Section Control Directives

Asm430 has three predefined sections into which various parts of a program are assembled. Uninitialized data is assembled into the .bss section, initialized data into the .data section, and executable code into the .text section.

A430 also uses sections or segments, but there are no predefined segment names. Often, it is convenient to adhere to the names used by the C compiler: DATA16_Z for uninitialized data, CONST for constant (initialized) data, and CODE for executable code. The following table uses these names.

A pair of segments can be used to make initialized, modifiable data PROM-able. The ROM segment would contain the initializers and would be copied to RAM segment by a start-up routine. In this case, the segments must be exactly the same size and layout.

Description	Asm430 Directive (CCS)	A430 Directive (IAR)
Reserve size bytes in the .bss (uninitialized data) section	.bss(1)	(2)
Assemble into the .data (initialized data) section	.data	RSEG const
Assemble into a named (initialized) section	.sect	RSEG
Assemble into the .text (executable code) section	.text	RSEG code
Reserve space in a named (uninitialized) section	.usect(1)	(2)
Alignment on byte boundary	.align 1	(3)
Alignment on word boundary	.align 2	EVEN

(1) .bss and .usect do not require switching back and forth between the original and the uninitialized section. For example:

```
; IAR Assembler Example
             RSEG   DATA16_N      ; Switch to DATA segment
             EVEN                ; Ensure proper alignment
ADCResult:   DS     2            ; Allocate 1 word in RAM
Flags:       DS     1            ; Allocate 1 byte in RAM
             RSEG   CODE         ; Switch back to CODE segment
; CCS Assembler Example #1
ADCResult   .usect  ".bss",2,2   ; Allocate 1 word in RAM
Flags       .usect  ".bss",1     ; Allocate 1 byte in RAM
; CCS Assembler Example #2
            .bss    ADCResult,2,2 ; Allocate 1 word in RAM
            .bss    Flags,1       ; Allocate 1 byte in RAM
```

(2) Space is reserved in an uninitialized segment by first switching to that segment, then defining the appropriate memory block, and then switching back to the original segment. For example:

```
            RSEG    DATA16_Z
LABEL:      DS      16           ; Reserve 16 byte
            RSEG    CODE
```

(3) Initialization of bit-field constants (.field) is not supported, therefore, the section counter is always byte-aligned.

8.3.4 Constant Initialization Directives

Description	Asm430 Directive (CCS)	A430 Directive (IAR)
Initialize one or more successive bytes or text strings	.byte or .string	DB
Initialize a 32-bit IEEE floating-point constant	.double or .float	DF
Initialize a variable-length field	.field	(1)
Reserve size bytes in the current section	.space	DS
Initialize one or more text strings	Initialize one or more text strings	DB
Initialize one or more 16-bit integers	.word	DW
Initialize one or more 32-bit integers	.long	DL

(1) Initialization of bit-field constants (.field) is not supported. Constants must be combined into complete words using DW.

```
; Asm430 code                 ; A430 code
.field 5,3   \
.field 12,4  | ->             DW (30<<(4+3))|(12<<3)|5  ; equals 3941
.field 30,8  /
```

8.3.5 Listing Control Directives

Description	Asm430 Directive (CCS)	A430 Directive (IAR)
Allow false conditional code block listing	.fclist	LSTCND-
Inhibit false conditional code block listing	.fcnolist	LSTCND+
Set the page length of the source listing	.length	PAGSIZ
Set the page width of the source listing	.width	COL
Restart the source listing	.list	LSTOUT+
Stop the source listing	.nolist	LSTOUT-
Allow macro listings and loop blocks	.mlist	LSTEXP+ (macro) LSTREP+ (loop blocks)
Inhibit macro listings and loop blocks	.mnolist	LSTEXP- (macro) LSTREP- (loop blocks)
Select output listing options	.option	(1)
Eject a page in the source listing	.page	PAGE
Allow expanded substitution symbol listing	.sslist	(2)
Inhibit expanded substitution symbol listing	.ssnolist	(2)
Print a title in the listing page header	.title	(3)

(1) No A430 directive directly corresponds to .option. The individual listing control directives (above) or the command-line option -c (with suboptions) should be used to replace the .option directive.
(2) There is no directive that directly corresponds to .sslist and .ssnolist.
(3) The title in the listing page header is the source file name.

8.3.6 File Reference Directives

Description	Asm430 Directive (CCS)	A430 Directive (IAR)
Include source statements from another file	.copy or .include	#include or $
Identify one or more symbols that are defined in the current module and used in other modules	.def	PUBLIC or EXPORT
Identify one or more global (external) symbols	.global	(1)
Define a macro library	.mlib	(2)
Identify one or more symbols that are used in the current module but defined in another module	.ref	EXTERN or IMPORT

(1) The directive .global functions as either .def if the symbol is defined in the current module, or .ref otherwise. PUBLIC or EXTERN must be used as applicable with the A430 assembler to replace the .global directive.
(2) The concept of macro libraries is not supported. Include files with macro definitions must be used for this functionality.

Modules may be used with the Asm430 assembler to create individually linkable routines. A file may contain multiple modules or routines. All symbols except those created by DEFINE, #define (IAR preprocessor directive) or MACRO are "undefined" at module end. Library modules are, furthermore, linked conditionally. This means that a library module is included in the linked executable only if a public symbol in the module is referenced externally. The following directives are used to mark the beginning and end of modules in the A430 assembler.

Additional A430 Directives (IAR)	A430 Directive (IAR)
Start a program module	NAME or PROGRAM
Start a library module	MODULE or LIBRARY
Terminate the current program or library module	ENDMOD

8.3.7 Conditional Assembly Directives

Description	Asm430 Directive (CCS)	A430 Directive (IAR)
Optional repeatable block assembly	.break	(1)
Begin conditional assembly	.if	IF
Optional conditional assembly	.else	ELSE
Optional conditional assembly	.elseif	ELSEIF
End conditional assembly	.endif	ENDIF
End repeatable block assembly	.endloop	ENDR
Begin repeatable block assembly	.loop	REPT

(1) There is no directive that directly corresponds to .break. However, the EXITM directive can be used with other conditionals if repeatable block assembly is used in a macro, as shown:

```
SEQ     MACRO   FROM,TO     ; Initialize a sequence of byte constants
        LOCAL   X
X       SET     FROM
        REPT    TO-FROM+1   ; Repeat from FROM to TO
        IF      X>255       ; Break if X exceeds 255
        EXITM
        ENDIF
        DB      X           ; Initialize bytes to FROM...TO
X       SET     X+1         ; Increment counter
        ENDR
        ENDM
```

8.3.8 Symbol Control Directives

The scope of assembly-time symbols differs in the two assemblers. In Asm430, definitions can be global to a file or local to a module or macro. Local symbols can be undefined with the .newblock directive. In A430, symbols are either local to a macro (LOCAL), local to a module (EQU), or global to a file (DEFINE). In addition, the preprocessor directive #define also can be used to define local symbols.

Description	Asm430 Directive (CCS)	A430 Directive (IAR)
Assign a character string to a substitution symbol	.asg	SET or VAR or ASSIGN
Undefine local symbols	.newblock	(1)
Equate a value with a symbol	.equ or .set	EQU or =
Perform arithmetic on numeric substitution symbols	.eval	SET or VAR or ASSIGN
End structure definition	.endstruct	(2)
Begin a structure definition	.struct	(2)
Assign structure attributes to a label	.tag	(2)

(1) No A430 directive directly corresponds to .newblock. However, #undef may be used to reset a symbol that was defined with the #define directive. Also, macros or modules may be used to achieve the .newblock functionality because local symbols are implicitly undefined at the end of a macro or module.

(2) Definition of structure types is not supported. Similar functionality is achieved by using macros to allocate aggregate data and base address plus symbolic offset, as shown:

```
MYSTRUCT: MACRO
            DS 4
            ENDM
LO          DEFINE  0
HI          DEFINE  2
            RSEG    DATA16_Z
X           MYSTRUCT
            RSEG    CODE
            MOV     X+LO,R4
            ...
```

8.3.9 Macro Directives

Description	Asm430 Directive (CCS)	A430 Directive (IAR)
Define a macro	.macro	MACRO
Exit prematurely from a macro	.mexit	EXITM
End macro definition	.endm	ENDM

8.3.10 Miscellaneous Directives

Description	Asm430 Directive (CCS)	A430 Directive (IAR)
Send user-defined error messages to the output device	.emsg	#error
Send user-defined messages to the output device	.mmsg	#message(1)
Send user-defined warning messages to the output device	.wmsg	(2)
Define a load address label	.label	(3)
Directive produced by absolute lister	.setsect	ASEG(4)
Directive produced by absolute lister	.setsym	EQU or =(4)
Program end	.end	END

(1) The syntax of the #message directive is: #message "<string>"
This causes '#message <string>' to be output to the project build window during assemble and compile time.

(2) Warning messages cannot be user-defined. #message may be used, but the warning counter is not incremented.

(3) The concept of load-time addresses is not supported. Run-time and load-time addresses are assumed to be the same. To achieve the same effect, labels can be given absolute (run-time) addresses by the EQU directives.

```
; Asm430 code          ; A430 code
.label  load_start     load_start:
Run_start:                 <code>
   <code>              load_end:
Run_end:               run_start: EQU 240H
.label  load_end       run_end:   EQU run_start+load_end-load_start
```

(4) Although not produced by the absolute lister, ASEG defines absolute segments and EQU can be used to define absolute symbols.

```
MYFLAG    EQU   23EH       ; MYFLAG is located at 23E
          ASEG  240H       ; Absolute segment at 240
MAIN:     MOV   #23CH, SP  ; MAIN is located at 240
...
```

8.3.11 Alphabetical Listing and Cross Reference of Asm430 Directives

Asm430 Directive (CCS)	A430 Directive (IAR)	Asm430 Directive (CCS)	A430 Directive (IAR)
.align	ALIGN	.loop	REPT
.asg	SET or VAR or ASSIGN	.macro	MACRO
.break	See Section 8.3.7	.mexit	EXITM
.bss	See Section 8.3.8	.mlib	See Section 8.3.6
.byte or .string	DB	.mlist	LSTEXP+ (macro)
.cdecls	C pre-processor declarations are inherently supported.		LSTREP+ (loop blocks)
.copy or .include	#include or $	.mmsg	#message (XXXXXX)
.data	RSEG	.mnolist	LSTEXP- (macro)
.def	PUBLIC or EXPORT		LSTREP- (loop blocks)
.double	Not supported	.newblock	See Section 8.3.8
.else	ELSE	.nolist	LSTOUT-
.elseif	ELSEIF	.option	See Section 8.3.5
.emsg	#error	.page	PAGE
.end	END	.ref	EXTERN or IMPORT
.endif	ENDIF	.sect	RSEG
.endloop	ENDR	.setsect	See Section 8.3.10
.endm	ENDM	.setsym	See Section 8.3.10
.endstruct	See Section 8.3.8	.space	DS
.equ or .set	EQU or =	.sslist	Not supported
.eval	SET or VAR or ASSIGN	.ssnolist	Not supported
.even	EVEN	.string	DB
.fclist	LSTCND-	.struct	See Section 8.3.8
.fcnolist	LSTCND+	.tag	See Section 8.3.8
.field	See Section 8.3.4	.text	RSEG
.float	See Section 8.3.4	.title	See Section 8.3.5
.global	See Section 8.3.6	.usect	See Section 8.3.8
.if	IF	.width	COL
.label	See Section 8.3.10	.wmsg	See Section 8.3.10
.length	PAGSIZ	.word	DW
.list	LSTOUT+		

8.3.12 Unsupported A430 Directives (IAR)

The following IAR assembler directives are not supported in the CCS Asm430 assembler:

Conditional Assembly Directives	**Macro Directives**	
REPTC(1)	LOCAL(2)	
REPTI		
File Referencing Directives	**Miscellaneous Directives**	**Symbol Control Directives**
NAME or PROGRAM	RADIX	DEFINE
MODULE or LIBRARY	CASEON	SFRB
ENDMOD	CASEOFF	SFRW
Listing Control Directives	**C-Style Preprocessor Directives(3)**	**Symbol Control Directives**
LSTMAC (±)	#define	ASEG
LSTCOD (±)	#undef	RSEG
LSTPAG (±)	#if, #else, #elif	COMMON
LSTXREF (±)	#ifdef, #ifndef	STACK
	#endif	ORG
	#include	
	#error	

(1) There is no direct support for IAR REPTC and REPTI directives in CCS. However, equivalent functionality can be achieved using the CCS .macro directive:

```
; IAR Assembler Example
            REPTI   zero,"R4","R5","R6"
            MOV     #0,zero
            ENDR
; CCS Assembler Example
zero_regs .macro list
            .var item
            .loop
            .break ($ismember(item, list) = 0)
            MOV #0,item
            .endloop
            .endm
```

Code that is generated by calling "zero_regs R4,R5,R6":

```
            MOV #0,R4
            MOV #0,R5
            MOV #0,R6
```

(2) In CCS, local labels are defined by using $n (with n=0…9) or with NAME?. Examples are $4, $7, or Test?.

(3) The use of C-style preprocessor directives is supported indirectly through the use of .cdecls. More information on the .cdecls directive can be found in the *MSP430 Assembly Language Tools User's Guide*.

9 Writing Portable C Code for CCS and MSP430-GCC for MSP430

Source code for the TI CCS C compiler and source code for the GCC for MSP430 compiler are not fully compatible. Standard ANSI/ISO C code is portable between these tools, but implementation-specific extensions differ and must be ported. This appendix describes the major differences between the two compilers.

9.1 Interrupt Vector Definition

The syntax for ISR declarations used by the CCS C compiler (using the #pragma vector =) is not supported by GCC for MSP430. It is, however, possible to write correct C code for both compilers by wrapping the different declarations in a preprocessor directive:

```
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=PORT1_VECTOR
__interrupt void Port1_ISR(void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(PORT1_VECTOR))) Port1_ISR (void)
#else
#error Compiler not supported!
#endif
```

Further information on the use of the GCC compiler can be found in the document Using the GNU Compiler Collection.

10 FET-Specific Menus

This appendix describes the CCS menus that are specific to the FET.

10.1 Menus

10.1.1 Debug View: Run → Free Run

The debugger uses the device JTAG signals to debug the device. On some MSP430 devices, these JTAG signals are shared with the device port pins. Normally, the debugger maintains the pins in JTAG mode so that the device can be debugged. During this time, the port functionality of the shared pins is not available.

However, when Free Run is selected (by opening a pulldown menu next to the Run icon on top of the Debug View), the JTAG drivers are set to 3-state, and the device is released from JTAG control (TEST pin is set to GND) when GO is activated. Any active on-chip breakpoints are retained, and the shared JTAG port pins revert to their port functions.

At this time, the debugger has no access to the device and cannot determine if an active breakpoint (if any) has been reached. The debugger must be manually commanded to stop the device, at which time the state of the device is determined (that is, was a breakpoint reached?).

See FAQ 9.

10.1.2 Run → Connect Target

Regains control of the device when ticked.

10.1.3 Run → Advanced → Make Device Secure

Blows the JTAG fuse on the target device. After the fuse is blown, no further communication through JTAG with the device is possible.

10.1.4 Project → Properties → Debug → MSP430 Properties → Clock Control

Disables the specified system clock while the debugger has control of the device (following a STOP or breakpoint). All system clocks are enabled following a GO or a single step (STEP or STEP INTO). Can only be changed when the debugger is inactive. See FAQ 12.

10.1.5 Window → Show View → Breakpoints

Opens the MSP430 Breakpoints View window. This window can be used to set basic and advanced breakpoints. Advanced settings such as Conditional Triggers and Register Triggers can be selected individually for each breakpoint by accessing the properties (right click on corresponding breakpoint). Pre-defined breakpoints such as Break on Stack Overflow can be selected by opening the Breakpoint pulldown menu, which is located next to the Breakpoint icon at the top of the window. Breakpoints may be combined by dragging and dropping within the Breakpoint View window. A combined breakpoint is triggered when all breakpoint conditions are met.

10.1.6 Window → Show View → Other... Debug → Trace Control

The Trace View enables the use of the state storage module. The state storage module is present only in devices that contain the full version of the Enhanced Emulation Module (EEM) (see Table 3-1). After a breakpoint is defined, the State Storage View displays the trace information as configured. Various trace modes can be selected when clicking the Configuration Properties icon at the top right corner of the window. Details on the EEM are available in the application report *Advanced Debugging Using the Enhanced Emulation Module (EEM) With Code Composer Studio IDE*.

10.1.7 Project → Properties → Debug → MSP430 Properties → Target Voltage

The target voltage of the MSP-FET430UIF can be adjusted between 1.8 V and 3.6 V. This voltage is available on pin 2 of the 14-pin target connector to supply the target from the USB FET. If the target is supplied externally, the external supply voltage should be connected to pin 4 of the target connector, so the USB FET can set the level of the output signals accordingly. Can only be changed when the debugger is inactive.

11 Device-Specific Menus

11.1 MSP430L092

11.1.1 Emulation Modes

The MSP430L092 can operate in two different modes: the L092 mode and C092 emulation mode. The purpose of the C092 emulation mode is to mimic a C092 with up to 1920 bytes of code at its final destination for mask generation by using an L092. The operation mode must be set in CCS before launching the debugger. The selection happens in the project properties under Device Options at the bottom, after selecting MSP430L092 as Device Variant as shown in Figure 11-1. Figure 11-2 shows how to select the C092 mode.

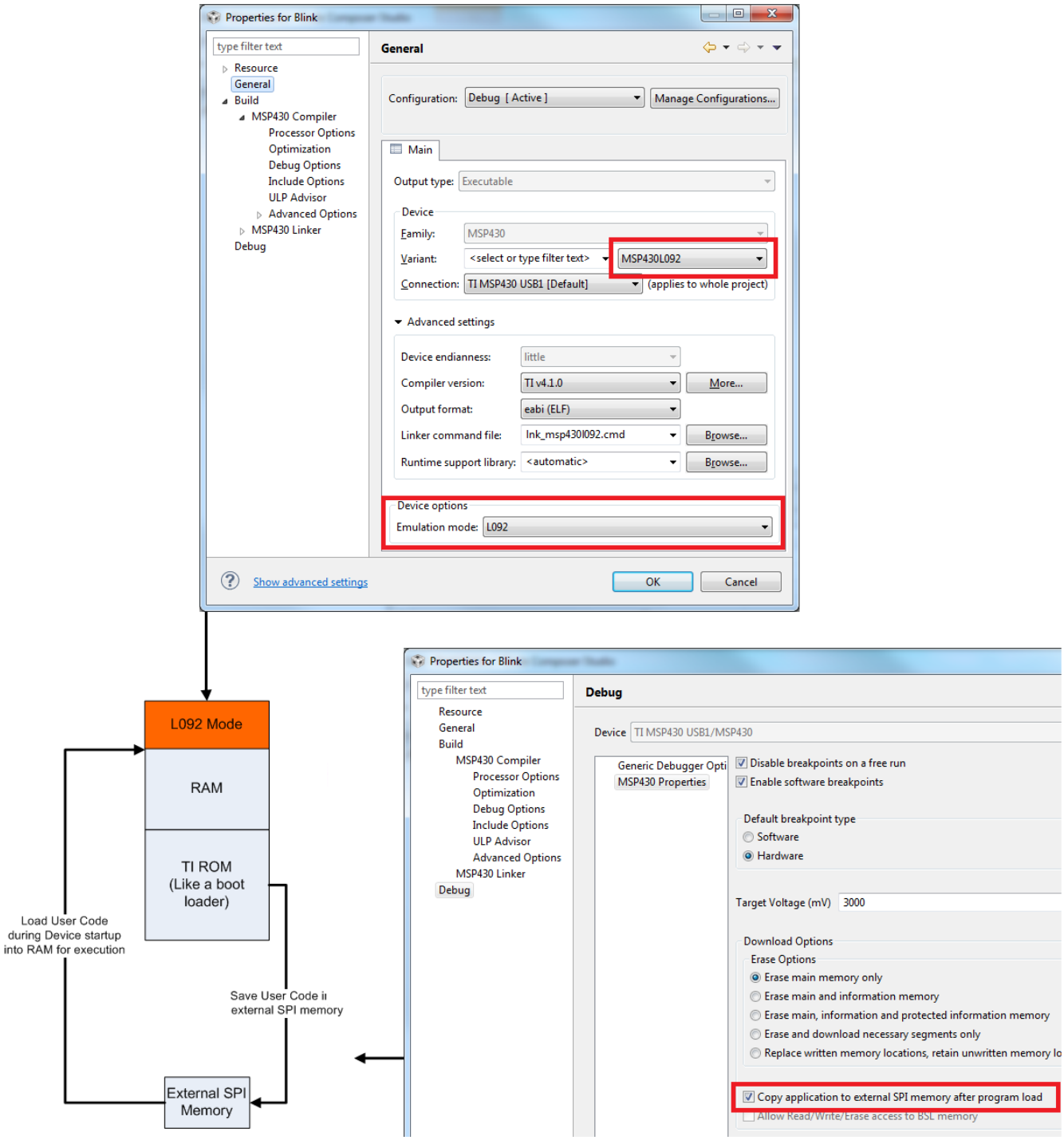

Figure 11-1. MSP430L092 Modes

11.1.2 Loader Code

The Loader Code in the MSP430L092 is a ROM-code from TI that provides a series of services. It enables customers to build autonomous applications without needing to develop a ROM mask. Such an application consists of an MSP430 device containing the loader (for example, MSP430L092) and an SPI memory device (for example, '95512 or '25640). Those and similar devices are available from various manufacturers. The majority of use cases for an application with a loader device and external SPI memory for native 0.9-V supply voltage are late development, prototyping, and small series production. The external code download may be set in the CCS Project Properties → Debug → MSP430 Properties → Download Options → Copy application to external SPI memory after program load (see Figure 11-1).

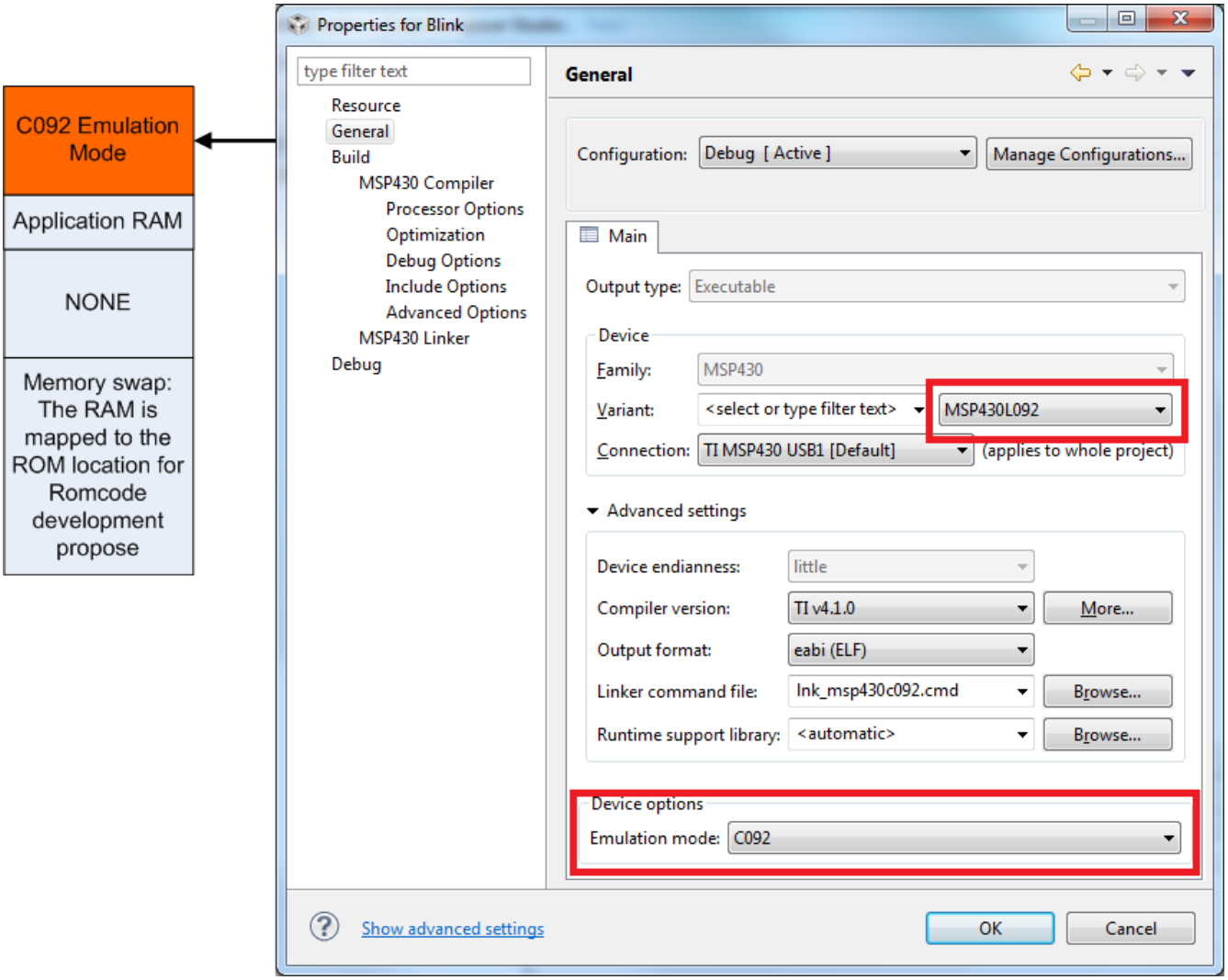

Figure 11-2. MSP430L092 in C092 Emulation Mode

11.1.3 C092 Password Protection

The MSP430C092 is a customer-specific ROM device, which is protected by a password. To start a debug session, the password must be provided to CCS. Open the MSP430C092.CCXML file in your project, click Target Configurations in the Advanced Setup section, Advanced Target Configuration. The CPU Properties become visible after MSP430 is selected. Figure 11-3 shows how to provide a HEX password in CCS v6.1 target configuration.

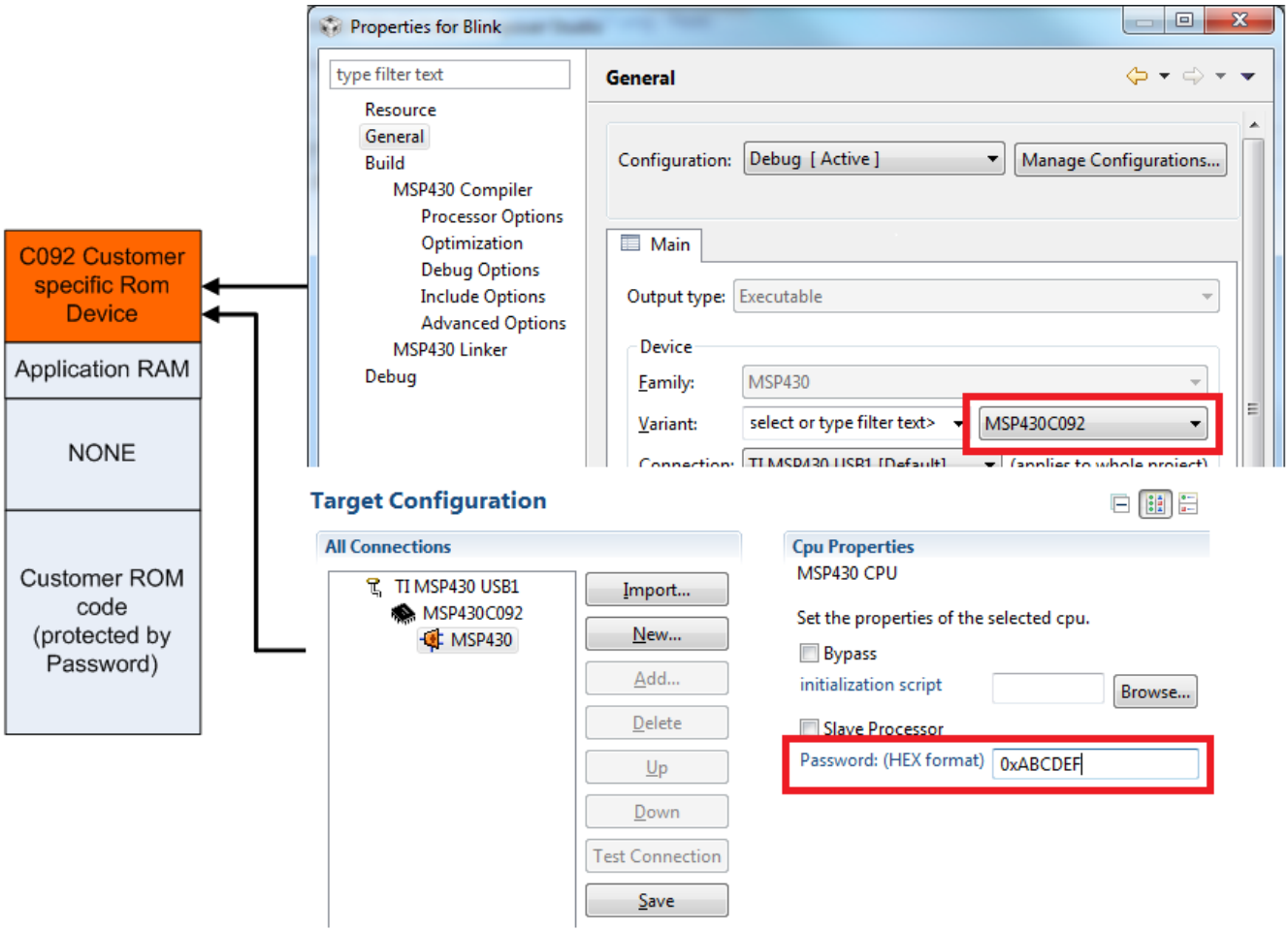

Figure 11-3. MSP430C092 Password Access

11.2 MSP430F5xx and MSP430F6xx BSL Support

Most of the MSP430F5xx and MSP430F6xx devices support a custom BSL that is protected by default. To program the custom BSL, this protection must be disabled in CCS Project Properties → Debug → MSP430 Properties → Download Options → Allow Read/Write/Erase access to BSL memory (see Figure 11-4).

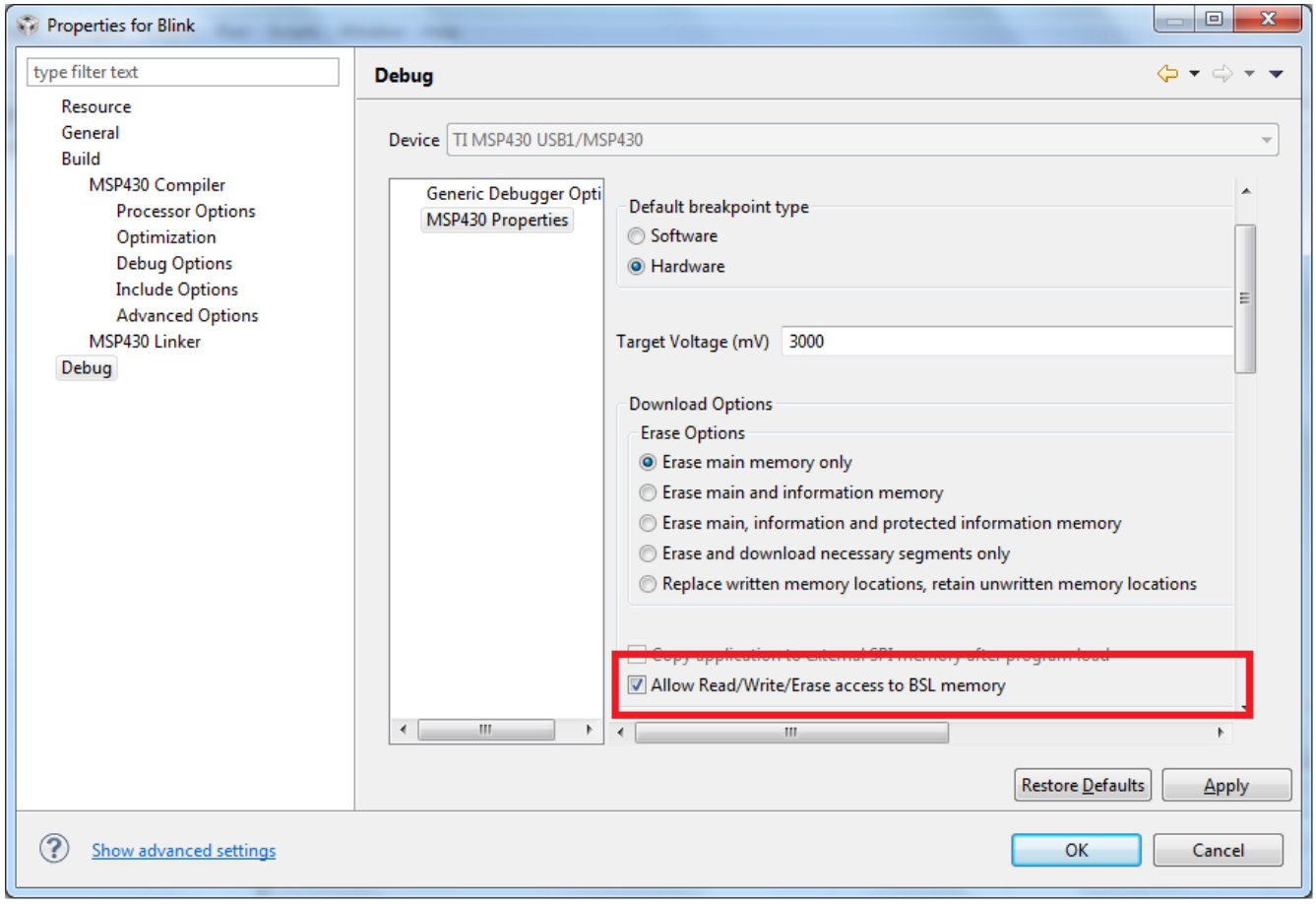

Figure 11-4. Allow Access to BSL

11.3 MSP430FR5xx and MSP430FR6xx Password Protection

Selected MSP430FR5xx and MSP430FR6xx devices provide JTAG protection by a user password. When debugging these MSP430 derivatives, the hexadecimal JTAG password must be provided to start a debug session. Open the MSP430Fxxxx.CCXML file in your project, click Target Configurations in the Advanced Setup section, Advanced Target Configuration. The CPU Properties become visible after MSP430 is selected (see Figure 11-5). Details regarding the password protection of MSP430FR5xx and MSP430FR6xx devices may be found in the device user's guides.

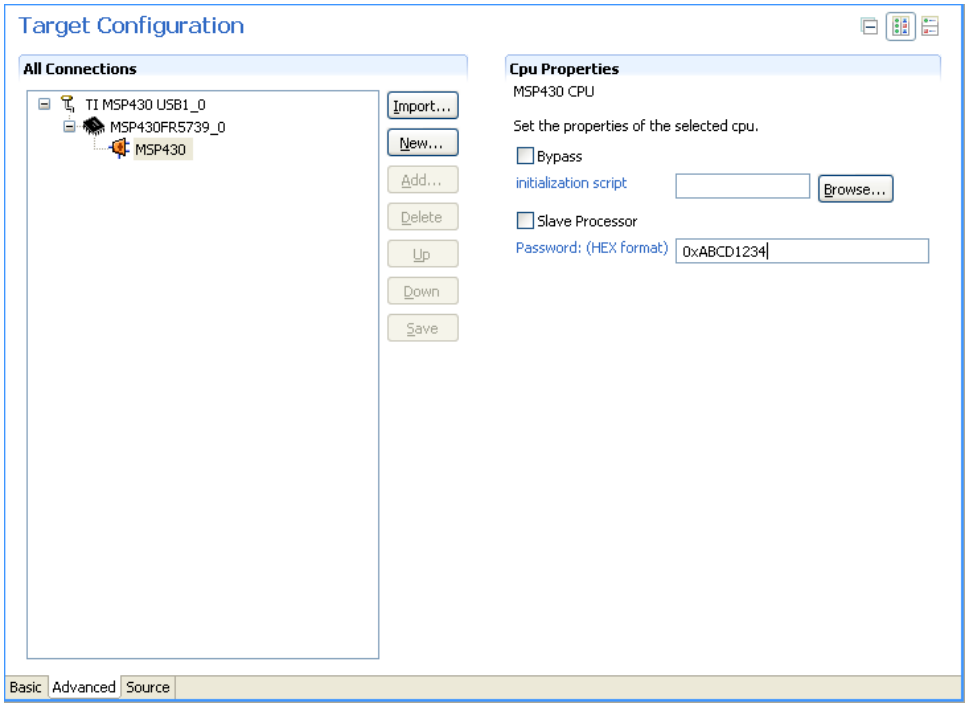

Figure 11-5. MSP430 Password Access

11.4 MSP430 Ultra-Low-Power LPMx.5 Mode

11.4.1 What is LPMx.5

LPMx.5 is an ultra-low power mode in which the entry and exit are handled differently than in the other low-power modes.

LPMx.5 gives the lowest power consumption available on a device. To achieve this, entry to LPMx.5 disables the LDO of the PMM module, which removes the supply voltage from the core and the JTAG module of the device. Because the supply voltage is removed from the core, all register contents and SRAM contents are lost. Exit from LPMx.5 causes a BOR event, which forces a complete reset of the system.

Note

See the corresponding MSP430 device family user's guide for additional LPMx.5 and ultra-low-power debug mode details.

11.4.2 Debugging LPMx.5 Mode on MSP430 Devices That Support the Ultra-Low-Power Debug Mode

To enable the ultra-low power debug mode feature the "Enable Ultra Low Power debug / LPMx.5 debug" checkbox must be enabled by clicking Project Properties -> Debug -> MSP430 Properties -> Enable Ultra Low Power debug / LPMx.5 debug (see Figure 11-6).

When the ultra-low power debug mode is enabled a notification is displayed in the Debugger log every time the target device enters and leaves LPMx.5 mode.

Press the Halt or Reset button CCS to wake up the target device from LPMx.5. Execution of the code is halted at the start of the program. All breakpoints that had been active before LPMx.5 are restored and reactivated automatically.

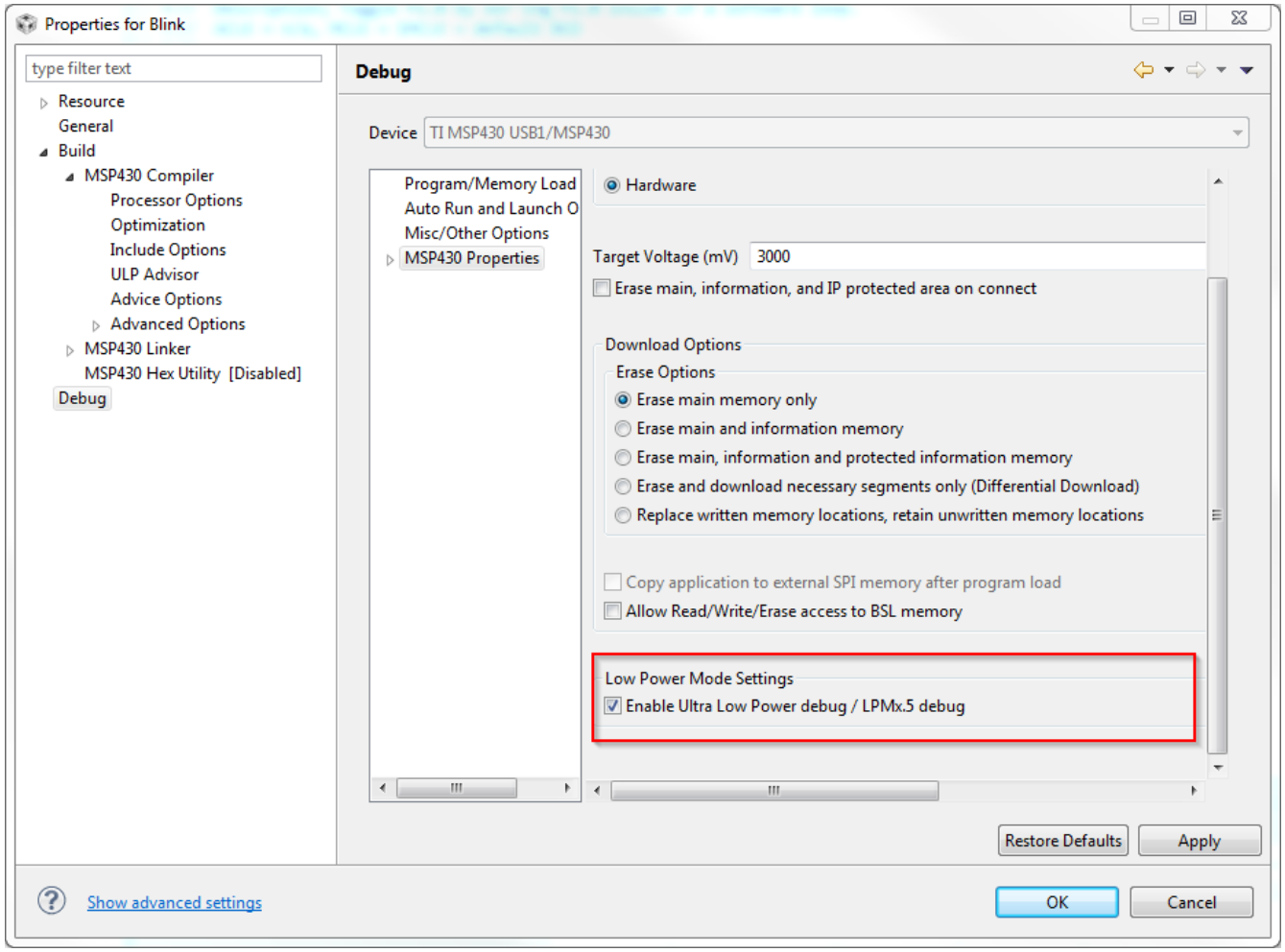

Figure 11-6. Enable Ultra-Low-Power Debug Mode

11.4.2.1 Limitations

When a target device is in LPMx.5 mode, it is not possible to set or remove advanced conditional or software breakpoints. It is, however, possible to set hardware breakpoints. In addition, only hardware breakpoints that were set during LPMx.5 can be removed in the LPMx.5 mode. Attach to running target is not possible in combination with LPMx.5 mode debugging, as this results in a device reset.

When using the "Free Run" option in combination with LPMx.5 mode, the target device does not resume code execution after LPMx.5 wakeup. In this case, suspend the debug session by clicking the "Suspend" button, and then resume the session by clicking the "Resume" button.

11.4.3 Debugging LPMx.5 Mode on MSP430 Devices That Do Not Support the Ultra-Low-Power Debug Mode

On MSP430 devices that do not support the ultra-low-power mode, the LPMx.5 low-power mode can be debugged using the "Free Run" option. This configuration provides the absolute current and energy consumption of MSP430 LPMx.5 low-power mode.

11.4.3.1 Limitations

Using this configuration presents some limitations:

1. Breakpoint
   a. Setting or erasing any kind of breakpoint is not possible when the device is in LPMx.5.
2. Device State
   a. There are no notifications about the current device state. From the perspective of the IDE, the device is running.
3. Pause
   a. The pause button might not work reliably when the device is in LPMx.5 mode. The device might not leave LPMx.5 mode when the pause button is pressed. In this case, the debug session must be restarted. During debugging, an option is to trap the device in active mode after wake-up from LPMx.5, so that the device can be paused/suspend reliably when it is in a known power mode other than LPMx.5.
4. Debugger connection
   a. To make sure that the debugger can always connect and synchronize to the MSP430 device.
      1. Do not enter LPMx.5 directly after code start. A 500-ms delay is required between code start and LPMx.5 entry to ensure reliable debugger synchronization.
      2. If 4-wire JTAG shows connection and synchronization errors, use 2-wire SBW instead of the 4-wire JTAG protocol.
      3. Make sure that the code removes the lock I/O setting for all MSP430 port pins.

12 Revision History

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

Changes from May 12, 2018 to May 4, 2020	Page
• Changed document title and all instances of Code Composer Studio v8.x to v10.x	3
• Changed the MSP-GANG430 programmer to MSP-GANG and removed the MSP-PRGS430 in Section 3.1.6, *How to Generate Binary Format Files (TI-TXT and INTEL-HEX)*	9
• Changed Figure 3-3, *Download Options*	15
• Changed the versions of IAR Embedded Workbench IDE for migration in Section 7, *Migration of C Code from IAR 2.x, 3.x, 4.x, 5.x, 6.x or 7.x to CCS*	42
• Changed the versions of IAR Embedded Workbench IDE for migration in Section 8, *Migration of Assembler Code from IAR 2.x, 3.x, 4.x, 5.x, 6.x or 7.x to CCS*	47

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), 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, regulatory 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 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.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265