Code Composer Studio™ v5.1 User's Guide for MSP430™

User's Guide

Literature Number: SLAU157T
May 2005–Revised September 2011

Contents

List of Figures

List of Tables

Read This First

About This Manual

This manual describes the use of Texas Instruments™ Code Composer Studio™ v5.1 (CCSv5.1) with the MSP430™ ultra-low-power microcontrollers.

How to Use This Manual

Read and follow the instructions in the Get Started Now! chapter. This chapter provides 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 1.3, *Important MSP430 Documents on the CD-ROM and Web*.

This manual applies to the use with Texas Instruments' MSP-FET430UIF, MSP-FET430PIF, and eZ430 development tools series.

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

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.

Related Documentation From Texas Instruments

CCSv5.1 documentation

MSP430™ Assembly Language Tools User's Guide, literature number SLAU131

MSP430™ Optimizing C/C++ Compiler User's Guide, literature number SLAU132

MSP430™ development tools documentation

MSP430™ Hardware Tools User's Guide, literature number SLAU278

eZ430-F2013 Development Tool User's Guide, literature number SLAU176

eZ430-RF2480 User's Guide, literature number SWRA176

eZ430-RF2500 Development Tool User's Guide, literature number SLAU227

eZ430-RF2500-SEH Development Tool User's Guide, literature number SLAU273

eZ430-Chronos™ Development Tool User's Guide, literature number SLAU292

MSP-EXP430G2 LaunchPad Experimenter Board User's Guide, literature number SLAU318

MSP430xxxx device data sheets

MSP430x1xx Family User's Guide, literature number SLAU049

MSP430x2xx Family User's Guide, literature number SLAU144

MSP430x3xx Family User's Guide, literature number SLAU012

MSP430x4xx Family User's Guide, literature number SLAU056

MSP430x5xx/x6xx Family User's Guide, literature number SLAU208

CC430 Family User's Guide, literature number SLAU259

If You Need Assistance

Support for the MSP430 microcontrollers and the FET development tools is provided by the Texas Instruments Product Information Center (PIC). Contact information for the PIC can be found on the TI web site at www.ti.com/support. A Code Composer Studio specific Wiki page (FAQ) is available, and the Texas Instruments E2E Community support forums for the MSP430 and Code Composer Studio v5.1 provide open interaction with peer engineers, TI engineers, and other experts. Additional device-specific information can be found on the MSP430 web site.

FCC Warning

This equipment is intended for use in a laboratory test environment only. It generates, uses, and can radiate radio frequency energy and has not been tested for compliance with the limits of computing devices pursuant to subpart J of part 15 of FCC rules, which are designed to provide reasonable protection against radio-frequency interference. Operation of this equipment in other environments may cause interference with radio communications, in which case, the user is required to take whatever measures may be required to correct this interference at his own expense.

Get Started Now!

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

Topic **Page**

1.1 Software Installation

To install Code Composer Studio™ v5.1 (CCS), run setup_CCS_x.x.x.x.exe from the DVD. If the CCS package was downloaded, please ensure to extract the full zip archive before running setup_CCS_x.x.x.x.exe. Follow the instructions shown on the screen. The hardware drivers for the USB JTAG emulators (MSP-FET430UIF and eZ430 series) are installed automatically when installing CCS. The driver for the parallel-port FET (MSP-FET430PIF) are not installed by default, but can be selected manually during the installation process.

NOTE: Support of MSP-FET430PIF (parallel port emulators).

The driver and IDE components supporting the MSP-FET430PIF parallel port interface are not installed by default. Select them manually during the CCSv5.1 installation process.

Please fully extract the zip archive setup_CCS_x_x_x.zip file before running setup_CCS_x.x.x.x.exe.

Table 1-1. System Requirements

	Recommended System Requirements	Minimum System Requirements
Processor	Dual Core	1.5 GHz
RAM	2 GB	1 GB
Free Disk Space	2 GB	300 MB (depends on features selected during installation)
Operating System	Microsoft® Windows® XP with SP2 (32/64 bit) or Windows Vista® with SP1 (32/64 bit) or Windows 7® (32/64 bit)	Microsoft® Windows® XP with SP2 (32/64 bit) or Windows Vista® (32/64 bit) or Windows 7® (32/64 bit)

1.2 Flashing the LED

This section demonstrates on the FET the equivalent of the C-language "Hello world!" introductory program. CCSv5.1 includes C and ASM code files as well as fully pre-configured projects. The following describes how an application that flashes the LED is developed, downloaded to the FET, and run.

1. Start Code Composer Studio Start → All Programs → Texas Instruments → Code Composer Studio → Code Composer Studio.
2. Create a new Project by selecting File → New → CCS Project.
3. Enter a project name and select Device Variant
4. If using a USB Flash Emulation Tool such as the MSP-FET430UIF or the eZ430 Development Tool, they should be already configured by default. In case you are using the MSP-FET430PIF LPT interface you need to select TI MSP430 LPTx (in case support for the MSP430 Parallel Port Tools was selected during the installation).
5. Select "Blink The LED" basic Example in the Project templates and examples section.
6. Click Finish.

NOTE: The predefined example works with most MSP430 boards. Certain MSP430x4xx boards use Port P5.0 for the LED connection. In addition the MSP430L092 board requires a different code example. Code for these examples are available. See Table 1-2 for details.

Table 1-2. Code Examples

MSP430 Devices	Code Example
MSP430x1xx device family	<...>\msp430x1xx\C-Source\msp430x1xx.c
MSP430x2xx device family	<...>\msp430x2xx\C-Source\msp430x2xx.c
MSP430x4xx device family	<...>\msp430x4xx\C-Source\msp430x4xx.c
MSP430x5xx device family	<...>\msp430x5xx\C-Source\msp430x5xx.c

Table 1-2. Code Examples (continued)

MSP430 Devices	Code Example
MSP430x6xx device family	<...>\msp430x6xx\C-Source\msp430x6xx.c
MSP430L092	<...>\msp430x5xx\C-Source\msp430l092.c

7. To compile the code and download the application to the target device, go to Run → Debug (F11).
8. The application may be started by selecting Run → Resume (F8) or clicking the Play button on the toolbar.

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

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

Predefined projects, which are located in <Installation Root>\ccsv5\ccs_base\msp430\examples\example projects, can be imported by selecting Project → Import Existing CCS/CCE Eclipse Project.

1.3 Important MSP430™ Documents on the CD-ROM and Web

The primary sources of MSP430 and CCSv5.1 information are the device-specific data sheets and user's guides. The most up-to-date versions of these documents available at the time of production have been provided on the CD-ROM included with this tool. The MSP430 web site (www.ti.com/msp430) contains the latest version of these documents.

Development Flow

This chapter discusses how to use Code Composer Studio (CCS) to develop application software and how to debug that software.

Topic **Page**

2.1 Using Code Composer Studio (CCS)

The following sections are a brief overview of how to use CCS. For a full discussion of software development flow with CCS in assembly or C, see *MSP430 Assembly Language Tools User's Guide* (SLAU131) and *MSP430 Optimizing C/C++ Compiler User's Guide* (SLAU132).

2.1.1 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 the application on the MSP430 (see Section 2.1.2, *Project Settings*). Also, the MSP430 Code Composer Studio Help presents a more comprehensive overview of the process.

1. Start the 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 please select "Empty Assembly-only Project" in the "Project template and examples" section.
4. If using a USB Flash Emulation Tool such as the MSP-FET430UIF or the eZ430 Development Tool, they should be already configured by default. In case you are using the MSP-FET430PIF LPT interface you need to select TI MSP430 LPTx (in case support for the MSP430 Parallel Port Tools was selected during the installation).
5. For C projects the setup is complete now, main.c will be shown and code can be entered. In case of an assembly project, a new source file will have to be created (File → New → Source File). Enter the file name and remember to add the .asm suffix. 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 .h files to simplify code development.

CCS is supplied with files for each device that define the device registers and the bit names. Using these files is recommended 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" 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 Debugging #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 will return to the C/C++ view (code editor) automatically.
12. Click File → Exit to exit CCS.

2.1.2 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/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 standalone operation, disable Software Breakpoints (Project → Properties → Debug → MSP430 Properties → Enable Software Breakpoints). If Software Breakpoints are enabled, ensure proper termination of each debug session while the target is connected; otherwise, the target may not be operational standalone as the application on the device will still contain the software breakpoint instructions.

2.1.3 Using an Existing CCE v2, CCE v3, CCE v3.1 and CCS v4.x Project

CCS v5.1 supports the conversion of workspaces and projects created in version CCE v2, v3, v3.1 and CCSv4.x to the CCSv5.1 format (File → Import → General → Existing Projects into Workspace → Next). Browse to legacy CCE workspace containing the project to be imported. The Import Wizard lists all projects in the given workspace. Specific Projects can then be selected and converted. CCEv2 and CCEv3 projects may require manual work on the target configuration file (*.ccxml) after import.

The IDE 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 #define statements, which are located at the end of each .h file, in the section "Interrupt Vectors".

2.1.4 Stack Management

The reserved stack size can be configured through the project options dialog (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).

Note that the stack can overflow due to small size or application errors. See Section 2.2.2.1 for a method of tracking the stack size.

2.1.5 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-GANG430 and MSP-PRGS430 programmers, as well as INTEL-HEX format files for TI factory device programming. The tool can be used either standalone in a command line (located in <Installation Root>\ccsv5\ccs_base\tools\compiler\msp430\bin) or directly within CCS. In the latter case, a post-build step can be configured to generate the file automatically after every build by selecting predefined formats such as TI-TXT and INTEL-HEX in the "Apply Predefined Step" pull-down menu (Project → Properties → Build → Build Steps Tab → Post-Build Step → Apply Predefined Step). The generated file is stored in the <Workspace>\<Project>\Debug\ directory.

2.1.6 Overview of Example Programs and Projects

Example programs for MSP430 devices are provided in <Installation Root>\ccsv5\ccs_base\msp430\examples. Assembly and C sources are available in the appropriate subdirectory.

To use the examples, create a new project and add the example source file to the project by clicking Project → Add Files... In addition, example projects corresponding to the code examples are provided in <Installation Root>\ccsv5\ccs_base\msp430\examples\example projects. The projects can be imported by Project → Import Existing CCS/CCE Eclipse Project (see Section 1.2 for more information).

2.2 Using the Integrated Debugger

See Appendix D for a description of FET-specific menus within CCS.

2.2.1 Breakpoint Types

The debugger breakpoint mechanism uses a limited number of on-chip debugging resources (specifically, N breakpoint registers, see Table 2-1). 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 → MSP430 Properties → Enable Software Breakpoints), 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 standalone, 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 2-1. 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
CC430F51xx	MSP430Xv2	X	X	3	X	X		
CC430F61xx	MSP430Xv2	X	X	3	X	X		
MSP430AFE2xx	MSP430	X	X	2		X		
MSP430BT5190	MSP430Xv2	X	X	8	X	X	X	X
MSP430F11x1	MSP430	X		2				
MSP430F11x2	MSP430	X		2				
MSP430F12x	MSP430	X		2				
MSP430F12x2	MSP430	X		2				
MSP430F13x	MSP430	X		3	X			
MSP430F14x	MSP430	X		3	X			
MSP430F15x	MSP430	X		8	X	X	X	X
MSP430F16x	MSP430	X		8	X	X	X	X
MSP430F161x	MSP430	X		8	X	X	X	X
MSP430F20xx	MSP430	X	X	2		X		
MSP430F21x1	MSP430	X		2		X		
MSP430F21x2	MSP430	X	X	2		X		
MSP430F22x2	MSP430	X	X	2		X		
MSP430F22x4	MSP430	X	X	2		X		
MSP430F23x	MSP430	X		3	X	X		
MSP430F23x0	MSP430	X		2		X		
MSP430F24x	MSP430	X		3	X	X		
MSP430F241x	MSP430X	X		8	X	X	X	X
MSP430F2410	MSP430	X		3	X	X		
MSP430F261x	MSP430X	X		8	X	X	X	X
MSP430G2xxx	MSP430	X	X	2		X		
MSP430F41x	MSP430	X		2		X		
MSP430F41x2	MSP430	X	X	2		X		
MSP430F42x	MSP430	X		2		X		
MSP430FE42x	MSP430	X		2		X		
MSP430FE42x2	MSP430	X		2		X		
MSP430FW42x	MSP430	X		2		X		
MSP430F42x0	MSP430	X		2		X		
MSP430FG42x0	MSP430	X		2		X		
MSP430F43x	MSP430	X		8	X	X	X	X
MSP430FG43x	MSP430	X		2		X		
MSP430F43x1	MSP430	X		2		X		
MSP430F44x	MSP430	X		8	X	X	X	X
MSP430F44x1	MSP430	X		8	X	X	X	X
MSP430F461x	MSP430X	X		8	X	X	X	X
MSP430FG461x	MSP430X	X		8	X	X	X	X
MSP430F461x1	MSP430X	X		8	X	X	X	X
MSP430F47x	MSP430	X		2		X		
MSP430FG47x	MSP430	X		2		X		
MSP430F47x3	MSP430	X		2		X		

[1] The 2-wire JTAG debug interface is also referred to as Spy-Bi-Wire (SBW) interface. Note that this interface is supported only by the USB emulators (eZ430-xxxx and MSP-FET430UIF USB JTAG emulator) and the MSP-GANG430 production programming tool. The MSP-FET430PIF parallel port JTAG emulator does not support communication in 2-wire JTAG mode.

Table 2-1. Device Architecture, Breakpoints and Other Emulation Features (continued)

Device	MSP430 Architecture	4-Wire JTAG	2-Wire JTAG [1]	Break-points (N)	Range Break-points	Clock Control	State Sequencer	Trace Buffer
MSP430F47x4	MSP430	X		2		X		
MSP430F471xx	MSP430X	X		8	X	X	X	X
MSP430F51x1	MSP430Xv2	X	X	3	X	X		
MSP430F51x2	MSP430Xv2	X	X	3	X	X		
MSP430F52xx	MSP430Xv2	X	X	8	X	X	X	X
MSP430F530x	MSP430Xv2	X	X	3	X	X		
MSP430F5310	MSP430Xv2	X	X	3	X	X		
MSP430F532x	MSP430Xv2	X	X	8	X	X	X	X
MSP430F533x	MSP430Xv2	X	X	8	X	X	X	X
MSP430F534x	MSP430Xv2	X	X	8	X	X	X	X
MSP430F54xx	MSP430Xv2	X	X	8	X	X	X	X
MSP430F54xxA	MSP430Xv2	X	X	8	X	X	X	X
MSP430F550x	MSP430Xv2	X	X	3	X	X		
MSP430F5510	MSP430Xv2	X	X	3	X	X		
MSP430F552x	MSP430Xv2	X	X	8	X	X	X	X
MSP430F563x	MSP430Xv2	X	X	8	X	X	X	X
MSP430FR57xx	MSP430Xv2	X	X	3	X	X		
MSP430F643x	MSP430Xv2	X	X	8	X	X	X	X
MSP430F663x	MSP430Xv2	X	X	8	X	X	X	X
MSP430F67xx	MSP430Xv2	X	X	3	X	X		
MSP430L092	MSP430Xv2	X		2		X		

2.2.2 Using Breakpoints

If the debugger is started with greater than N breakpoints set and software breakpoints are disabled, a message is shown that informs the user that not all breakpoints can be enabled. Note that 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 → Generic Debugger 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 → Generic Debug Options → Enable CIO function use.

2.2.2.1 Breakpoints in CCSv5.1

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/watchpoints can be changed in the debugger by right clicking on the breakpoint and selecting 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 debug 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.
- **Watch**

  Stops code execution if a specific 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 BP there). Then set the watchpoint and delete (or disable) the code breakpoint in the function and run/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 2-1). 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 the application report *Advanced Debugging Using the Enhanced Emulation Module (EEM) With CCS Version 4* (SLAA393).

Frequently Asked Questions

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

Topic **Page**

A.1 Hardware

For a complete list of hardware related FAQs, see the *MSP430 Hardware Tools User's Guide* SLAU278.

A.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* (literature number SLAU132).
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/or statements that have no effect** and can affect debugging. To prevent this, these variables can be declared `volatile`; for example, `volatile int i;`.

A.3 Debugging

The debugger is part of CCS and can be used as a standalone application. This section is applicable when using the debugger both standalone 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:
   - Ensure that the correct debug interface and corresponding port number have been selected in Project → Properties → General → Device → Connection.
   - Ensure that the jumper settings are configured correctly on the target hardware.
   - Ensure that no other software application (for example, printer drivers) has reserved or taken control of the COM/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/parallel port is successfully recognized by the Windows OS. Check the PC BIOS for the parallel port settings (see FAQ Debugging #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.

   Ensure 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 utilize interrupts and low-power modes.** See FAQ Debugging #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 MSP-FET430PIF and 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. This occurs for MSP-FET430PIF and MSP-FET430UIF but is mainly seen on MSP-FET430UIF.

   Workarounds:
   - Connect the device $\overline{\text{RST}}$/NMI pin to JTAG header (pin 11), MSP-FET430PIF/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.
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 Debugging #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 via a resistor on the PCB.

   $\overline{\text{RST}}$/NMI 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 Debugging #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 utilizes 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/44x 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 2.2.2 for a detailed overview.

IAR 2.x/3.x/4.x to CCS C-Migration

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

Topic **Page**

B.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.

B.2 Intrinsic Functions

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

B.3 Data and Function Placement

B.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.

B.3.2 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 LOCATION(alpha, "MYSEGMENT")
int alpha;

#pragma LOCATION(beta, "MYSEGMENT")
int beta;
```

See Section B.5.3 for information on how to translate memory segment names between IAR and CCS.

B.3.3 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 B.5.3 for information on how to translate memory segment names between IAR and CCS.

B.4 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* (SLAU132) 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
```

B.5 Other Differences

B.5.1 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;
```

B.5.2 Custom Boot Routine

With the IAR compiler, the C startup 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 startup 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);
}
```

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

B.5.3 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

B.5.4 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__

IAR 2.x/3.x/4.x to CCS Assembler Migration

Source for the TI CCS assembler and source code for the IAR assembler are not 100% compatible. The instruction mnemonics are identical, while the assembler directives are somewhat different. This appendix documents the differences between the CCS assembler directives and the IAR 2.x/3.x assembler directives.

Topic **Page**

C.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* (literature number SLAU131).

C.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/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 MSP430x11x1/12x(2) - IAR Assembler
;-------------------------------------------------------------------------
   ORG     0FFFEh      ; MSP430 RESET Vector
   DW      RESET       ;
   ORG     0FFF2h      ; Timer_A0 Vector
   DW      TA0_ISR     ;
;------------------------------------------------------------------------- ;
Interrupt Vectors Used MSP430x11x1/12x(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. Note that the linker command files are different between IAR and CCS and cannot be reused. See Section B.5.3 for information on how to translate memory segment names between IAR and CCS.

C.3 Translating A430 Assembler Directives to Asm430 Directives

C.3.1 Introduction

The following sections describe, in general, how to convert assembler directives for the IAR A430 assembler (A430) to Texas Instruments 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* (SLAU131), from Texas Instruments, 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.

C.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"

C.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.

C.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  /
```

C.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/.ssnolist.
(3) The title in the listing page header is the source file name.

C.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

C.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
```

C.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
            ...
```

C.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

C.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/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
...
```

C.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 Conditional Assembly Directives	.mexit	EXITM
.bss	See Symbol Control Directives	.mlib	See File Referencing Directives
.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 Symbol Control Directives
.else	ELSE	.nolist	LSTOUT-
.elseif	ELSEIF	.option	See Listing Control Directives
.emsg	#error	.page	PAGE
.end	END	.ref	EXTERN or IMPORT
.endif	ENDIF	.sect	RSEG
.endloop	ENDR	.setsect	See Miscellaneous Directives
.endm	ENDM	.setsym	See Miscellaneous Directives
.endstruct	See Symbol Control Directives	.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 Symbol Control Directives
.fcnolist	LSTCND+	.tag	See Symbol Control Directives
.field	See Constant Initialization Directives	.text	RSEG
.float	See Constant Initialization Directives	.title	See Listing Control Directives
.global	See File Referencing Directives	.usect	See Symbol Control Directives
.if	IF	.width	COL
.label	See Miscellaneous Directives	.wmsg	See Miscellaneous Directives
.length	PAGSIZ	.word	DW
.list	LSTOUT+		

C.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/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* (literature number SLAU131).

FET-Specific Menus

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

Topic **Page**

D.1 Menus

D.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 (by opening a pulldown menu next to the Run icon on top of the Debug View) is selected, 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 Debugging #9.

D.1.2 Run → Connect Target

Regains control of the device when ticked.

D.1.3 Run → Advanced → Make Device Secure

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

D.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/STEP INTO). Can only be changed when the debugger is inactive. See FAQ Debugging #12.

D.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.

D.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 2-1). 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 CCS Version 4* (SLAA393).

D.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.

Device Specific Menus

E.1 MSP430L092

E.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 target configuration: Open the MSP430L092.CCXML file in your project, click Target Configuration in the Advanced Setup section, Advanced Target Configuration. The CPU Properties become visible after MSP430 is selected. Figure E-1 shows how to select the L092 mode and how to use the Loader Code (See Loader Code). Figure E-2 shows how to select the C092 mode.

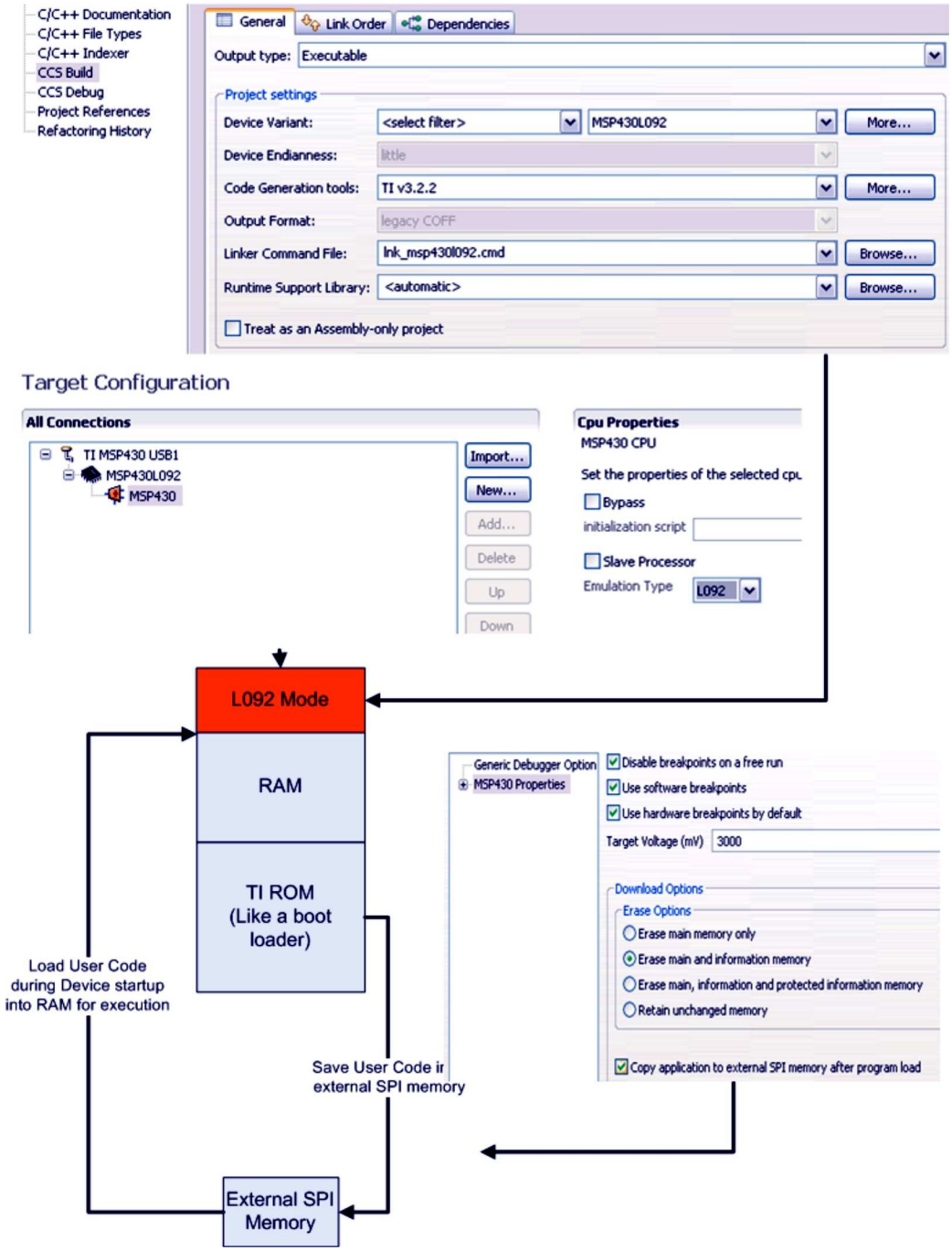

Figure E-1. MSP430L092 Modes

E.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 E-1).

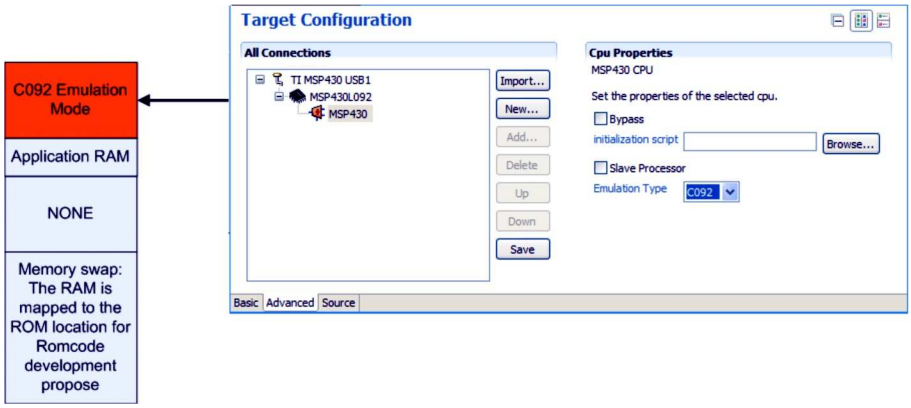

Figure E-2. MSP430L092 in C092 Emulation Mode

E.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 E-3 shows how to provide a HEX password in CCSv4 target configuration.

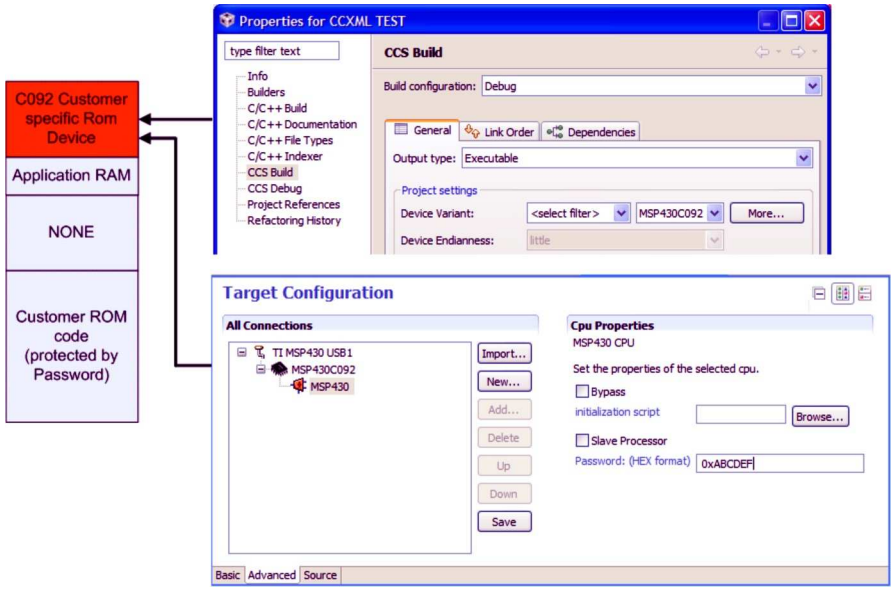

Figure E-3. MSP430C092 Password Access

E.2 MSP430F5xx/F6xx BSL Support

Most of the MSP430F5xx and 'F6xx 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 E-4).

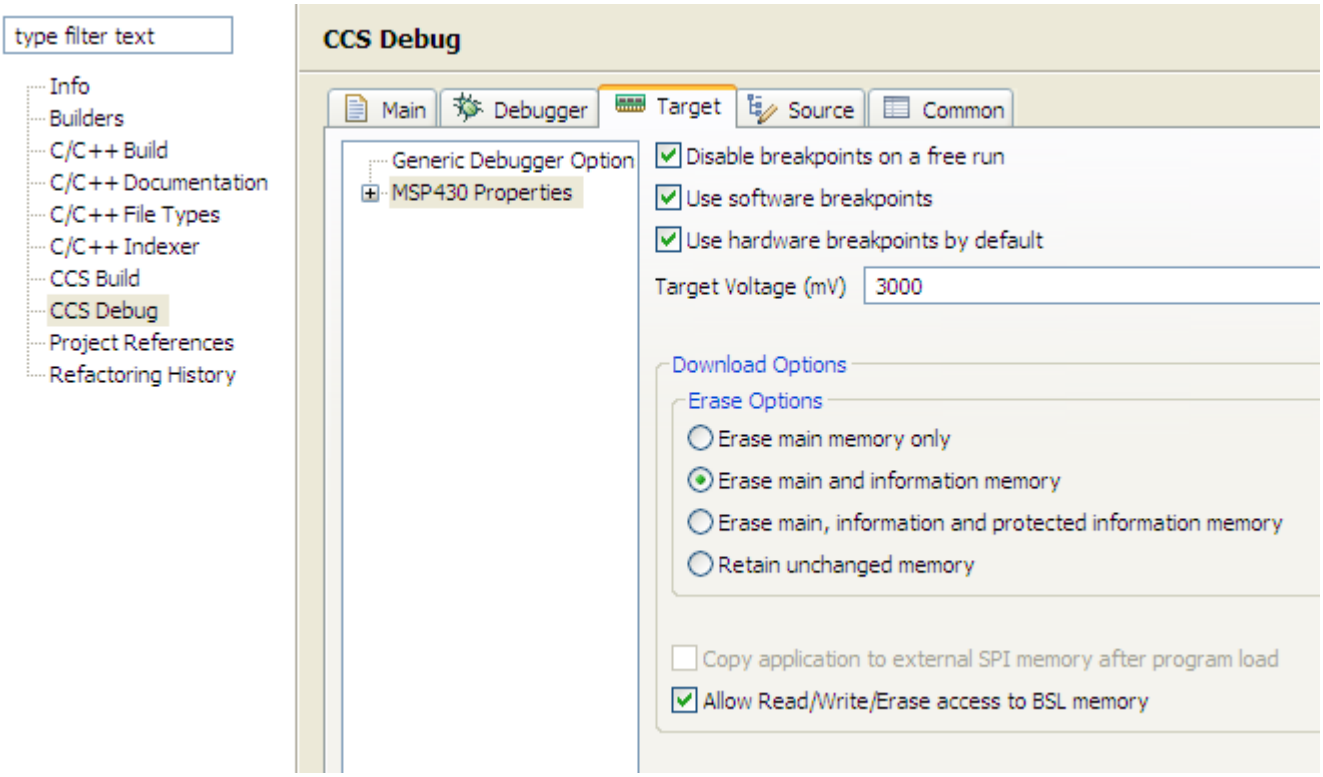

Figure E-4. Allow Access to BSL

E.3 MSP430F5xx/F6xx Password Protection

Selected MSP430F5xx and 'F6xx devices provide JTAG protection by a user password. When debugging such an 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 E-5).

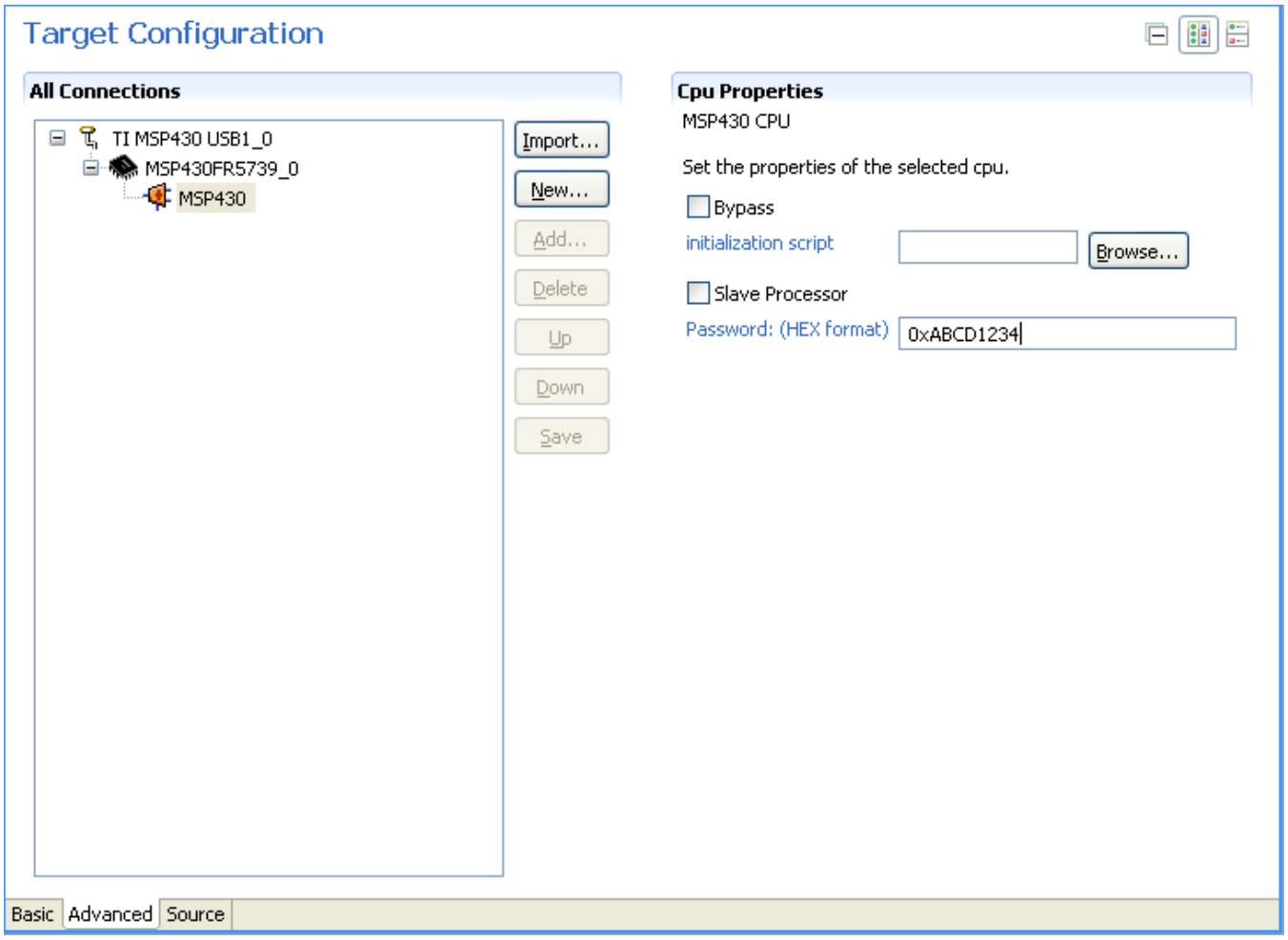

Figure E-5. MSP430 Password Access

E.4 LPMx.5 CCS Debug Support (MSP430FR57xx Only)

LPMx.5 is a new low-power mode in which the entry and exit is handled differently compared to other low-power modes. When used properly, LPMx.5 provides the lowest power consumption available on a device. To achieve this, entry to LPMx.5 disables the LDO of the PMM module, removing 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.

E.4.1 Debugging With LPMx.5

To enable the LPMx.5 debug feature, the Halt on device wake up (required for debugging LPMx.5 mode) checkbox must be enabled (see Figure E-6). To enable LPMx.5 debug, click Project Properties → Debug → MSP430 Properties → Halt on device wakeup (required for debugging LPMx.5 mode).

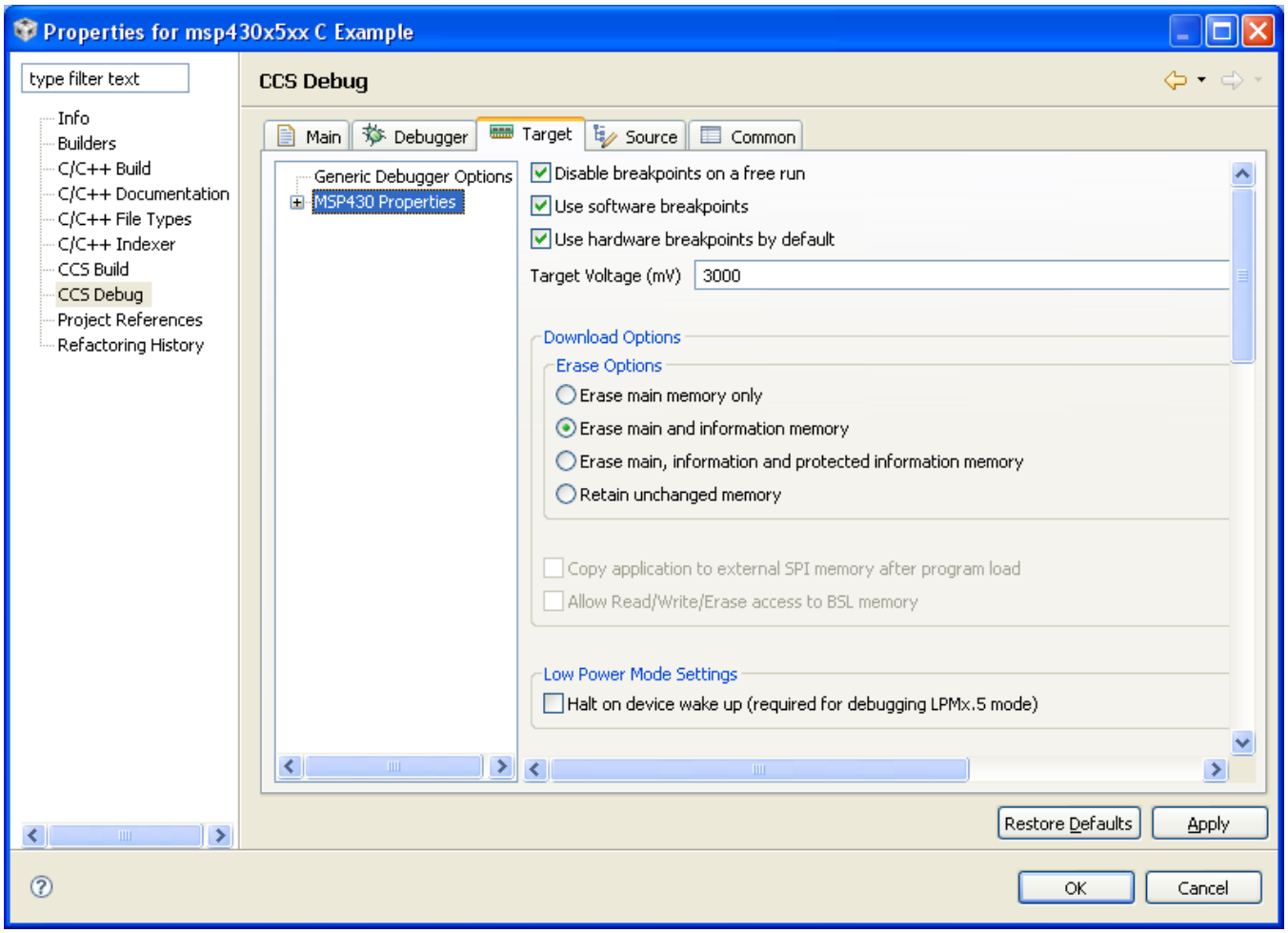

Figure E-6. Enabling LPMx.5 Debug Support

If the LPMx.5 debug mode is enabled, a notification is displayed in the debugger console log every time the target device enters and leaves LPMx.5 mode. Pressing the Halt or Reset button in CCS wakes the target device from LPMx.5 and stops it at code start. All breakpoints that were active before LPMx.5 are restored and reactivated automatically.

E.4.2 LPMx.5 Debug 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 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 device reset.

Revision History

Version	Changes/Comments
SLAU157T	Updated information throughout for Code Composer Studio v5.1. Added emulation features for CC430F512x, CC430F514x, CC430F614x.
SLAU157S	Added emulation features for MSP430F52xx, F533x, F643x, F67xx.
SLAU157R	Added emulation features for MSP430FR57xx, LPMx.5, and general MSP430 Password Protection instructions in Appendix E.
SLAU157Q	Added emulation features for MSP430F5310.
SLAU157P	Added emulation features for MSP430AFE253, MSP430F532x, and MSP430F534x.
SLAU157O	Added emulation features for MSP430BT5190, MSP430F530x, and MSP430F563x. Added BSL support for MSP430F5xx/F6xx in Appendix E and enhanced System Pre Init section in Appendix A.
SLAU157N	Added emulation features for MSP430L092/C092 and memory information in Appendix E.
SLAU157M	Added emulation features for MSP430G2xxx, MSP430F51x1, MSP430F51x2, MSP430F550x, MSP430F5510, MSP430F551x, MSP430F552x, MSP430F663x.
SLAU157L	Updated information throughout for Code Composer Studio v4.1. Added emulation features for MSP430F44x1, MSP430F461x, MSP430F461x1.
SLAU157K	Added emulation features for MSP430F54xxA, MSP430F55xx. Updated and extended Table 2-1 with architecture information.
SLAU157J	Updated information throughout for Code Composer Studio v4. Removed information on hardware. It was moved into the *MSP430 Hardware Tools User's Guide* (SLAU278)
SLAU157I	Added MSP-FET430U100A kit in Section 1.7 and MSP-TS430PZ100A target socket module schematic (Figure B-19) and PCB (Figure B-20). Added emulation features for CC430F513x, CC430F612x, CC430F613x, MSP430F41x2, MSP430F47x, MSP430FG479, and MSP430F471xx in Table 2-1. Updated MSP-TS430PN80 target socket module schematic (Figure B-15) with information on MSP430F47x and MSP430FG47x. Removed information throughout on MSP-FET430Pxx0 and MSP-FET430X110 kits.
SLAU157H	Updated information throughout for Code Composer Essentials v3.1.
SLAU157G	Added MSP-FET430U5x100 kit and MSP-TS430PZ5x100 target socket module schematic.
SLAU157F	Added crystal information to Section 1.7. Added overview of debug interfaces as Table 1-1. Added eZ430-F2013, T2012, and eZ430-RF2500. Updated information throughout for Code Composer Essentials v3.
SLAU157E	Added MSP-TS430PW28 target socket module, schematic (Figure B-5) and PCB (Figure B-6). Updated MSP-FET430U28 kit content information (DW or PW package support) in Section 1.7. Added emulation features for MSP430F21x2 to Table 2-1. Updated MSP-TS430PW14 target socket module schematic (Figure B-1). Updated MSP-TS430DA38 target socket module schematic (Figure B-7).
SLAU157D	Added Section 1.12. Updated Table 2-1. Updated Appendix F.
SLAU157C	Updated Appendix F. Added emulation features for MSP430F22x2, MSP430F241x, MSP430F261x, MSP430FG42x0 and MSP430F43x in Table 2-1.
SLAU157B	Renamed MSP-FET430U40 to MSP-FET430U23x0. Replaced MSP-FET430U40 schematic and PCB figures with renamed MSP-FET430U23x0 figures. Added FAQ Hardware #2 in Section A.1. Added FAQ Debugging #4 in Section A.3.

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

EVALUATION BOARD/KIT IMPORTANT NOTICE

Texas Instruments (TI) provides the enclosed product(s) under the following conditions:

This evaluation board/kit is intended for use for **ENGINEERING DEVELOPMENT, DEMONSTRATION, OR EVALUATION PURPOSES ONLY** and is not considered by TI to be a finished end-product fit for general consumer use. Persons handling the product(s) must have electronics training and observe good engineering practice standards. As such, the goods being provided are not intended to be complete in terms of required design-, marketing-, and/or manufacturing-related protective considerations, including product safety and environmental measures typically found in end products that incorporate such semiconductor components or circuit boards. This evaluation board/kit does not fall within the scope of the European Union directives regarding electromagnetic compatibility, restricted substances (RoHS), recycling (WEEE), FCC, CE or UL, and therefore may not meet the technical requirements of these directives or other related directives.

Should this evaluation board/kit not meet the specifications indicated in the User's Guide, the board/kit may be returned within 30 days from the date of delivery for a full refund. THE FOREGOING WARRANTY IS THE EXCLUSIVE WARRANTY MADE BY SELLER TO BUYER AND IS IN LIEU OF ALL OTHER WARRANTIES, EXPRESSED, IMPLIED, OR STATUTORY, INCLUDING ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE.

The user assumes all responsibility and liability for proper and safe handling of the goods. Further, the user indemnifies TI from all claims arising from the handling or use of the goods. Due to the open construction of the product, it is the user's responsibility to take any and all appropriate precautions with regard to electrostatic discharge.

EXCEPT TO THE EXTENT OF THE INDEMNITY SET FORTH ABOVE, NEITHER PARTY SHALL BE LIABLE TO THE OTHER FOR ANY INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES.

TI currently deals with a variety of customers for products, and therefore our arrangement with the user **is not exclusive.**

TI assumes **no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein.**

Please read the User's Guide and, specifically, the Warnings and Restrictions notice in the User's Guide prior to handling the product. This notice contains important safety information about temperatures and voltages. For additional information on TI's environmental and/or safety programs, please contact the TI application engineer or visit www.ti.com/esh.

No license is granted under any patent right or other intellectual property right of TI covering or relating to any machine, process, or combination in which such TI products or services might be or are used.

FCC Warning

This evaluation board/kit is intended for use for **ENGINEERING DEVELOPMENT, DEMONSTRATION, OR EVALUATION PURPOSES ONLY** and is not considered by TI to be a finished end-product fit for general consumer use. It generates, uses, and can radiate radio frequency energy and has not been tested for compliance with the limits of computing devices pursuant to part 15 of FCC rules, which are designed to provide reasonable protection against radio frequency interference. Operation of this equipment in other environments may cause interference with radio communications, in which case the user at his own expense will be required to take whatever measures may be required to correct this interference.