User's Guide

MSP430 GCC Toolchain

ABSTRACT

This manual describes the setup and basic operation of the MSP430™ GCC toolchain and the software development environment.

Table of Contents

Read This First 3
- How to Use This User's Guide 3
- Related Documentation 3
- If You Need Assistance 4
- Trademarks 4

1 Introduction 5

2 Installing MSP430 GCC Toolchain 5
- 2.1 Installing MSP430 GCC in CCS Releases Before v7.2 6
- 2.2 Installing MSP430 GCC as Stand-Alone Package 8

3 Using MSP430 GCC Within CCS 9
- 3.1 Create New Project 9
- 3.2 Debug Using MSP-FET, MSPFET430UIF, eZ-FET, eZ430 10
- 3.3 Build Options for MSP430 GCC 10
- 3.4 CCS Compared to MSP430 GCC 29

4 MSP430 GCC Stand-Alone Package 29
- 4.1 MSP430 GCC Stand-Alone Package Folder Structure 30
- 4.2 Package Content 30
- 4.3 MSP430 GCC Options 31
- 4.4 MSP430 Built-in Functions 33
- 4.5 Using MSP430 GCC Support Files 34
- 4.6 Quick Start: Blink the LED 35
- 4.7 GDB Settings 37

5 MSP430 GCC Features 39
- 5.1 C/C++ Attributes 39
- 5.2 Hints for Reducing the Size of MSP430 GCC Programs 41
- 5.3 C Runtime Library (CRT) Startup Behavior 42
- 5.4 Using printf with MSP430 GCC 43
- 5.5 Link-time Optimization (LTO) 43
- 5.6 The __int20 Type and Pointers in the Large Memory Model 43

6 Building MSP430 GCC From Sources 44
- 6.1 Required Tools 44
- 6.2 Building MSP430 GCC (Mitto Systems Limited) 44
- 6.3 Building MSP430 GCC Stand-Alone Full Package 45

7 MSP430 GCC and MSPGCC 46
- 7.1 Calling Convention 46
- 7.2 Other Portions of the ABI 46

8 Appendix 47
- 8.1 GCC Intrinsic Support 47
- 8.2 NOP Instructions Required Between Interrupt State Changes 48

9 References 48

Revision History 48

Copyright © 2020 Texas Instruments Incorporated

List of Figures

Figure 2-1. MSP430 GCC With CCS Installer 6
Figure 2-2. MSP430 GCC With CCS Installer 6
Figure 2-3. Installing MSP430 GCC Through CCS Apps Center 7
Figure 2-4. MSP430 GCC Stand-Alone Package Installer 8
Figure 2-5. MSP430 GCC Stand-Alone Package Installation Directory 8
Figure 3-1. Creating New CCS Project Using MSP430 GCC 9
Figure 3-2. CCS Project Using MSP430 GCC 10
Figure 3-3. MSP430 GCC Settings 11
Figure 3-4. MSP430 GCC Settings: Runtime 12
Figure 3-5. MSP430 GCC Settings: Symbols 13
Figure 3-6. MSP430 GCC Settings: Directories 14
Figure 3-7. MSP430 GCC Settings: Optimization 15
Figure 3-8. MSP430 GCC Settings: Preprocessor 16
Figure 3-9. MSP430 GCC Settings: Assembler 17
Figure 3-10. MSP430 GCC Settings: Debugging 18
Figure 3-11. MSP430 GCC Settings: Diagnostic Options 19
Figure 3-12. MSP430 GCC Settings: Miscellaneous 20
Figure 3-13. MSP430 GCC Linker Settings 21
Figure 3-14. MSP430 GCC Linker Basic Settings 22
Figure 3-15. MSP430 GCC Linker Libraries Settings 23
Figure 3-16. MSP430 GCC Linker Symbols Settings 24
Figure 3-17. MSP430 GCC Linker Miscellaneous Settings 25
Figure 3-18. MSP430 GCC GNU Objcopy Utility Settings 26
Figure 3-19. MSP430 GCC GNU Objcopy Utility General Options Settings 27
Figure 3-20. MSP430 GCC GNU Objcopy Utility Miscellaneous Settings 28

List of Tables

Table 1-1. MSP430 TI and GCC Toolchain Comparison 5
Table 3-1. MSP430 GCC Settings 11
Table 3-2. MSP430 GCC Settings: Runtime 12
Table 3-3. MSP430 GCC Settings: Symbols 13
Table 3-4. MSP430 GCC Settings: Directories 14
Table 3-5. MSP430 GCC Settings: Optimization 15
Table 3-6. MSP430 GCC Settings: Preprocessor 16
Table 3-7. MSP430 GCC Settings: Assembler 17
Table 3-8. MSP430 GCC Settings: Debugging 18
Table 3-9. MSP430 GCC Settings: Diagnostic Options 19
Table 3-10. MSP430 GCC Settings: Miscellaneous 20
Table 3-11. MSP430 GCC Linker Settings 21
Table 3-12. MSP430 GCC Linker Basic Settings 22
Table 3-13. MSP430 GCC Linker Libraries Settings 23
Table 3-14. MSP430 GCC Linker Symbols Settings 24
Table 3-15. MSP430 GCC Linker Miscellaneous Settings 25
Table 3-16. MSP430 GCC GNU Objcopy Utility Settings 26
Table 3-17. MSP430 GCC GNU Objcopy Utility General Options Settings 27
Table 3-18. MSP430 GCC GNU Objcopy Utility Miscellaneous Settings 28
Table 4-1. MSP430 GCC Stand-Alone Package 29
Table 4-2. MSP430 GCC Command Options 31
Table 4-3. MSP430 GCC Assembler Options 33
Table 4-4. MSP430 GCC Linker Options 33
Table 4-5. MSP430 Objdump Options 33

Copyright © 2020 Texas Instruments Incorporated

Read This First

How to Use This User's Guide

This manual describes only the setup and basic operation of the MSP430™ GCC toolchain and the software development environment. It does not fully describe the MSP430 GCC toolchain or MSP430 microcontrollers or the complete development software and hardware systems. For details on these items, see the appropriate documents listed in Section Related Documentation.

This manual applies to the use of MSP430 GCC as stand-alone package or within the Code Composer Studio™ (CCS) IDE v10.x and 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 or contact your local TI sales office.

Related Documentation

The primary sources of MSP430 information are the device-specific data sheets and user's guides. The MSP430 website contains the most recent version of these documents.

The GCC documentation can be found at http://www.gnu.org. All related information for the MSP430 GCC toolchain is available at http://www.ti.com/tool/msp430-gcc-opensource.

Documents that describe the Code Composer Studio tools (CCS IDE, assembler, C compiler, linker, and librarian) can be found at http://www.ti.com/tool/ccstudio. The TI Resource Explorer and the Tools forum on TI E2E™ provide additional help.

MSP430 GCC documentation

Using the GNU Compiler Collection, Richard M. Stallman (http://gcc.gnu.org/onlinedocs/gcc.pdf). Refer to the *MSP430 Options* section.

GDB: The GNU Project Debugger, Free Software Foundation, Inc. (https://sourceware.org/gdb/current/onlinedocs/)

GCC for MSP430™ Microcontrollers Quick Start Guide

Calling Convention and ABI Changes in MSP GCC

CCS documentation

MSP430™ Assembly Language Tools User's Guide

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

Code Composer Studio™ IDE for MSP430™ MCUs User's Guide

MSP430 development tools documentation

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™ Development Kit User's Guide

Advanced debugging using the enhanced emulation module (EEM) with Code Composer Studio IDE

MSP430 device data sheets

Copyright © 2020 Texas Instruments Incorporated

MSP430 device family user's guides

MSP430x1xx Family User's Guide

MSP430x2xx Family User's Guide

MSP430x3xx Family User's Guide

MSP430F4xx Family User's Guide

MSP430F5xx and MSP430F6xx Family User's Guide

MSP430FR4xx and MSP430FR2xx Family User's Guide

MSP430FR57xx Family User's Guide

MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide

If You Need Assistance

Support for the MSP430 devices and the hardware development tools is provided by the TI Product Information Center (PIC). Contact information for the PIC can be found on the TI website. The TI E2E™ Community support forums provide open interaction with peer engineers, TI engineers, and other experts. Additional device-specific information can be found on the MSP430 website.

Trademarks

MSP430™, Code Composer Studio™, E2E™, eZ430-Chronos™, and LaunchPad™ are trademarks of Texas Instruments Incorporated.
Windows® is a registered trademark of Microsoft Corporation.
Linux® is a registered trademark of Linus Torvalds.
macOS® is a registered trademark of Apple Inc.
All other trademarks are the property of their respective owners.

Copyright © 2020 Texas Instruments Incorporated

1 Introduction

The MSP430 GCC toolchain uses the MSP430 ABI and is compatible with the TI toolchain. This free GCC toolchain supports all MSP430 devices and has no code size limit. In addition, this toolchain can be used as a stand-alone package or used within Code Composer Studio (CCS) IDE v6.0 or later. Get started today in Windows, Linux, or macOS environments.

Table 1-1 compares the MSP430 TI and GCC toolchain.

Table 1-1. MSP430 TI and GCC Toolchain Comparison

	Proprietary TI Code Generation Tools	MSP430 GCC Toolchain	MSPGCC Toolchain
Code Size and Performance	✓✓✓	✓	✓
ABI	TI	TI	Community
Integrated in CCS	✓	✓(1)	✗
Stand-alone	✗	✓	✓
Support	TI	TI	Community
Cost Free	✓	✓	✓

(1) The combination of CCS and GCC is completely free of charge with no code size limit.

The MSP430 GCC supports the following:

- MSP430 CPU 16-bit architecture
- MSP430 CPUX 20-bit architecture
- MSP430 CPUXv2 20-bit architecture
- Code and data placement in the lower (<64K) and upper (>64K) memory areas and across the memory boundary
- The hardware multiplier of the MSP430 microcontrollers

This manual describes the use of the MSP430 GCC toolchain with the MSP430 ultra-low-power microcontrollers. The MSP430 GCC toolchain can be used within CCS version 6.0 or later, or it can be used as a stand-alone package. The toolchain supports Windows®, Linux®, and macOS® operating systems. This manual describes only CCS for Windows operating systems. The versions of CCS for Linux and macOS operating systems are similar and, therefore, are not described separately.

2 Installing MSP430 GCC Toolchain

MSP430 GCC supports Windows, Linux, and macOS:

- Windows 7 32 bit or 64 bit
- Windows 8 32 bit or 64 bit
- Windows 10 32 bit or 64 bit
- Linux 32 bit or 64 bit
- macOS 64 bit

You can install the MSP430 GCC using any of the following methods:

- MSP430 GCC toolchain is installed by default by CCS v7.2 and higher.
- In CCS releases prior to v7.2, the MSP430 GCC (toolchain only) is available in the CCS Apps Center. The corresponding MSP430 GCC support files (header and linkers) are downloaded with a standard MSP430 emulation package. For details, see Section 2.1.
- MSP430 GCC can be also downloaded as stand-alone package. For details, see Section 2.2.

Copyright © 2020 Texas Instruments Incorporated

2.1 Installing MSP430 GCC in CCS Releases Before v7.2

The MSP430 GCC toolchain can be installed in CCS v6.0 or higher in two ways: either when CCS is installed or as an add-on to an existing CCS installation.

1. During the install process of CCS, select the MSP430 GCC toolchain to be installed as an "add-on" (see Figure 2-1). MSP430 GCC is installed the first time you run CCS (see Figure 2-2).

Figure 2-1. MSP430 GCC With CCS Installer

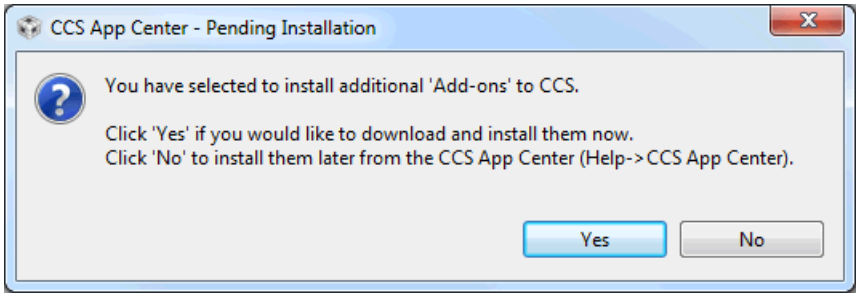

Figure 2-2. MSP430 GCC With CCS Installer

Copyright © 2020 Texas Instruments Incorporated

2. If CCS is already installed without MSP430 GCC, MSP430 GCC can be added at a later time through the CCS Apps Center (see Figure 2-3).
   a. Go to the menu **View** → **CCS App Center**.
   b. Select MSP430 GCC
   c. Click the **Install Software** button to start the installation.

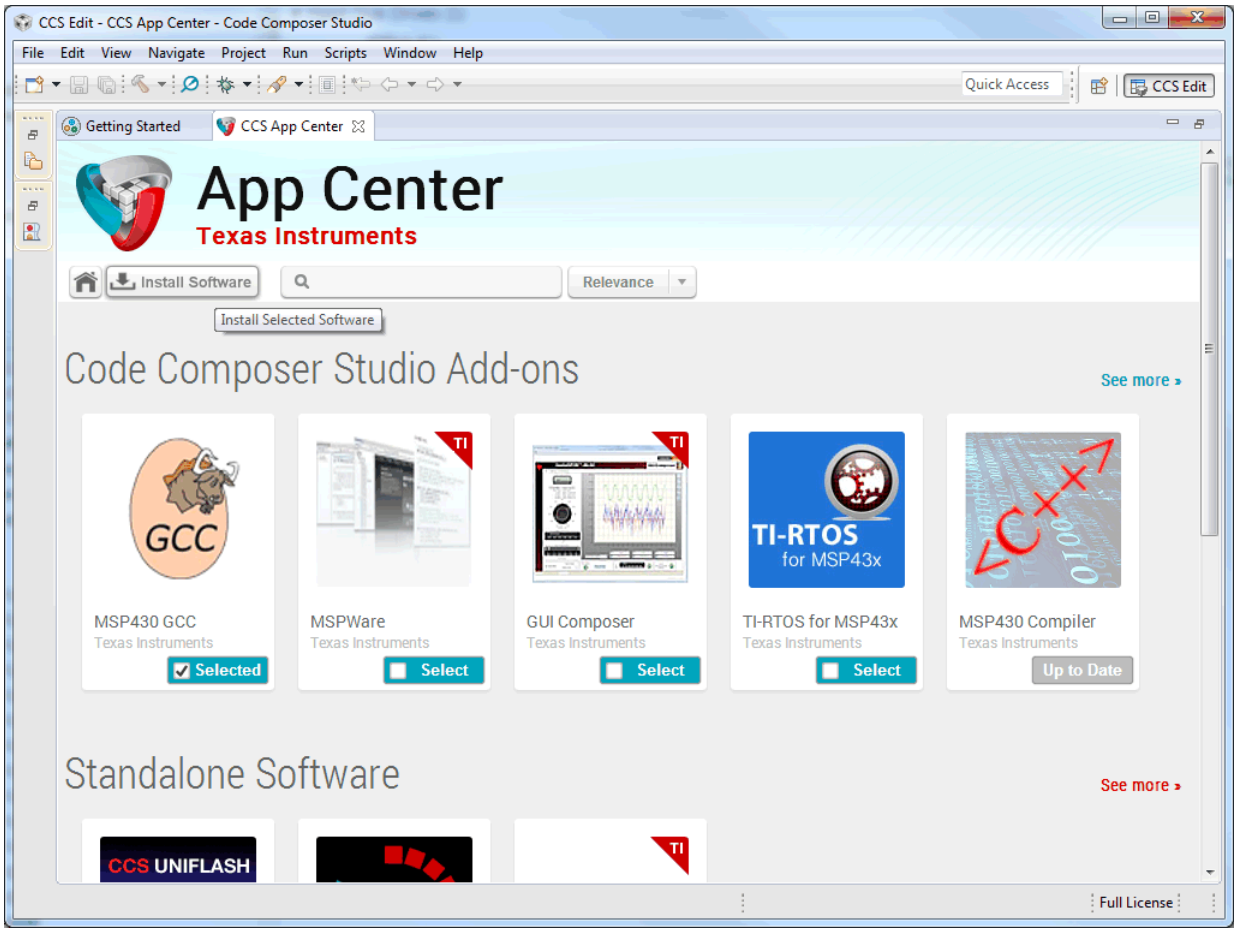

Figure 2-3. Installing MSP430 GCC Through CCS Apps Center

3. The GCC toolchain is installed to the following directory in the CCS installation: ccsv#\tools\compiler \gcc_msp430_x.x.x (where xxx denotes the version number).

Copyright © 2020 Texas Instruments Incorporated

2.2 Installing MSP430 GCC as Stand-Alone Package

The MSP430 GCC full stand-alone package can be downloaded from the TI website for all supported operating systems. The MSP430 GCC stand-alone package contains the toolchain, device support files, debug stack, and USB drivers.

To install the package:

1. Download the corresponding package installer and run it (see Figure 2-4).

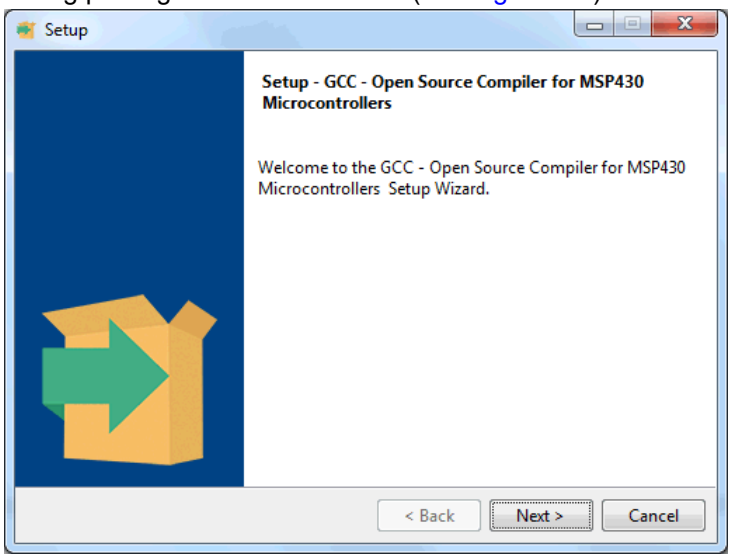

Figure 2-4. MSP430 GCC Stand-Alone Package Installer

2. Select the install directory and click **Next** (see Figure 2-5).

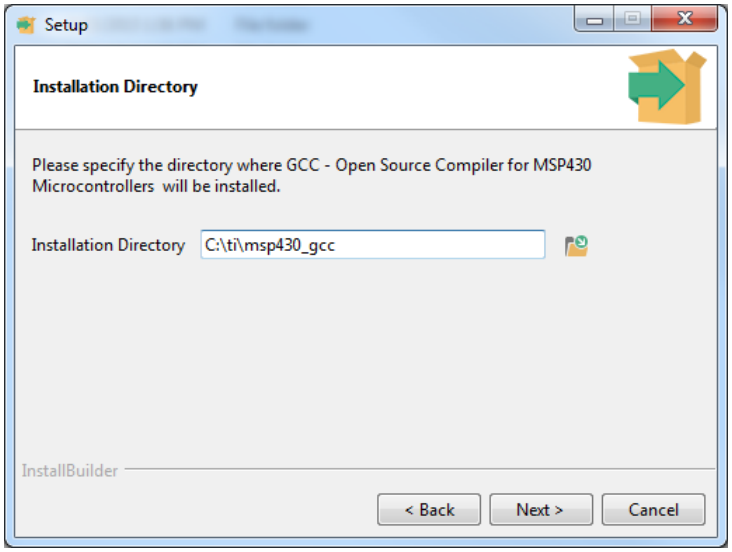

Figure 2-5. MSP430 GCC Stand-Alone Package Installation Directory

Note

For the Linux installer, apply **sudo chmod +x** <installer> before executing the package.

Copyright © 2020 Texas Instruments Incorporated

3 Using MSP430 GCC Within CCS

3.1 Create New Project

This section describes the step-by-step instructions to create an assembly or C project from scratch and to download and run an application on the MSP430 MCU using the MSP430 GCC toolchain. Also, the CCS Help presents a more detailed information of the process.

1. Start CCS (**Start** → **All Programs** → **Texas Instruments** → **Code Composer Studio** → **Code Composer Studio**).
2. Create a new project (**File** → **New** → **CCS Project**). Select the appropriate MSP430 device variant in the Target field and enter the name for the project.
3. Select **GNU v7.3.0.9 (Mitto Systems Limited)** for Compiler version (or any newer version).
4. In the Project template and examples section, select **Empty Project (with main.c)**. For assembly-only projects, select **Empty Project**.

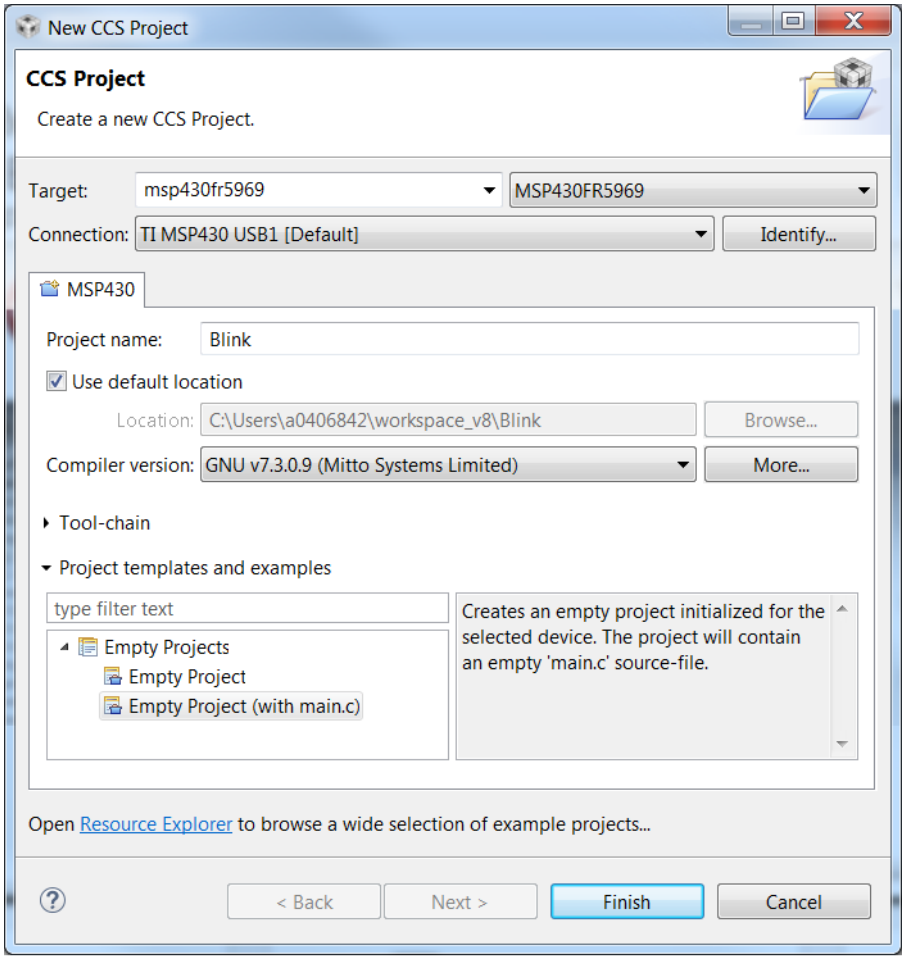

Figure 3-1. Creating New CCS Project Using MSP430 GCC

5. If you are 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.
6. For C projects, the setup is complete now.
7. Click **Finish** to create a new project that is then visible in the Project Explorer view.

   Notice that the project contains a .ld file (appropriate for the target selected). This is the linker script that contains the memory layout and section allocation. This file is the equivalent of the TI linker command file (.cmd) used by TI MSP430 Compiler and Linker.

Copyright © 2020 Texas Instruments Incorporated

8. Enter the program code into the main.c file.

   To use an existing source file for the 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.

   Now add the necessary source files to the project and build. Similar to TI tools, additional compiler and linker options can be set from **Project Properties**.
9. Build the project (**Project** → **Build Project**).

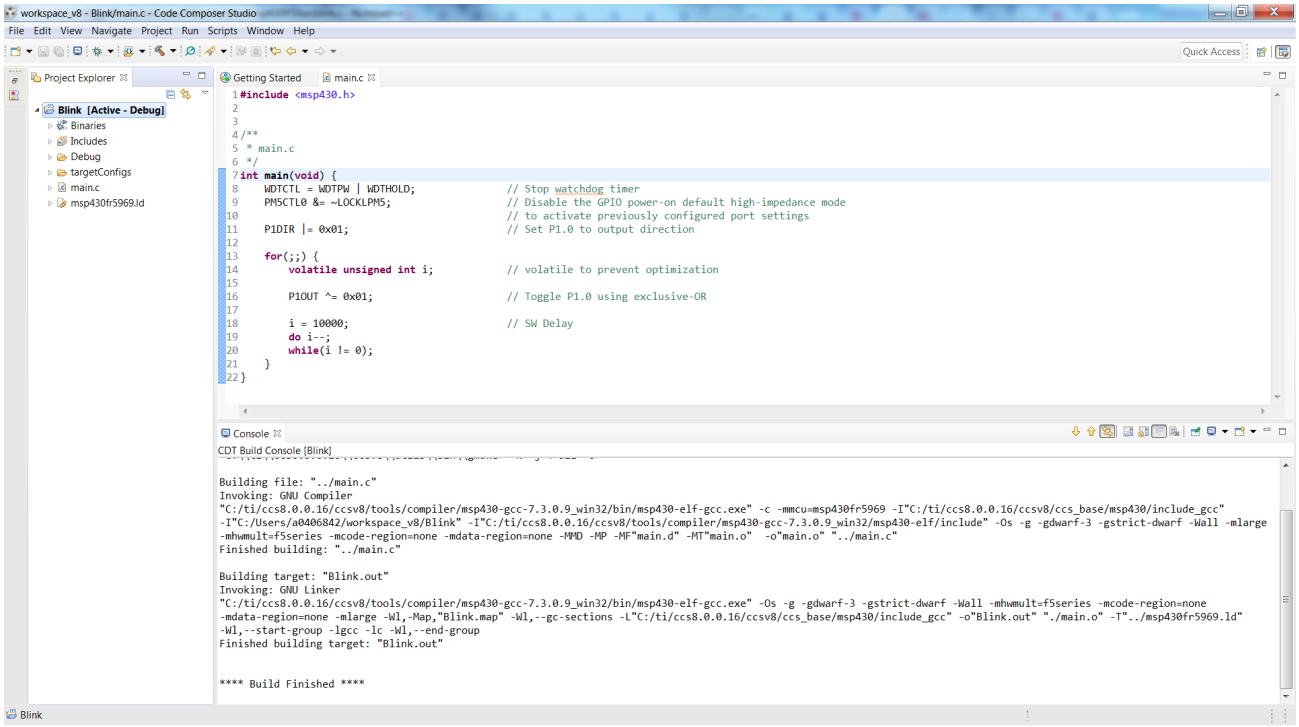

Figure 3-2. CCS Project Using MSP430 GCC

10. 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.
11. Click **Run** → **Resume (F8)** to start the application.
12. Click **Run** → **Terminate** to stop the application and to exit the debugger. CCS automatically returns to the C/C++ view (code editor).

3.2 Debug Using MSP-FET, MSPFET430UIF, eZ-FET, eZ430

MSP430 devices can be debugged in CCS using MSP-FET, MSPFET430UIF, eZ-FET, and eZ430 debuggers. For more details, refer to the *Code Composer Studio™ IDE for MSP430™ MCUs User's Guide* .

3.3 Build Options for MSP430 GCC

The settings required to configure the GCC are numerous and detailed and are not all described here. Most projects can be compiled and debugged with default factory settings.

To access the project settings for the active project, click **Project** → **Properties**.

The following project settings are common:

- Specify the target device for debug session (**Project** → **Properties** → **General** → **Device** → **Variant**). The corresponding Linker Command File and Runtime Support Library are selected automatically.

Copyright © 2020 Texas Instruments Incorporated

- To debug a C project more easily, disable optimization (-O0) or use -Og, which enables only those optimizations that do not interfere with debugging. The -Og option reduces code size and improves performance compared to -O0.
- Specify the search paths for the C preprocessor (**Project → Properties → Build → GNU Compiler → Directories → Include Paths (-I)**).
- Specify the search paths for any libraries being used (**Project → Properties → Build → GNU Linker → Libraries → Library search path (-L, --library-path)**).
- Specify the debugger interface (**Project → Properties → General → Device → Connection**). Select 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 work as expected stand-alone as the application on the device still contains the software breakpoint instructions.

3.3.1 GNU Compiler

Figure 3-3 shows the MSP430 GCC settings window.

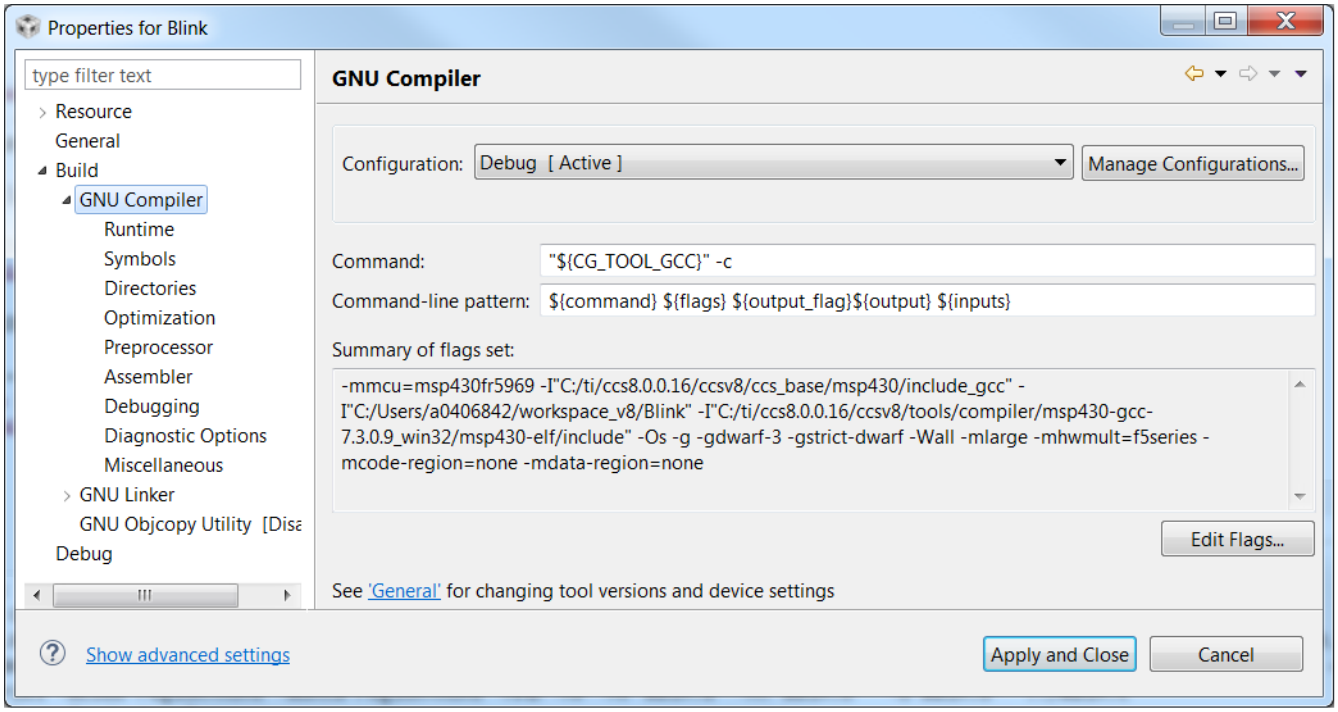

Figure 3-3. MSP430 GCC Settings

Table 3-1 describes the options that are available for MSP430 GCC Settings.

Table 3-1. MSP430 GCC Settings

Option	Description
Command	Compiler location
Command-line pattern	Command line parameters
Summary of flags set	Command line with which the compiler is called. Displays all the flags passed to the compiler.

Copyright © 2020 Texas Instruments Incorporated

3.3.2 GNU Compiler: Runtime

Figure 3-4 shows the MSP430 GCC Runtime settings window.

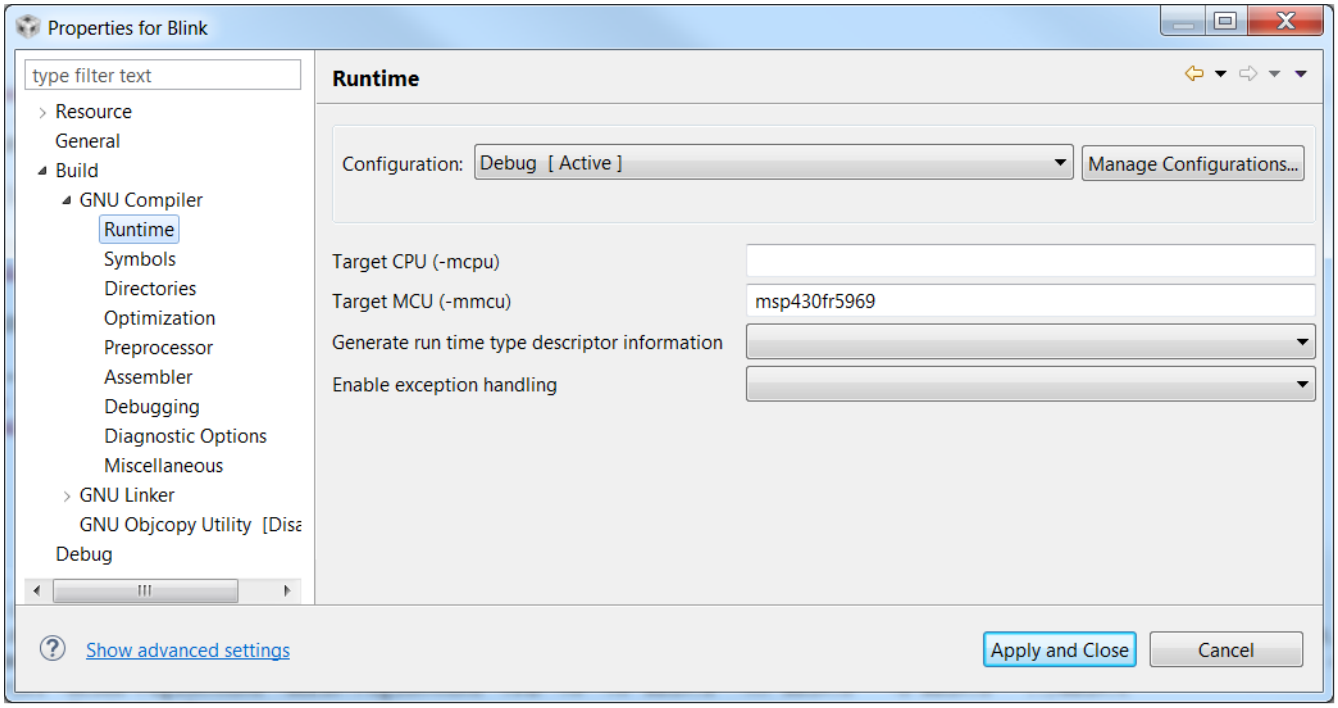

Figure 3-4. MSP430 GCC Settings: Runtime

Table 3-2 describes the options that are available for MSP430 GCC Runtime settings.

Table 3-2. MSP430 GCC Settings: Runtime

Option	Description
Target CPU (-mcpu)	Specifies the Instruction Set Architecture (ISA) to use. Accepted values are msp430, msp430x, and msp430xv2. This option is deprecated. The '-mmcu=' option should be used to select the ISA.
Target MCU (-mmcu)	Select the MCU to target. This is used to create a C preprocessor symbol based on the MCU name, converted to upper case and prefixed and postfixed with__. This in turn is used by the msp430.h header file to select an MCU-specific supplementary header file. The option also sets the ISA to use. If the MCU name is one that is known to only support the 430 ISA then that is selected, otherwise the 430X ISA is selected. A generic MCU name of msp430 can also be used to select the 430 ISA. Similarly, the generic msp430x MCU name selects the 430X ISA. In addition, an MCU-specific linker script is added to the linker command line. The script's name is the name of the MCU with ".ld" appended. Thus, specifying '-mmcu=xxx' on the gcc command line defines the C preprocessor symbol __XXX__ and causes the linker to search for a script called 'xxx.ld'. This option is also passed to the assembler.
Generate run time type descriptor information	Enable or disable generation of information about every class with virtual functions for use by the C++ runtime type identification features. • On (-frtti) • Off (-fno-rtti)
Enable exception handling	Enable or disable exception handling. Generates extra code needed to propagate exceptions. • On (-fexceptions) • Off (-fno-exceptions)

Copyright © 2020 Texas Instruments Incorporated

3.3.3 GNU Compiler: Symbols

Figure 3-5 shows the MSP430 GCC Symbols settings window.

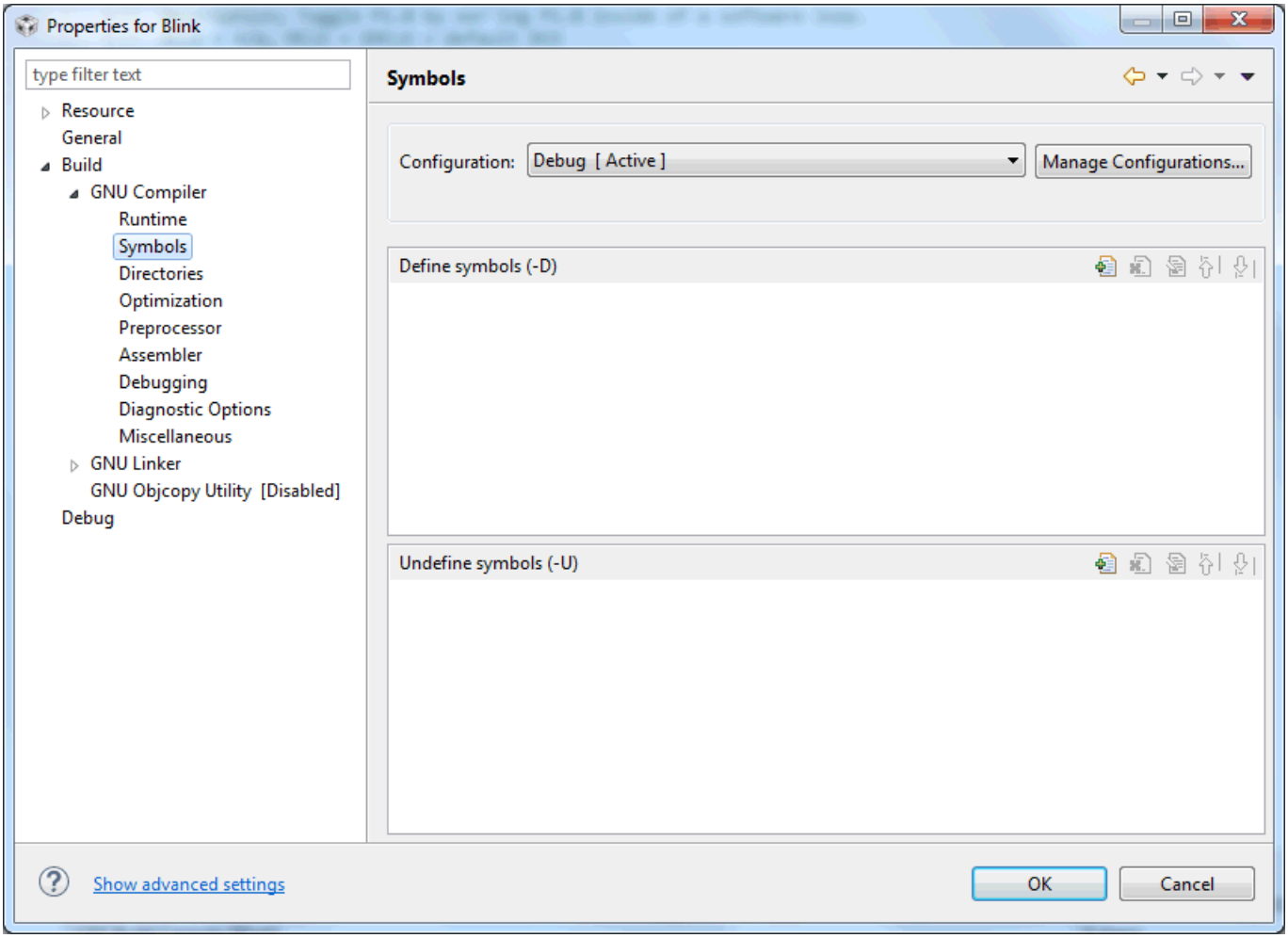

Figure 3-5. MSP430 GCC Settings: Symbols

Table 3-3 describes the options that are available for MSP430 GCC Symbols settings.

Table 3-3. MSP430 GCC Settings: Symbols

Option	Description
Define symbols (-D)	-D name Predefine name as a macro. -D name=definition Predefine name as a macro, with definition 1.
Undefine symbols (-U)	-U name Cancel any previous definition of name, either built-in or provided with a -D option.

Copyright © 2020 Texas Instruments Incorporated

3.3.4 GNU Compiler: Directories

Figure 3-6 shows the MSP430 GCC Directories settings window.

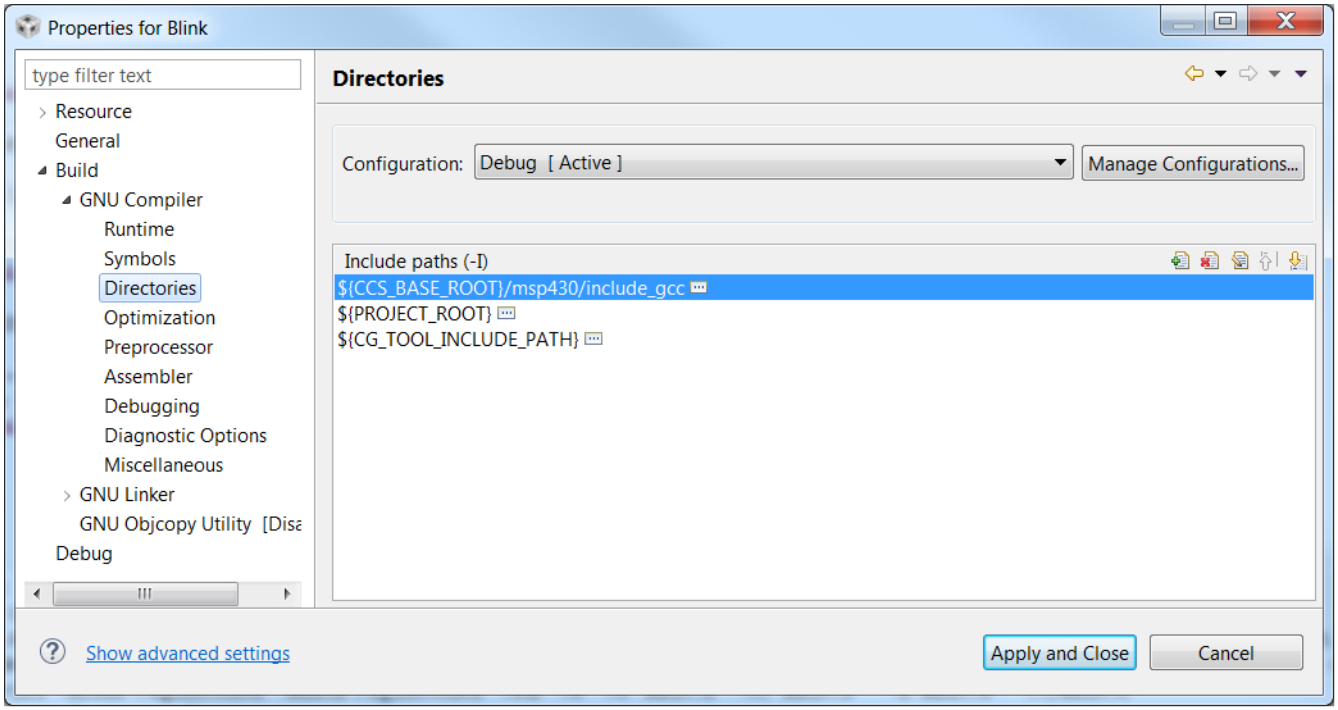

Figure 3-6. MSP430 GCC Settings: Directories

Table 3-4 describes the options that are available for MSP430 GCC Directories settings.

Table 3-4. MSP430 GCC Settings: Directories

Option	Description
Include paths (-I)	Add the directory to the list of directories to be searched for header files.

Copyright © 2020 Texas Instruments Incorporated

3.3.5 GNU Compiler: Optimization

Figure 3-7 shows the MSP430 GCC Optimization settings window.

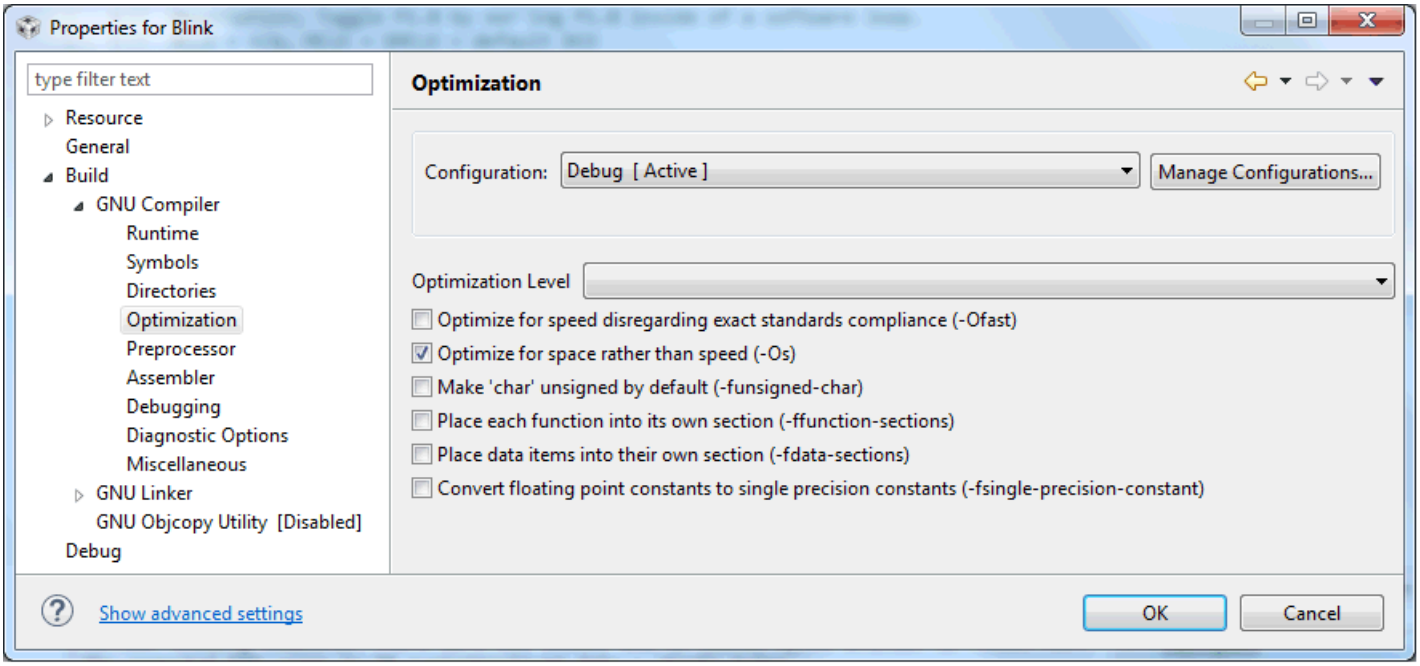

Figure 3-7. MSP430 GCC Settings: Optimization

Table 3-5 describes the options that are available for MSP430 GCC Optimization settings.

Table 3-5. MSP430 GCC Settings: Optimization

Option	Description
Optimization Level	Specifies the optimizations that the compiler applies to the generated object code. The options available are: • None (O0): Disable optimizations. This setting is equivalent to specifying the -O0 command-line option. The compiler generates unoptimized linear assembly language code. • Optimize (O1): The compiler performs all targets independent (that is, nonparallelized) optimizations, such as function inlining. This setting is equivalent to specifying the -O1 command-line option. The compiler omits all target-specific optimizations and generates linear assembly language code. • Optimize more (O2): The compiler performs all optimizations (both target-independent and target-specific). This setting is equivalent to specifying the -O2 command-line option. The compiler outputs optimized nonlinear parallelized assembly language code. • Optimize most (O3): The compiler performs all the level 2 optimizations, then the low-level optimizer performs global-algorithm register allocation. This setting is equivalent to specifying the -O3 command-line option. At this optimization level, the compiler generates code that is usually faster than the code generated from level 2 optimizations. • Optimize for space rather than speed (-Os): Enables all -O2 optimizations that do not typically increase code size. The -Os option also performs further optimizations designed to reduce code size. • Optimize for speed disregarding exact standards compliance (-Ofast): Enables all -O3 optimizations. The -Ofast option also enables optimizations that are not valid for all standard-compliant programs, such as -ffast-math.
Make 'char' unsigned by default (-funsigned-char)	Enable this option to ensure that the char is signed.
Place each function into its own section (-ffunction-sections)	Enable this option to place each function in its own section in the output file.
Place data items into their own section (-fdata-sections)	Enable this option to place each data item in its own section in the output file.

Copyright © 2020 Texas Instruments Incorporated

Option	Description
Convert floating point constants to single precision constants (-fsingle-precision-constant)	Treat floating-point constants as single precision instead of implicitly converting them to double-precision constants.

Note

Use the -ffunction-sections and -fdata-sections options in conjunction with the --gc-sections linker option to reduce code size by allowing the linker to remove unused sections.

3.3.6 GNU Compiler: Preprocessor

Figure 3-8 shows the MSP430 GCC Preprocessor settings window.

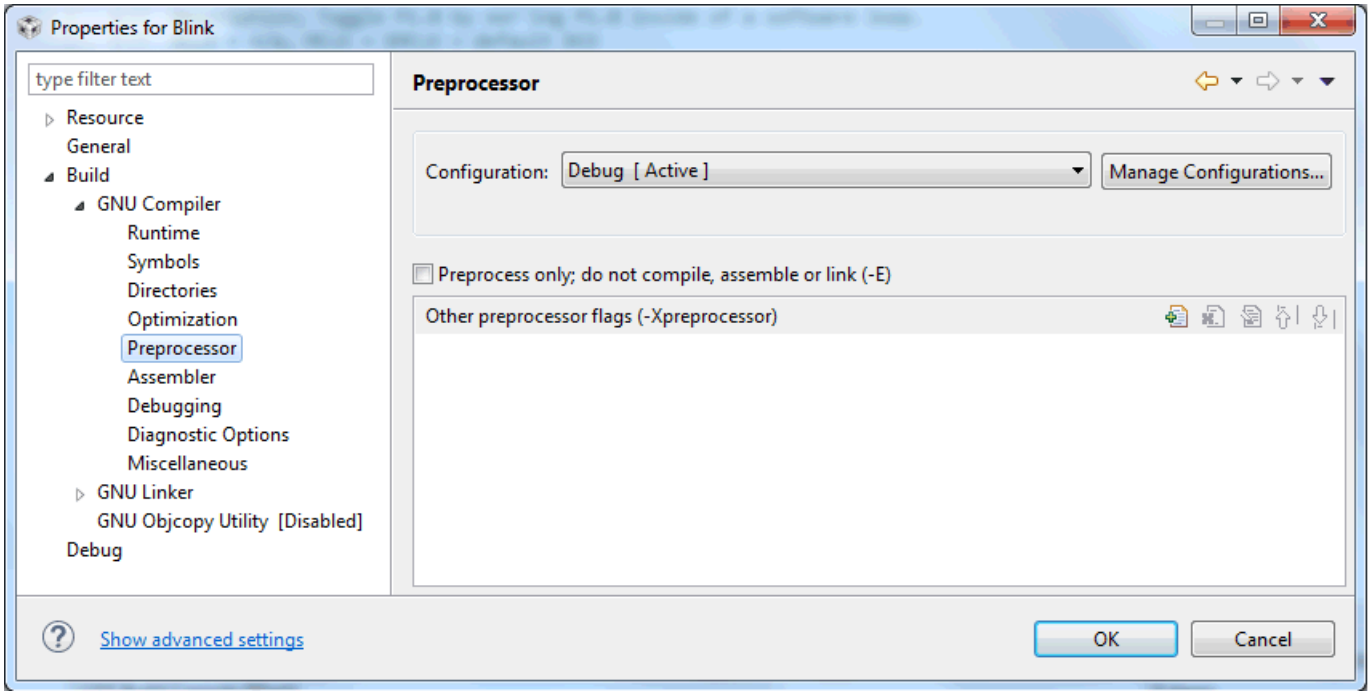

Figure 3-8. MSP430 GCC Settings: Preprocessor

Table 3-6 describes the options that are available for MSP430 GCC Preprocessor settings.

Table 3-6. MSP430 GCC Settings: Preprocessor

Option	Description
Preprocess only; do not compile, assemble, or link (-E)	Enable this option to preprocess only without compiling or assembling or linking.
Other preprocessor flags (-Xpreprocessor)	Use this to supply system-specific preprocessor options that GCC does not recognize. To pass an option that takes an argument, use -Xpreprocessor twice, once for the option and once for the argument.

Copyright © 2020 Texas Instruments Incorporated

3.3.7 GNU Compiler: Assembler

Figure 3-9 shows the MSP430 GCC Assembler settings window.

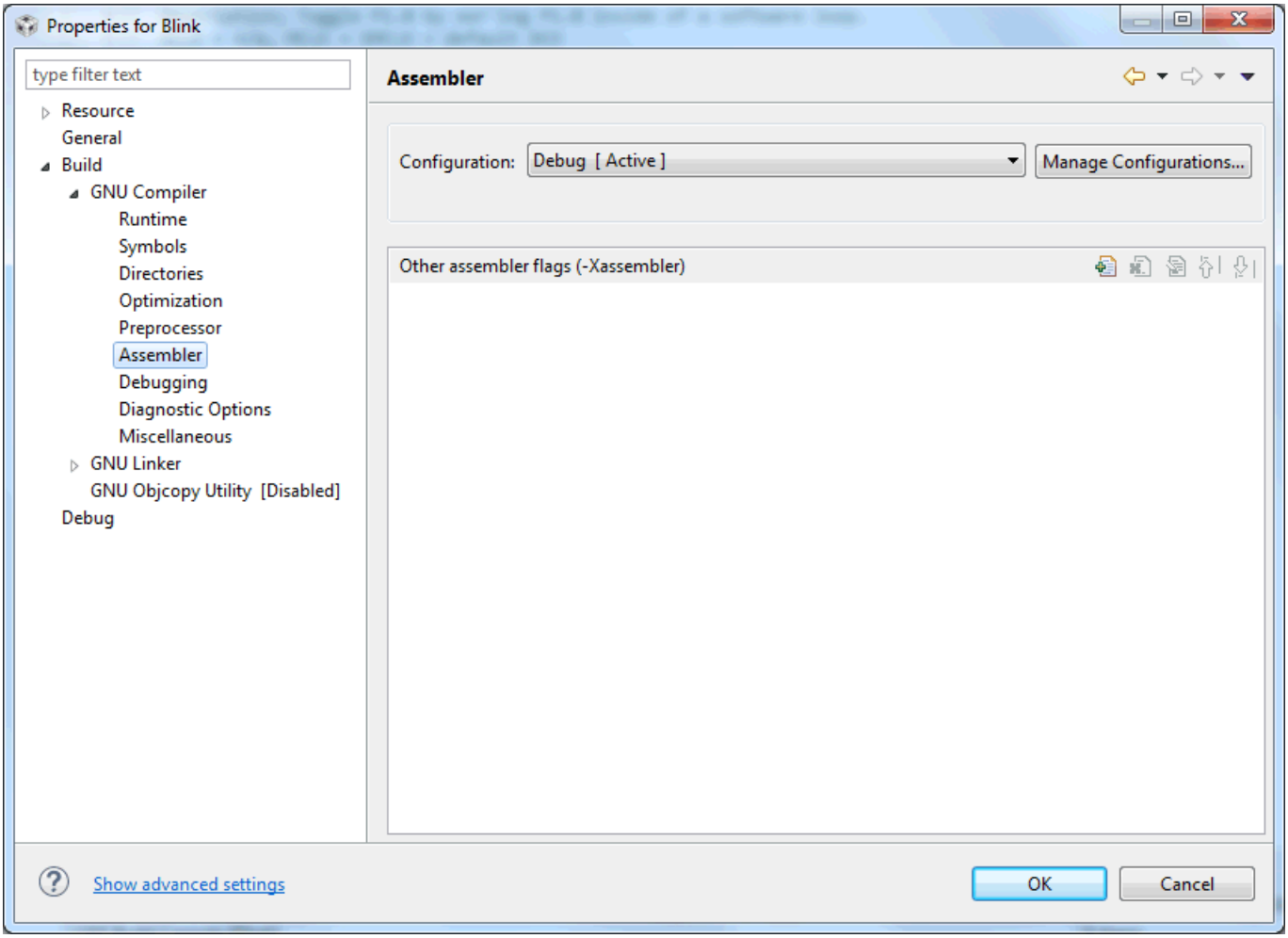

Figure 3-9. MSP430 GCC Settings: Assembler

Table 3-7 describes the options that are available for MSP430 GCC Assembler settings.

Table 3-7. MSP430 GCC Settings: Assembler

Option	Description
Other assembler flags (-Xassembler)	Specifies individual flag based on the user requirements.

Copyright © 2020 Texas Instruments Incorporated

3.3.8 GNU Compiler: Debugging

Figure 3-10 shows the MSP430 GCC Debugging settings window.

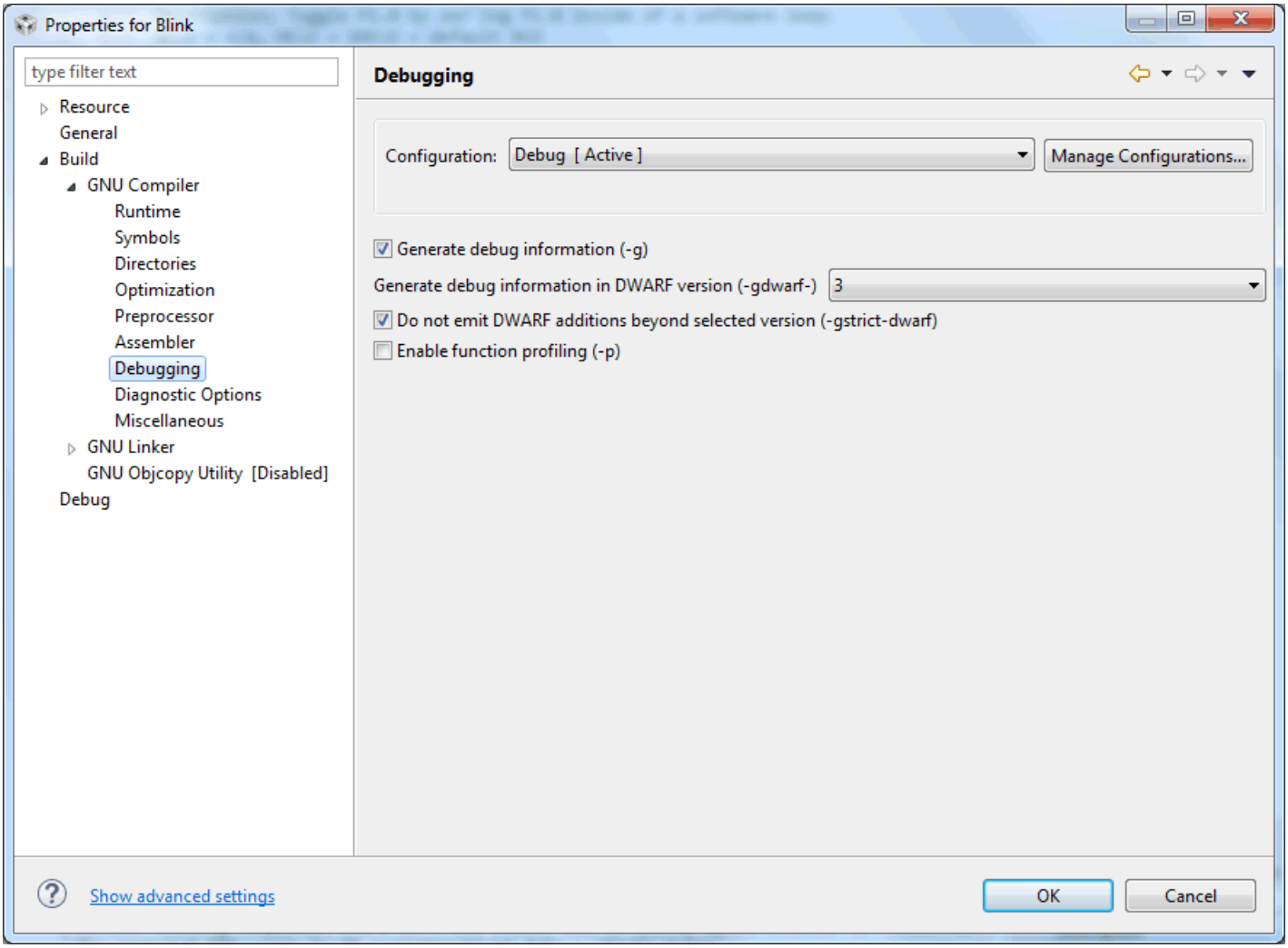

Figure 3-10. MSP430 GCC Settings: Debugging

Table 3-8 describes the options that are available for MSP430 GCC Debugging settings.

Table 3-8. MSP430 GCC Settings: Debugging

Option	Description
Generate debug information (-g)	Produce debugging information. This information is required by the GDB debugger.
Generate debug information in DWARF version (-gdwarf-)	Produce debugging information in DWARF format (if that is supported). The value of version may be 2, 3 or 4; the default version for most targets is 4.
Do not emit DWARF additions beyond selected version (-gstrict-dwarf)	Disallow using extensions of later DWARF standard version than selected with -gdwarf-version. On most targets using nonconflicting DWARF extensions from later standard versions is allowed.
Enable function profiling (-p)	Generate extra code to write profile information suitable for the analysis program. This option is required when compiling source files for which data is needed, and it is also required when linking.

Copyright © 2020 Texas Instruments Incorporated

3.3.9 GNU Compiler: Diagnostic Options

Figure 3-11 shows the MSP430 GCC Diagnostic Options settings window.

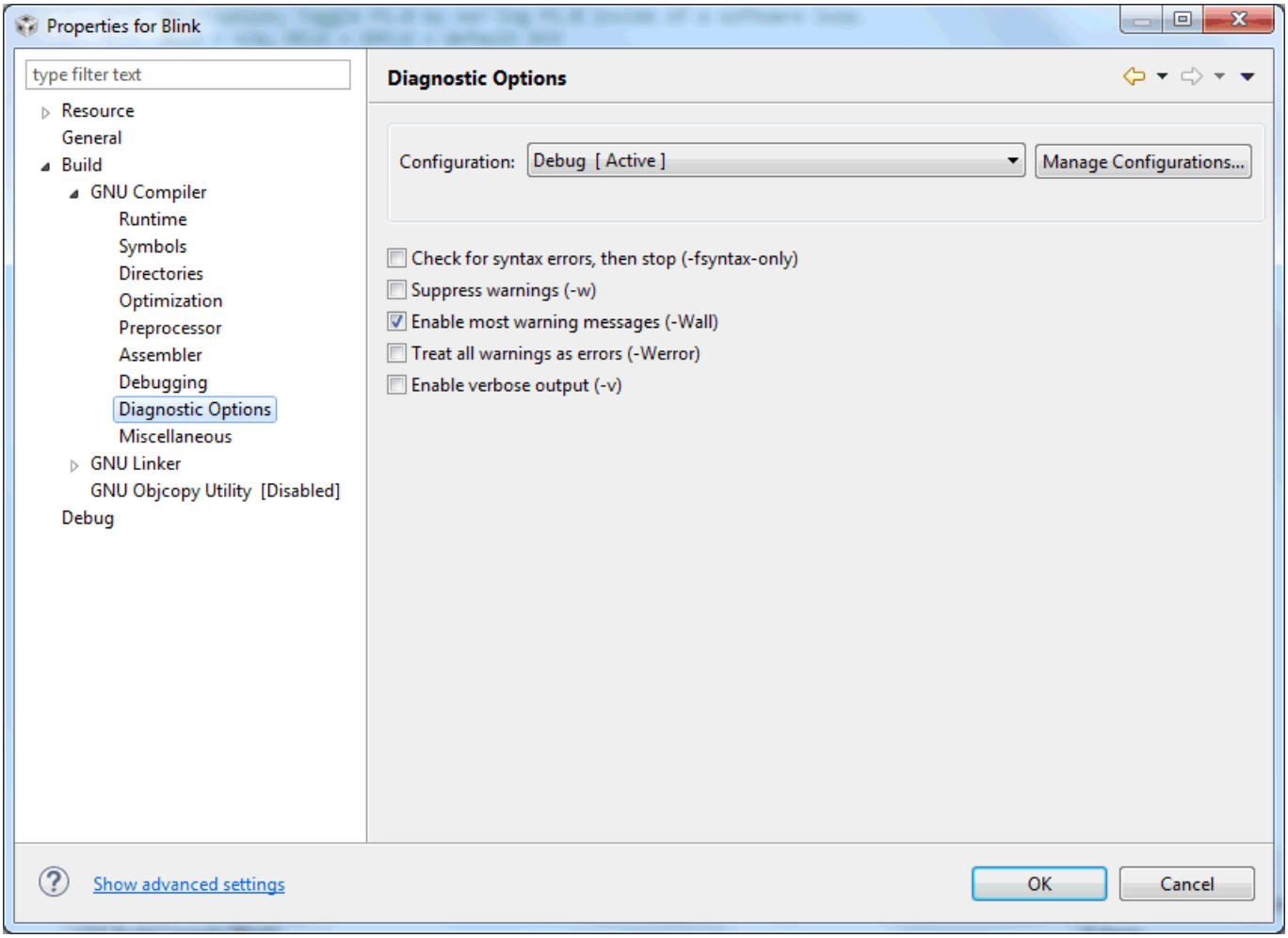

Figure 3-11. MSP430 GCC Settings: Diagnostic Options

Table 3-9 describes the options that are available for MSP430 GCC Diagnostic Options settings.

Table 3-9. MSP430 GCC Settings: Diagnostic Options

Option	Description
Check for syntax errors, then stop (-fsyntax-only)	Enable this option to check the syntax of the code and report any errors.
Suppress warnings (-w)	Inhibit all warning messages.
Enable most warning messages (-Wall)	Enable this option to enable all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.
Treat all warnings as errors (-Werror)	Enable this option to make all warnings into hard errors. Source code that triggers warnings is rejected.
Enable verbose output (-v)	Enable this option for the IDE to show each command line that it passes to the shell, along with all progress, error, warning, and informational messages that the tool emits. This setting is equivalent to specifying the –v command-line option. By default, this checkbox is clear. The IDE displays only error messages that the compiler emits. The IDE suppresses warning and informational messages.

Copyright © 2020 Texas Instruments Incorporated

3.3.10 GNU Compiler: Miscellaneous

Figure 3-12 shows the MSP430 GCC Miscellaneous settings window.

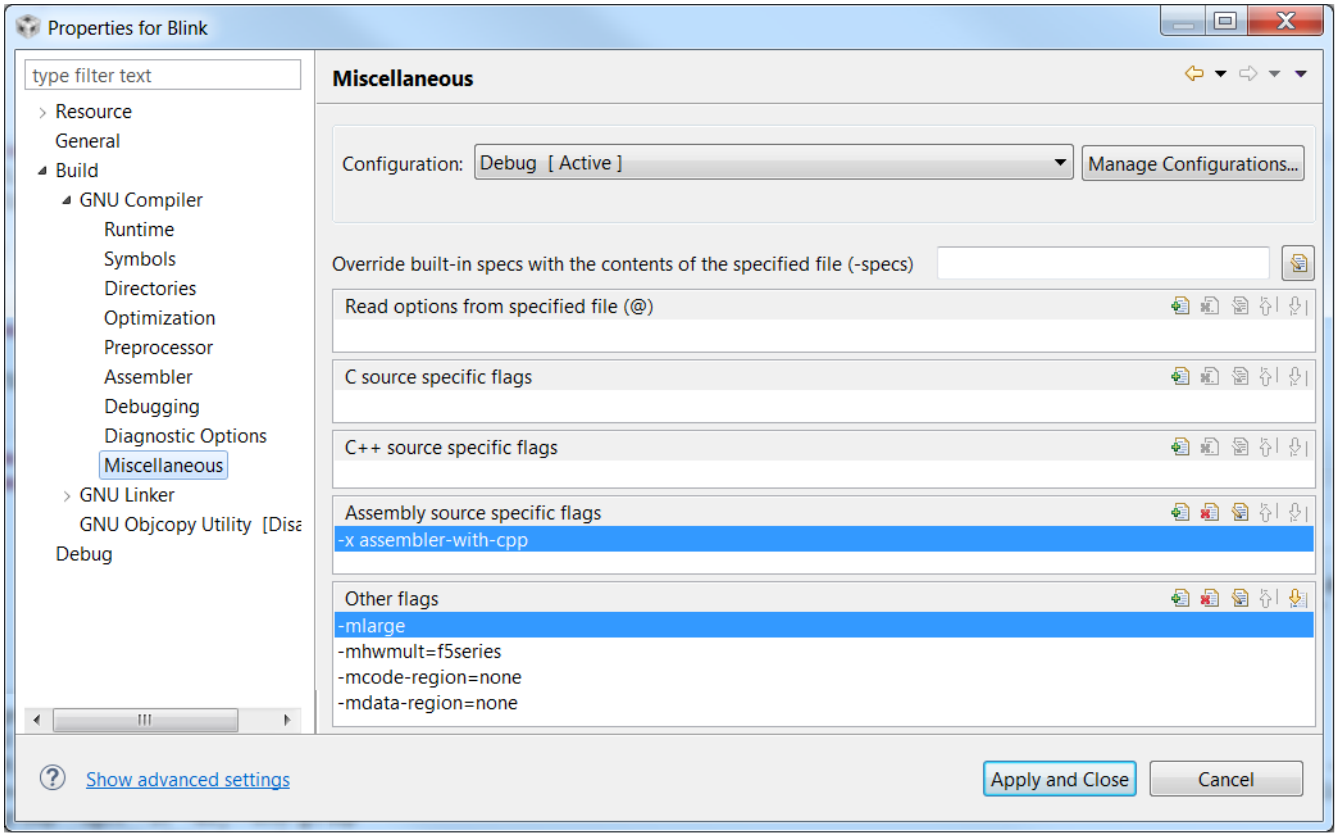

Figure 3-12. MSP430 GCC Settings: Miscellaneous

Table 3-10 describes the options that are available for MSP430 GCC Miscellaneous settings.

Table 3-10. MSP430 GCC Settings: Miscellaneous

Option	Description
Override built-in specs with the contents of the specified file (-specs)	The spec strings built into GCC can be overridden by using the -specs= command-line switch to specify a spec file.
Other flags	-mlarge Use large-model addressing (20-bit pointers, 20-bit size_t). -mcode-region=none -mdata-region=none The MSP430 compiler has the ability to automatically distribute code and data between low memory (addresses below 64K) and high memory. This only applies to parts that actually have both memory regions and only if the linker script for the part has been specifically set up to support this feature. See Table 4-2 for more information.

Copyright © 2020 Texas Instruments Incorporated

3.3.11 GNU Linker

Figure 3-13 shows the MSP430 GCC Linker settings window.

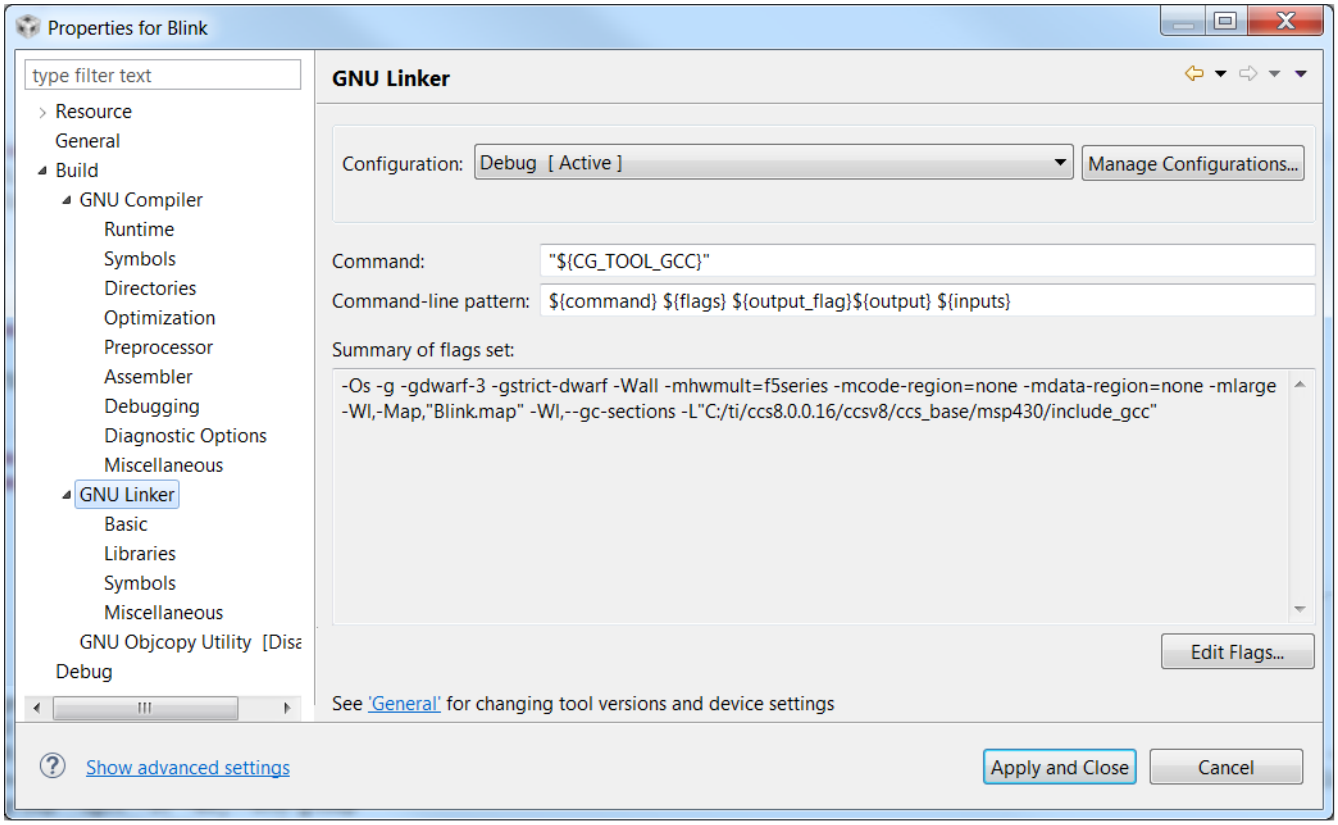

Figure 3-13. MSP430 GCC Linker Settings

Table 3-11 describes the options that are available for MSP430 GCC Linker settings.

Table 3-11. MSP430 GCC Linker Settings

Option	Description
Command	Linker location
Command-line pattern	Command line parameters
Summary of flags set	Command line with which the compiler is called. Displays all the flags passed to the linker.

Copyright © 2020 Texas Instruments Incorporated

3.3.12 GNU Linker: Basic

Figure 3-14 shows the MSP430 GCC Linker Basic settings window.

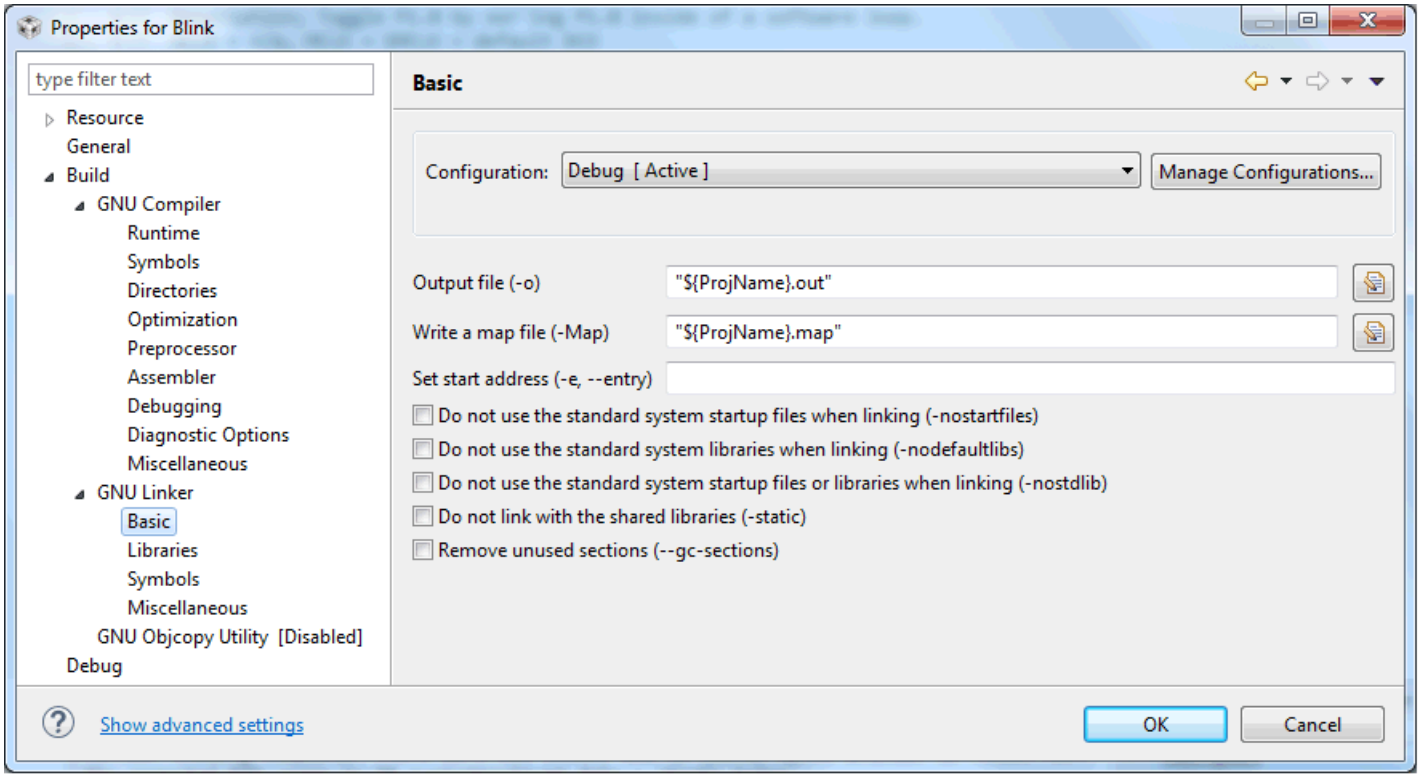

Figure 3-14. MSP430 GCC Linker Basic Settings

Table 3-12 describes the options that are available for MSP430 GCC Linker Basic settings.

Table 3-12. MSP430 GCC Linker Basic Settings

Option	Description
Output file (-o)	Use output as the name for the file produced by ld; if this option is not specified, the name 'a.out' is used by default. The script command OUTPUT can also specify the output file name.
Write a map file (-Map)	Print to the file mapfile a link map, which contains diagnostic information about where symbols are mapped by ld and information on global common storage allocation.
Set start address (-e, --entry)	Use entry as the explicit symbol for beginning program execution, rather than the default entry point.
Do not use the standard system startup files when linking (-nostartfiles)	Do not use the standard system startup files when linking. The standard system libraries are used unless -nostdlib or -nodefaultlibs is used.
Do not use the standard system libraries when linking (-nodefaultlibs)	Do not use the standard system libraries when linking. Only the specified libraries are passed to the linker, and options specifying linkage of the system libraries, such as -static-libgcc or -shared-libgcc, are ignored. The standard startup files are used unless -nostartfiles is used. The compiler may generate calls to memcmp, memset, memcpy, and memmove. These entries are usually resolved by entries in libc. These entry points should be supplied through some other mechanism when this option is specified.
Do not use the standard system startup files or libraries when linking (-nostdlib)	Do not use the standard system startup files or libraries when linking.
Do not link with the shared libraries (-static)	On systems that support dynamic linking, this prevents linking with the shared libraries. On other systems, this option has no effect.
Remove unused sections (--gc-sections)	Enable garbage collection of unused input sections. Ignored on targets that do not support this option.

Copyright © 2020 Texas Instruments Incorporated

3.3.13 GNU Linker: Libraries

Figure 3-15 shows the MSP430 GCC Linker Libraries settings window.

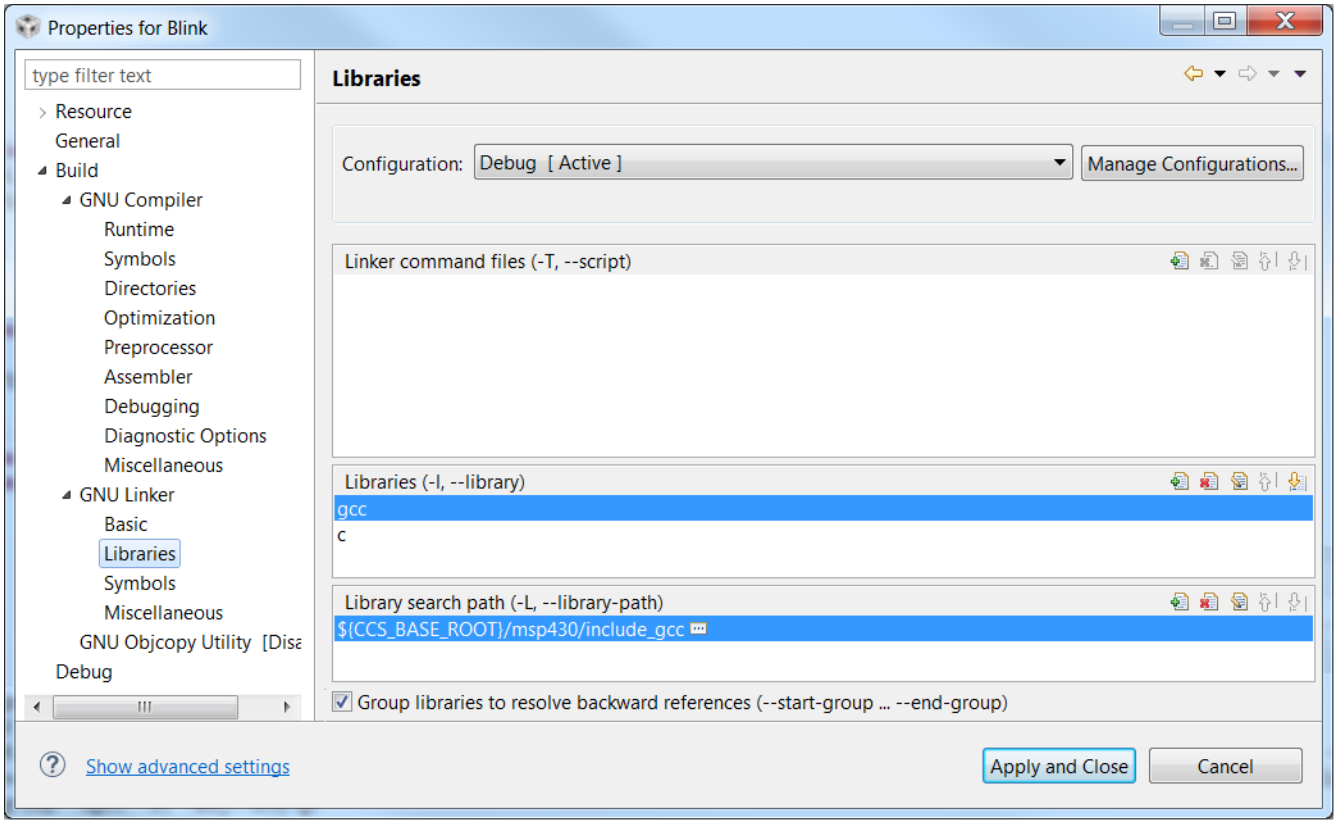

Figure 3-15. MSP430 GCC Linker Libraries Settings

Table 3-13 describes the options that are available for MSP430 GCC Linker Libraries settings.

Table 3-13. MSP430 GCC Linker Libraries Settings

Option	Description
Linker command files (-T, --script)	-T commandfile Read link commands from the file command file.
Libraries (-l, --library)	-l library Search the library named library when linking.
Library search path (-L, --library-path)	-L searchdir Add path searchdir to the list of paths that ld will search for archive libraries and ld control scripts.

Copyright © 2020 Texas Instruments Incorporated

3.3.14 GNU Linker: Symbols

Figure 3-16 shows the MSP430 GCC Linker Symbols settings window.

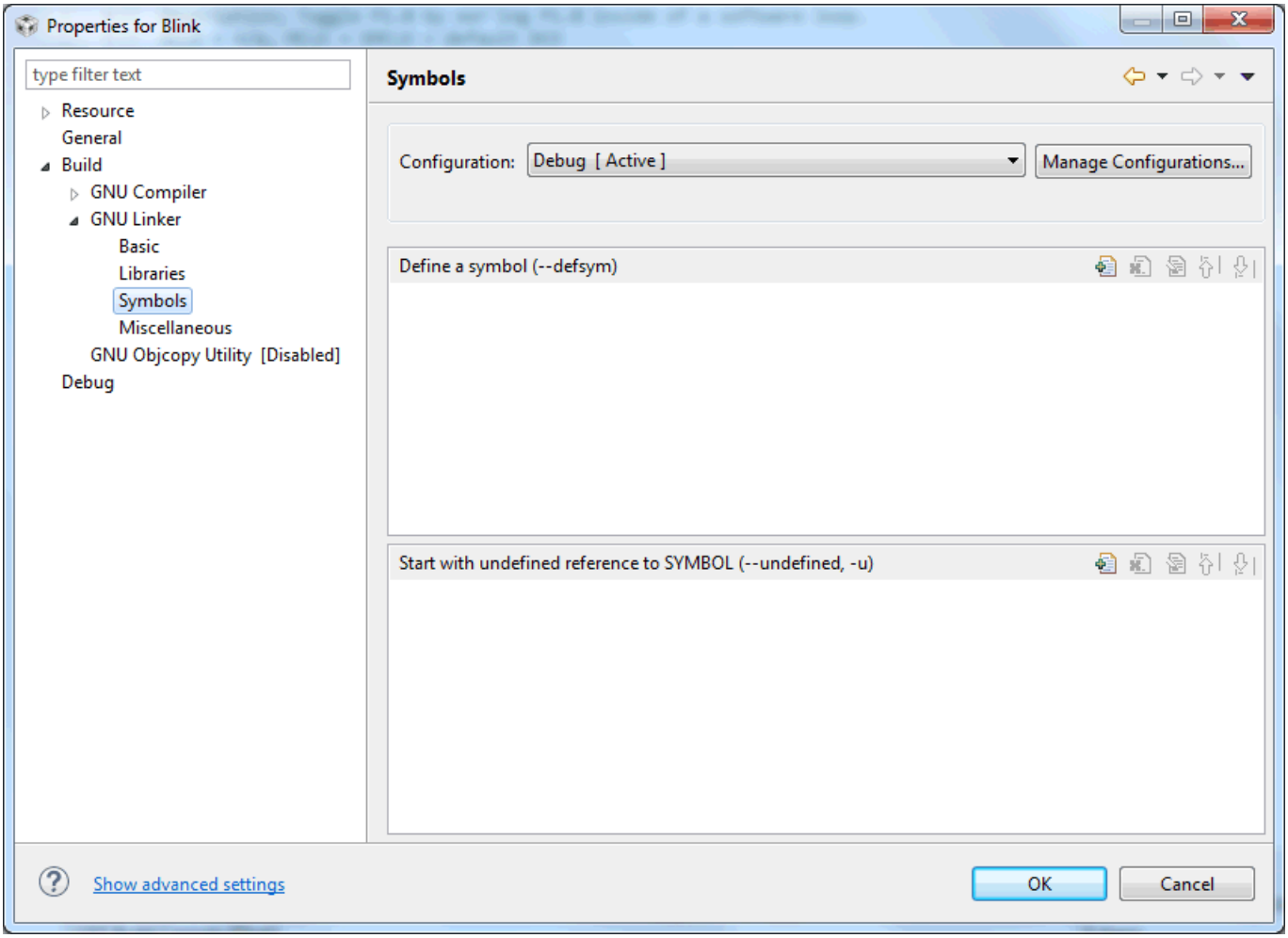

Figure 3-16. MSP430 GCC Linker Symbols Settings

Table 3-14 describes the options that are available for MSP430 GCC Linker Symbols settings.

Table 3-14. MSP430 GCC Linker Symbols Settings

Option	Description
Define a symbol (--defsym)	-defsym symbol=expression Create a global symbol in the output file, with the absolute address given by expression.
Start with undefined reference to SYMBOL (--undefined, -u)	Force symbol to be entered in the output file as an undefined symbol

Copyright © 2020 Texas Instruments Incorporated

3.3.15 GNU Linker: Miscellaneous

Figure 3-17 shows the MSP430 GCC Linker Miscellaneous settings window.

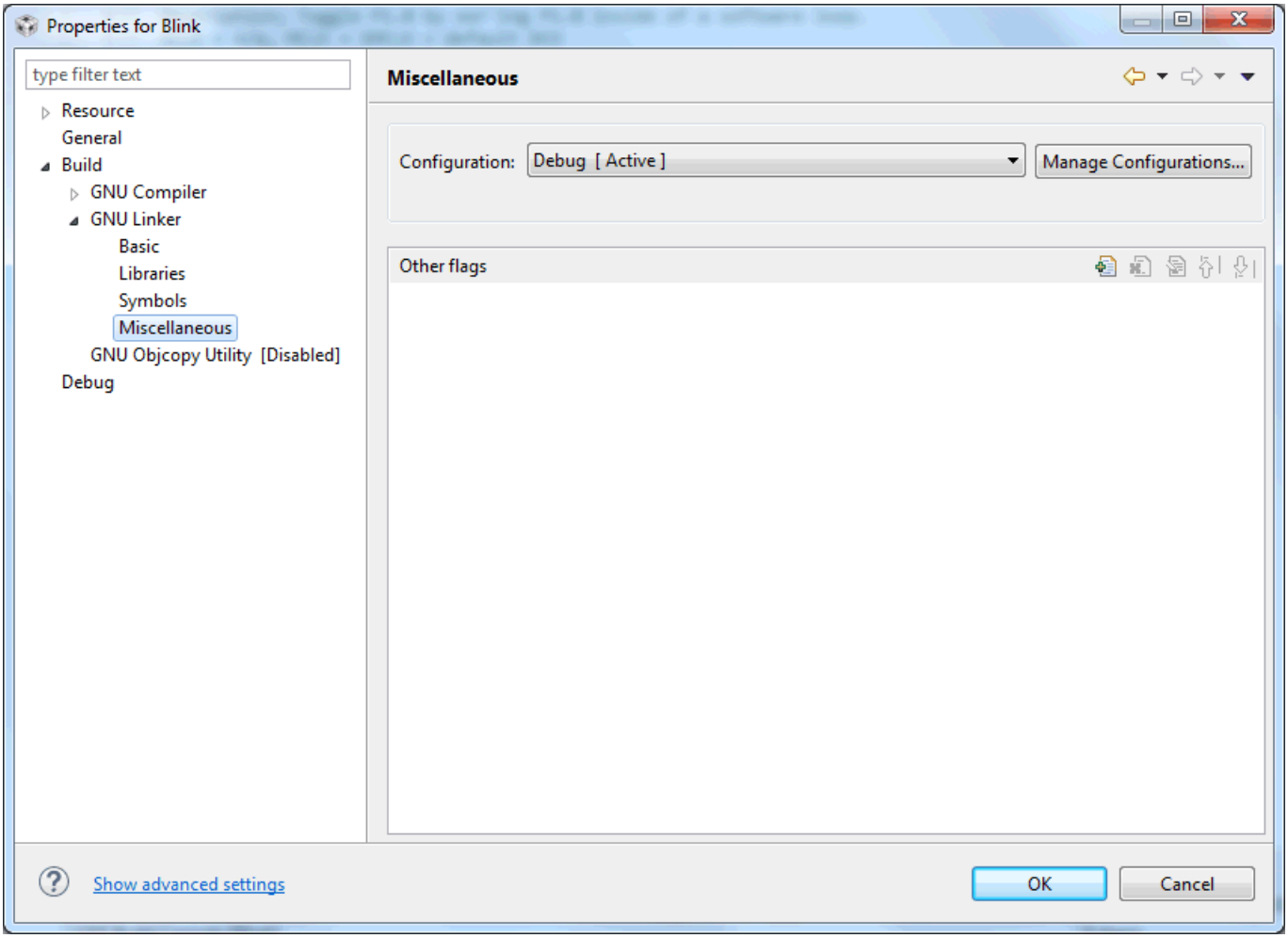

Figure 3-17. MSP430 GCC Linker Miscellaneous Settings

Table 3-15 describes the options that are available for MSP430 GCC Linker Miscellaneous settings.

Table 3-15. MSP430 GCC Linker Miscellaneous Settings

Option	Description
Other flags	Specifies individual flags based on the user requirements.

Copyright © 2020 Texas Instruments Incorporated

3.3.16 GNU Objcopy Utility

Figure 3-18 shows the MSP430 GCC GNU Objcopy Utility settings window.

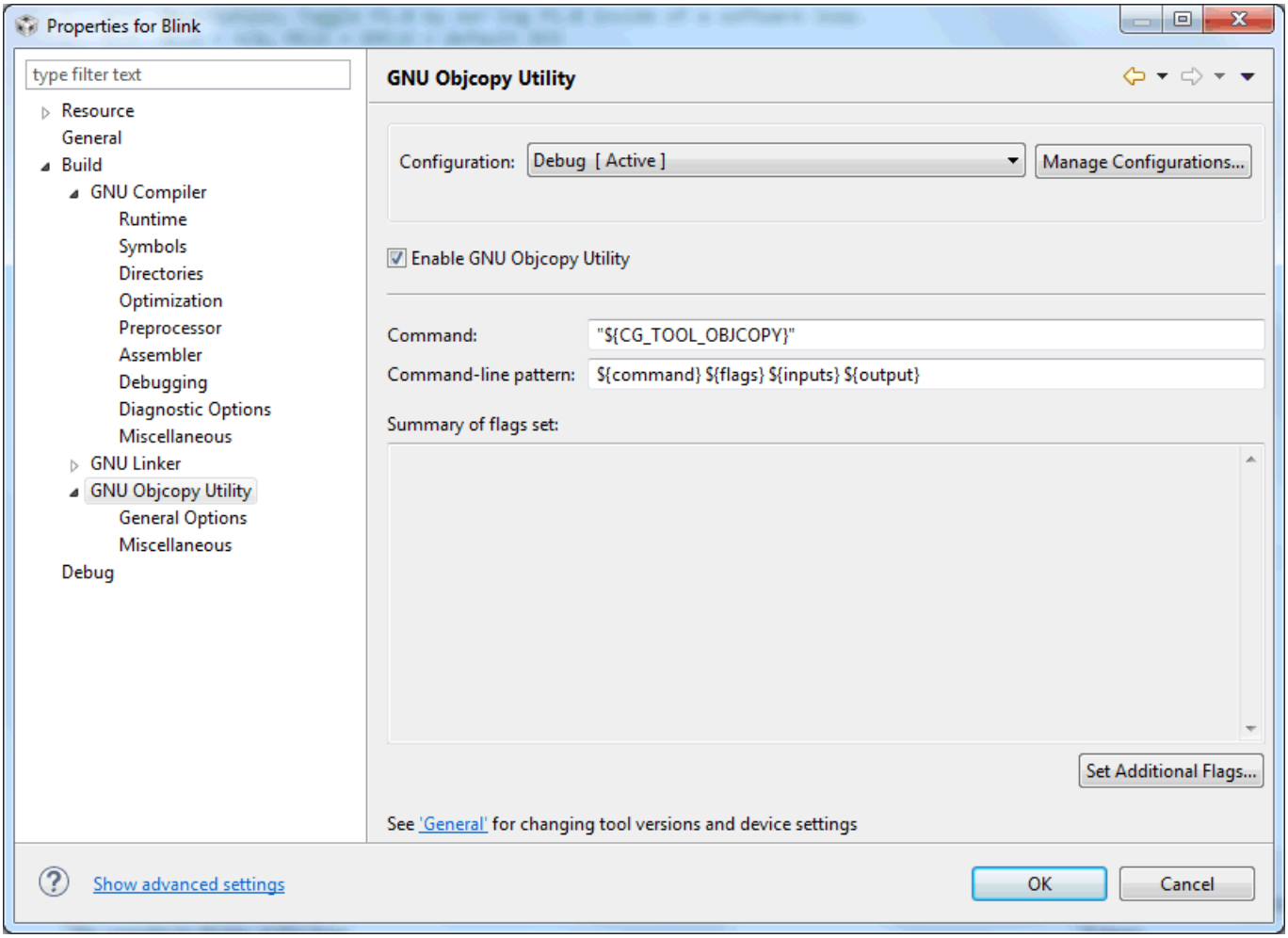

Figure 3-18. MSP430 GCC GNU Objcopy Utility Settings

Table 3-16 describes the options that are available for GNU Objcopy Utility.

Table 3-16. MSP430 GCC GNU Objcopy Utility Settings

Option	Description
Enable GNU Objcopy Utility	Enable this option to enable the GNU Objcopy Utility. It is disabled by default.
Command	GNU Objcopy location
Command-line pattern	Command line parameters
Summary of flags set	Command line with which the GNU Objcopy is called. Displays all the flags passed to the Objcopy command.

Copyright © 2020 Texas Instruments Incorporated

Figure 3-19 shows the MSP430 GCC GNU Objcopy Utility General Options settings window.

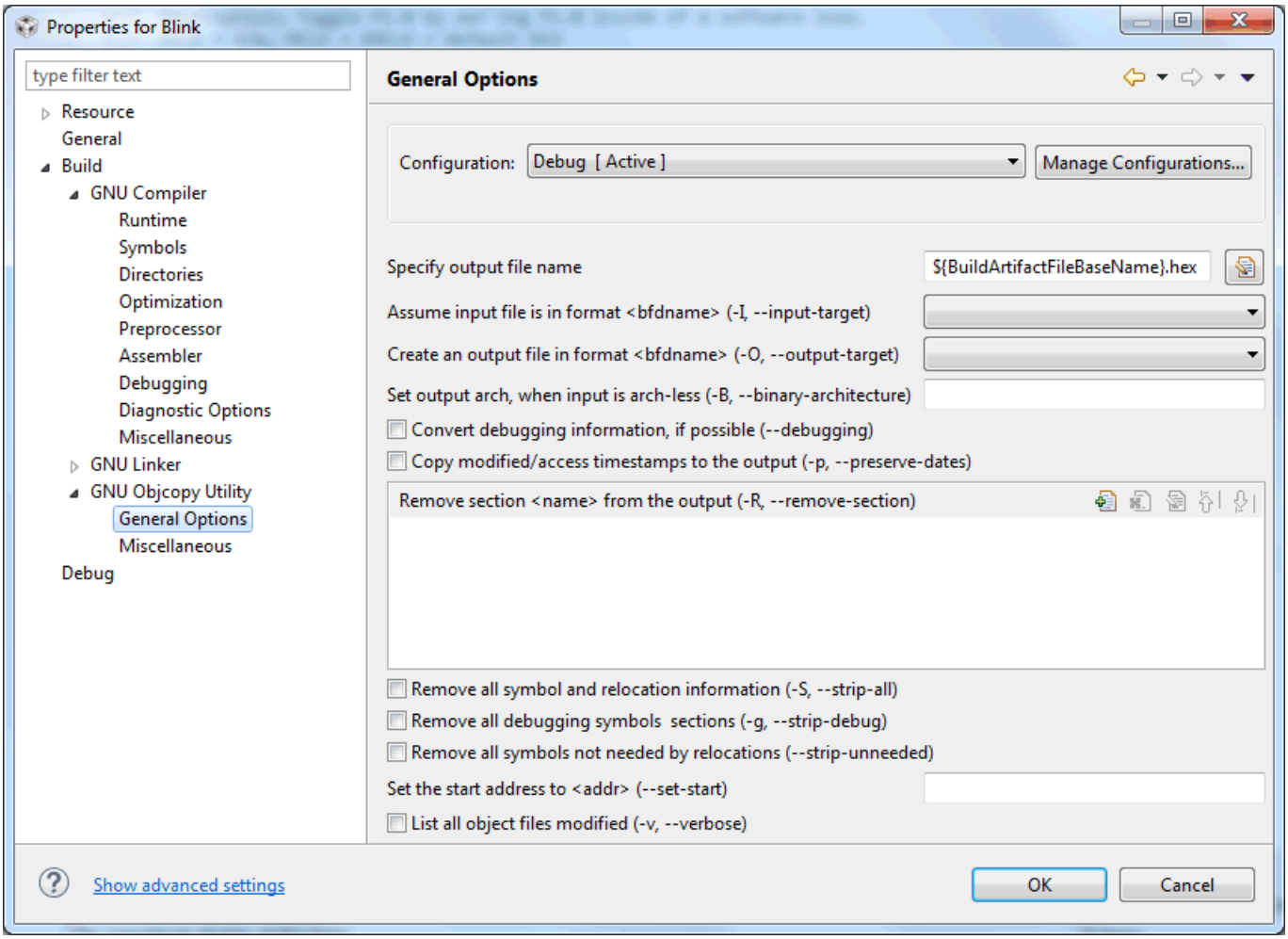

Figure 3-19. MSP430 GCC GNU Objcopy Utility General Options Settings

Table 3-17 describes the options that are available for GNU Objcopy Utility General Options.

Table 3-17. MSP430 GCC GNU Objcopy Utility General Options Settings

Option	Description
Specify output file name	Specifies the output file name
Assume input file is in format <bfdname> -I bfdname --input-target=bfdname	Consider the source file's object format to be bfdname, rather than attempting to deduce it.
Create an output file in format <bfdname> -O bfdname --output-target=bfdname	Write the output file using the object format bfdname.
Set output arch when input is arch-less -B bfdarch --binary-architecture=bfdarch	Useful when transforming an architecture-less input file into an object file. In this case the output architecture can be set to bfdarch.
Convert debugging information, if possible (--debugging)	Convert debugging information, if possible. This is not the default because only certain debugging formats are supported, and the conversion process can be time consuming.

Copyright © 2020 Texas Instruments Incorporated

Option	Description
Copy modified/access timestamps to the output (-p, --preserve-dates)	Set the access and modification dates of the output file to be the same as those of the input file.
Remove section <name> from the output -R sectionpattern --remove-section=sectionpattern	Remove any section matching sectionpattern from the output file. This option may be given more than once. Note that using this option inappropriately may make the output file unusable. Wildcard characters are accepted in sectionpattern. Using the -j and -R options together results in undefined behavior.
Remove all symbol and relocation information (-S, --strip-all)	Do not copy relocation and symbol information from the source file.
Remove all debugging symbols sections (-g, --strip-debug)	Do not copy debugging symbols or sections from the source file.
Remove all symbols not needed by relocations (--strip-unneeded)	Strip all symbols that are not needed for relocation processing.
Set the start address to <addr> (--set-start)	Set the start address of the new file to the specified value. Not all object file formats support setting the start address.
List all object files modified (-v, --verbose)	Verbose output: list all object files modified. In the case of archives, 'objcopy -V' lists all members of the archive.

Figure 3-20 shows the MSP430 GCC GNU Objcopy Utility Miscellaneous settings window.

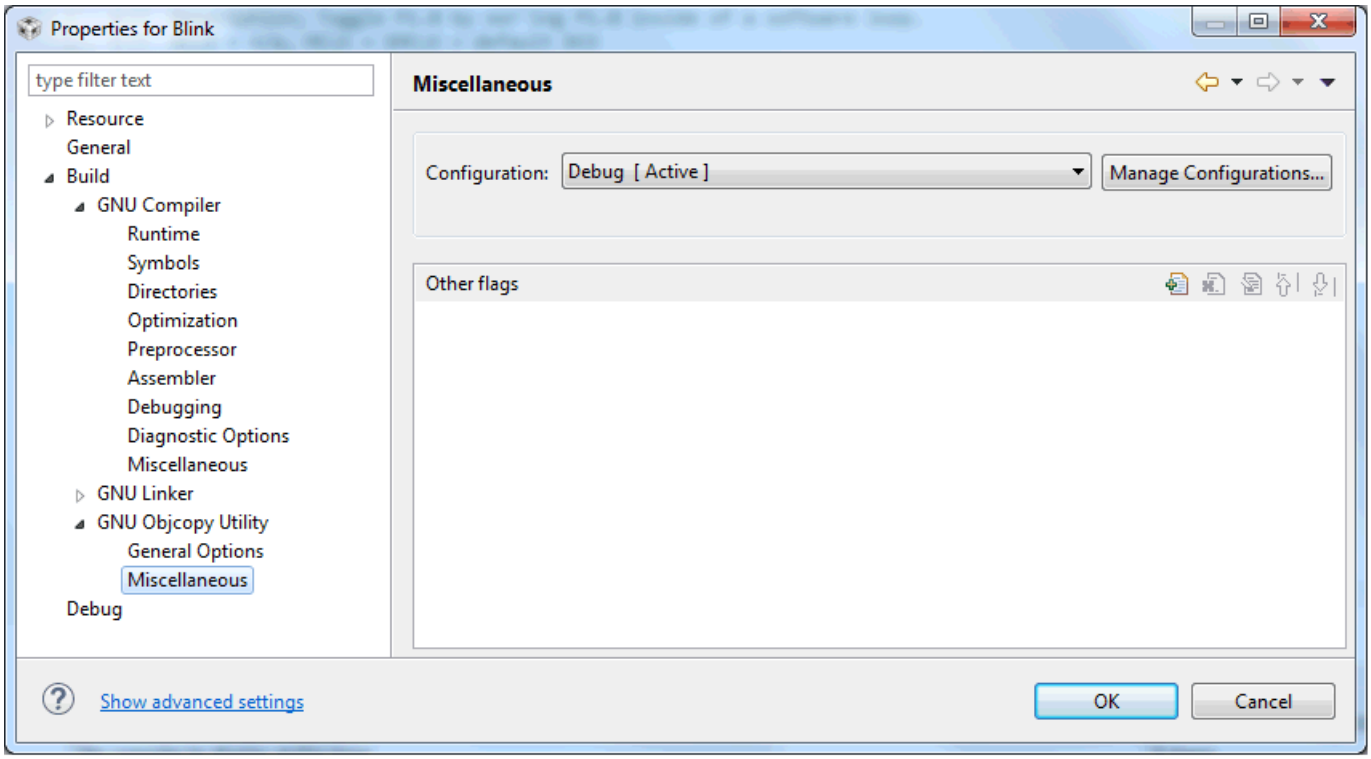

Figure 3-20. MSP430 GCC GNU Objcopy Utility Miscellaneous Settings

Table 3-18 describes the options that are available for GNU Objcopy Utility Miscellaneous.

Table 3-18. MSP430 GCC GNU Objcopy Utility Miscellaneous Settings

Option	Description
Other flags	Specifies individual flags based on the user requirements.

Copyright © 2020 Texas Instruments Incorporated

3.4 CCS Compared to MSP430 GCC

Some CCS features are not supported in MSP430 GCC. These features are:

- Optimizer Assistant
- ULP Advisor
- Memory Protection Unit and Intellectual Property Encapsulation GUI configuration
- Memory allocation

These features require the TI toolchain.

4 MSP430 GCC Stand-Alone Package

The MSP430 GCC stand-alone package is provided for users who prefer to use the MSP430 GCC toolchain with other IDE or console-based solutions for compiling and debugging. This stand-alone package supports different operating systems and is provided in different formats:

- GCC, Binutils, and GDB binaries for Windows, Linux, and macOS
- MSP430 header and linker files
- MSP430 GCC source code
- GDB agent configuration

The following table lists all the available MSP430 GCC stand-alone packages.

Table 4-1. MSP430 GCC Stand-Alone Package

Software	Description
msp430-gcc-full-linux-x64-installer-x.x.x.x.run	MSP430 GCC 64-bit Linux installer including support files, debug stack, and USB drivers. Run `sudo chmod +x <installer>` before executing the package.
msp430-gcc-full-osx-installer-x.x.x.x .app.zip	MSP430 GCC macOS installer including support files, debug stack, and USB drivers.
msp430-gcc-full-windows-installer-x.x.x.x.exe	MSP430 GCC Windows installer including support files, debug stack, and USB drivers.
msp430-gcc-x.x.x.x_{platform}.{zip,tar.bz2}	MSP430 GCC toolchain only. For linux32, linux64, macOS, win32, and win64 platforms.
msp430-gcc-x.x.x.x-source-full.tar.bz2	Full source tree of MSP430 GCC toolchain with scripts and instructions to build the toolchain from source.
msp430-gcc-x.x.x.x-source-patches.tar.bz2	Patch files for each of the components of MSP430 GCC (GCC, binutils, newlib, GDB). These can be applied on top of the corresponding upstream release of the component to build the toolchain from source.
msp430-gcc-support-files-x.xxx.zip	Header files and linker scripts.

Copyright © 2020 Texas Instruments Incorporated

4.1 MSP430 GCC Stand-Alone Package Folder Structure

The placeholder INSTALL_DIR refers to the directory where you installed the GCC MSP430 package.

- INSTALL_DIR
  - bin
    - MSP430 GCC, GDB, and Binutils binaries
    - GDB Agent
    - MSP430 Debug Stack
  - common
  - docs
    - Links to this user's guide and the quick start guide
  - emulation
    - Windows USB-FET Drivers
  - examples
    - Makefile-based examples to build and debug using MSP430 GCC
  - include
    - MSP430 Support Files (device header files and linker scripts)
  - lib
  - libexec
  - msp430-elf
    - lib
      - MSP430 Target Libraries
  - share
    - doc
      - HTML documentation for GCC, GDB, Binutils, and Newlib. The “index.html” file in each subdirectory is the recommended starting point for browsing.
  - msp430.dat
  - MSP430-GCC_manifest.html

4.2 Package Content

MSP430 GCC contains binary and sources software development tools for all TI MSP430 devices. The toolchain contains: compiler, assembler, linker, debugger, libraries, and other utilities.

These software development tools include:

- Compiler: MSP430 GCC (**msp430-elf-gcc** and **msp430-elf-g++**) is configured to compile C or C++.
- binutils: assembler, archiver, linker, librarian, and other programs.
- Newlib is the standard C library.
- Debugging: GDB (available with and without Python scripting support) and the GDB Agent:
  - **Note:** The Python-enabled GDB client (msp430-elf-gdb-py) requires Python 2.7 libraries to be installed on the host system.
- Source code: Toolchain source code is available at http://www.ti.com/tool/msp430-gcc-opensource.

Copyright © 2020 Texas Instruments Incorporated

4.3 MSP430 GCC Options

The GNU compiler, assembler, and linker in the MSP430 GCC toolchain support the target-specific options listed in Table 4-2, Table 4-3, and Table 4-4, in addition to the standard options. For the full list of options for the GNU compiler, refer to the GCC online documentation. For the full list of options for the GNU assembler (gas) and linker (ld), refer to the GNU binutils online documentation. The manifest distributed with the toolchain specifies the version numbers of each of the components.

Table 4-2. MSP430 GCC Command Options

Option	Description
-masm-hex	This option forces assembly output to always use hex constants. Normally such constants are signed decimals, but this option is available for test suite or aesthetic purposes.
-mcode-region= -mdata-region=	These options change the behavior of the compiler and linker. The names of function and data sections are modified so that they will be placed in a specific way across the upper and lower memory regions, according to the rules in the linker script. These options have no effect unless -mlarge is also passed. Possible values are: • upper – The compiler adds the ".upper" prefix to section names. – The linker adds the ".upper" prefix to all section names that do not already have a prefix. • either – The compiler adds the ".either" prefix to section names. – The linker adds the ".either" prefix to all section names that do not already have a prefix. – The linker places ".either" sections in the lower memory region. If the lower memory region overflows, the linker shuffles sections between the upper and lower memory regions to try to get the program to fit. • lower – Neither the compiler nor the linker adds the ".lower" prefix unless the -muse-lower-region-prefix option is also passed. – For -mdata-region=lower, the compiler assumes data is in the lower region of memory (in the 16-bit address range below address 0x10000), so MSP430 instructions can be generated to handle data. For other values passed to -mdata-region, the compiler assumes data could be in the entire 20-bit address range, so MSP430X instructions must be used to handle data. This results in increased code size compared to -mdata-region=lower. – Object files compiled with -mdata-region=lower cannot be linked with object files compiled with a different -mdata-region value. • none – Neither the compiler nor the linker makes any changes to section names. The default settings for these options are: -mdata-region=lower and -mcode-region=none.
-mhwmult=	This option describes the type of hardware multiply supported by the target. Accepted values: • 'none' for no hardware multiply • '16bit' for the original 16-bit-only multiply supported by early MCUs • '32bit' for the 16/32-bit multiply supported by later MCUs • 'f5series' for the 16/32-bit multiply supported by F5-series MCUs. • 'auto' can also be given. This tells GCC to deduce the hardware multiply support based upon the MCU name provided by the '-mmcu' option. If no -mmcu option is specified, then no hardware multiply support is assumed. 'auto' is the default setting. Hardware multiplies are normally performed by calling a library routine. This saves space in the generated code. When compiling at -O3 or higher, however, the hardware multiplier is invoked inline. This makes for larger but faster code. The hardware multiply routines disable interrupts while running and restore the previous interrupt state when they finish. This makes them safe to use inside interrupt handlers as well as in normal code.
-mdisable-device-warnings	Disable warnings emitted when the devices.csv file cannot be found by the compiler.

Copyright © 2020 Texas Instruments Incorporated

Option	Description
-minrt	This option is deprecated. The toolchain now dynamically decides which start up and initialization/ termination functions are required.
-mlarge	Use large-model addressing (20-bit pointers, 20-bit size_t). Small-model addressing is the default.
-mmax-inline-shift=<0,64>	This option takes an integer between 0 and 64 inclusive. The value sets the maximum number of inline shift instructions to emit to perform a shift operation by a constant amount. If this value will be exceeded, an MSPABI helper function is used instead. The default value is 4. This option affects only cases where a shift by multiple positions cannot be completed with a single instruction (for example, all shifts by more than one position on the MSP430 ISA). Shifts of a 32-bit value are at least twice as costly, so the value passed for this option is divided by 2 and the resulting value is used instead.
-mmcu=	This option selects the MCU to target. This is used to create a C preprocessor symbol based upon the MCU name, converted to upper case and prefixed and postfixed with '__'. This in turn is used by the 'msp430.h' header file to select an MCU-specific supplementary header file. The option also sets the ISA to use. If the MCU name is one that is known to only support the 430 ISA then that is selected, otherwise the 430X ISA is selected. A generic MCU name of 'msp430' can also be used to select the 430 ISA. Similarly, the generic 'msp430x' MCU name selects the 430X ISA. In addition, an MCU-specific linker script is added to the linker command line. The script's name is the name of the MCU with '.ld' appended. Thus specifying '-mmcu=xxx' on the gcc command line defines the C preprocessor symbol __XXX__ and causes the linker to search for a script called 'xxx.ld'. This option is also passed on to the assembler. Manually passing a linker script using the -T option prevents this linker script from being used automatically. To augment rather than displace this linker script, use the --script option to pass directly to the linker (-Wl,--script=).
-mrelax	This option enables both assembler and linker relaxation, which are optimizations that modify the code that was emitted by the compiler. See the descriptions of the -mQ option for the assembler and --relax option for the linker for more information. By default, assembler relaxation is disabled, and linker relaxation is enabled.
-msilicon-errata=NAME[,NAME...]	This option implements fixes for named silicon errata. Multiple silicon errata can be specified by multiple uses of the -msilicon-errata option or by including the errata names, separated by commas, on an individual -msilicon-errata option. Errata names currently recognized by the assembler are: cpu4 = PUSH #4 and PUSH #8 need longer encodings on the MSP430. This errata is enabled by default; it cannot be disabled. cpu8 = Do not set the SP to an odd value. cpu11 = Do not update the SR and the PC in the same instruction. cpu12 = Do not use the PC in a CMP or BIT instruction. cpu13 = Do not use an arithmetic instruction to modify the SR. cpu19 = Insert NOP after CPUOFF.
-msim	This option links to the simulator runtime libraries and linker script. Overrides any scripts that would be selected by the '-mmcu=' option.
-msmall	Use small-model addressing (16-bit pointers, 16-bit size_t). This is the default.
-msilicon-errata-warn=NAME[,NAME...]	This option is similar to the -msilicon-errata option, except that instead of fixing the specified errata, a warning message is issued instead. This option can either be used with -msilicon-errata to generate messages whenever a problem is fixed, or used on its own to inspect code for potential problems.
-mtiny-printf	This option links a reduced-size implementation of the printf() and puts() library functions. This reduced size implementation is not reentrant, so should be used with care in multi-threaded applications. Buffering of the string to be output has been removed from printf() and puts(), so user implementations of the system write() function must also implement buffering. The library implementation of write() that is included with MSP430 GCC does buffer the output string.
-muse-lower-region-prefix	This option adds the ".lower" prefix to function or data section names when -mdata-region=lower or -mcode-region=lower is passed.
-mwarn-mcu -mno-warn-mcu	This option enables or disables warnings about conflicts between the MCU name specified by the -mmcu option and the ISA set by the -mcpu option or the hardware multiply support set by the -mhwmult option. It also toggles warnings about unrecognized MCU names. This option is on by default.

Copyright © 2020 Texas Instruments Incorporated

Most assembler and linker options specific to MSP430 are passed automatically by the GCC wrapper, depending on the options passed to GCC itself. Table 4-3 and Table 4-4 list options that control behavior specific to the assembler and linker.

When passing an option to the assembler via the GCC wrapper (msp430-elf-gcc), the option must be prefixed with " `-Wa,` ". For example, to pass -mU to the assembler, you would pass `-Wa,-mU` to msp430-elf-gcc.

The assembler emits warnings when instructions to enable or disable interrupts are used without surrounding NOP instructions. See Section 8.2 for details. This behavior can be modified using the options in Table 4-3.

Table 4-3. MSP430 GCC Assembler Options

Option	Description
-mn	Insert NOPs around interrupt enable/disable instructions.
-mN	Do not insert NOPs around interrupt enable/disable instructions (default).
-mQ	Enable assembler relaxation. The assembler tries to replace some instructions with alternate versions that have smaller code sizes. This is disabled by default.
-mu	Warn or insert NOP instructions (default) around an instruction that may change the interrupt enable state if it is not known how the state will change. Whether a warning is emitted or a NOP is inserted is dependent on which of the -m{N,n,Y,y} options are set.
-mU	Do not warn or insert NOP instructions around an instruction that may change the interrupt enable state if it is not known how the state will change.
-my	Warn about missing NOPs around interrupt enable/disable instructions (default).
-mY	Do not warn about missing NOPs around interrupt enable/disable instructions.

When passing options to the linker via the GCC wrapper (msp430-elf-gcc), the option must be prefixed with " `-Wl,` ". For example, to pass --disable-sec-transformation to the linker, you would pass `-Wl,--disable-sec-transformation` to msp430-elf-gcc.

Table 4-4. MSP430 GCC Linker Options

Option	Description
--disable-sec-transformation	Disable the transformation of section names in object files being linked, based on the -mdata-region and -mcode-region options. For example, passing "-mdata-region=either -Wl,--disable-sec-transformation" to msp430-elf-gcc instructs the compiler to add the ".either" prefix to data section names being compiled, but the linker will not add the ".either" prefix to any section names in the object files it is passed.
--relax --no-relax	Enable or disable relaxation. The linker tries to replace some instructions with alternate versions that have smaller code sizes. Relaxation may grow short branch instructions that do not reach their target. Linker relaxation is enabled by default and cannot be disabled when linking executable files.

Table 4-5. MSP430 Objdump Options

Option	Description
--symtab-meta	Print the symbol meta-information entries from the .symtab_meta section of the specified ELF object or executable file.

4.4 MSP430 Built-in Functions

GCC provides special built-in functions to aid in the writing of interrupt handlers in C.

__bic_SR_register_on_exit (int mask)

This clears the indicated bits in the saved copy of the status register that currently resides on the stack. This only works inside interrupt handlers and the changes to the status register only take effect after the handler returns.

__bis_SR_register_on_exit (int mask)

This sets the indicated bits in the saved copy of the status register that currently resides on the stack. This only works inside interrupt handlers and the changes to the status register only take effect after the handler returns.

Copyright © 2020 Texas Instruments Incorporated

4.5 Using MSP430 GCC Support Files

MSP430 GCC uses the `devices.csv` file that is included with the MSP430 GCC Support Files package to get the device data for the device specified with the -mmcu option. This causes the source code to be built for the correct ISA and hardware multiplier with any necessary symbols defined. This ensures the correct operation of the toolchain. When using the -mmcu option, the toolchain automatically selects the correct header files and linker scripts for the device specified.

MSP430 GCC uses a few different methods to find the support files (in the following precedence order):

1. **Command-line options for compiler include path and linker library path.** The compiler looks in the directories specified by the -I option, and the linker looks in the directories specified by the -L option. Pass the path to the "include" directory in the MSP430 GCC Support Files package to both of these options. CCS uses this method by default, so users of the CCS IDE should not have to make any changes.
2. **Directory specified via environment variable.** If the command line options described above are not provided, the toolchain examines the MSP430_GCC_INCLUDE_DIR environment variable. Set this environment variable to the full path to the "include" directory in the MSP430 GCC Support Files package. For example, on Linux:

```
export MSP430_GCC_INCLUDE_DIR=/home/user/ti/gcc/include
```

3. **Default toolchain installation directory.** If neither a command line option nor the environment variable described above is provided, the toolchain checks the `msp430-elf/include/devices/` directory within the MSP430 GCC installation for the support files. Note that this "devices" directory does not exist in the latest toolchain installations, so the "include" directory from the support files package should be copied to this location. For example, on Linux:

```
cp -r /home/user/ti/gcc/include/ /home/user/ti/gcc/msp430-elf/include/devices/
```

Note

The toolchain stops searching for support files once it finds `devices.csv` . The results may be different than expected if one of the higher-precedence methods finds out-of-date support files, despite newer support files being pointed to by one of the lower-precedence methods.

Copyright © 2020 Texas Instruments Incorporated

4.6 Quick Start: Blink the LED

This document assumes that a version of the GNU Make utility is installed on the system and that it is available on the system path. The placeholder **INSTALL_DIR** refers to the directory where the GCC MSP430 package is installed. The directory **INSTALL_DIR/bin** should be on the system path.

4.6.1 Building with a Makefile

1. In the command terminal, go to the **INSTALL_DIR\examples** directory.
2. There are examples for Windows, macOS, and Linux. They are located in the corresponding subdirectories. Choose one of the examples suitable for the operating system and MSP430 target device.
3. Change to the directory and type **make**.
4. The binary can now be downloaded and debugged on the target hardware.

4.6.2 Building Manually with gcc

To build one of the examples manually, open a terminal and change to the example for the target device and operating system. The compiler executable **msp430-elf-gcc** must be available on your system path.

```
msp430-elf-gcc -I <Path to MSP430 Support Files> -L <Path to MSP430 Support Files>
-T DEVICE.ld -mmcu=DEVICE -O2 -g blink.c -o blink.o
```

The placeholder **<Path to MSP430 Support Files>** is the directory that contains the MSP430 support files (header files and linker scripts to support the different MSP430 devices).

The placeholder **DEVICE** tells the compiler and linker to create code for the target device. The command line argument -T DEVICE.ld is not normally required. When the -mmcu=DEVICE option is passed, the linker searches for the linker script “DEVICE.ld” in the current directory, and for paths specified with -L.

Example

```
msp430-elf-gcc -I ../../../include -L ../../../include -mmcu=msp430fr5969 \
-O2 -g blink.c -o blink.o
```

4.6.3 Debugging

4.6.3.1 Starting GDB Agent

On Microsoft Windows, the GDB Agent is available as either as a small GUI application or on the command line. On GNU Linux, only the command line version is available.

4.6.3.1.1 Using the GUI

Open the **INSTALL_DIR/bin** directory and double-click gdb_agent_gui.

1. After the program starts, click the button **Configure**, select **msp430.dat**, and click **Open**.
2. Click on the button **Start** under the Panel Controls.
3. The "Log" window now contains the status message "Waiting for client".
4. Leave the window open until the end of the debugging process.

4.6.3.1.2 Using the Command Line

Open a command terminal, change to INSTALL_DIR and type:

Linux

./bin/gdb_agent_console msp430.dat

Windows

.\bin\gdb_agent_console msp430.dat

Copyright © 2020 Texas Instruments Incorporated

4.6.3.2 Debugging With GDB

4.6.3.2.1 Running a Program in the Debugger

1. In the command terminal, go to the INSTALL_DIR\examples\[Selected example], and type the command **make debug**.
2. This command starts GDB and waits for commands. This is indicated by the prompt <gdb>.
3. To connect GDB to the GDB Agent, type the command **target remote :55000** and press enter.
4. To load the program binary to the MSP430 target device, type **load**.
5. Type the command **continue** (short version: c) to tell GDB to run the loaded program.
6. The LED on the target board blinks.

4.6.3.2.2 Setting a Breakpoint

1. Connect GDB to the GDB Agent as described in Section 4.6.3.2.1 and load a program to the device.
2. To set a breakpoint on a function, type **break function name**. By default this sets a software breakpoint (see Section 4.7.2). Use **hbreak** to set a hardware-assisted breakpoint, which speeds up debugging. A limited number of hardware breakpoints are available; the number is dependent on your device.
3. To set a breakpoint on a source line, type **break filename:line**.
4. When you run a program, program execution stops at the entry to the specified function or the specified line.

4.6.3.2.3 Single Stepping

1. Connect GDB to the GDB Agent as described in Section 4.6.3.2.1 and load a program to the device.
2. After the debugger has stopped the program at a breakpoint, you can step through the code:
   - To execute the source line, type **next**. **next** does not step into functions, it executes the complete function and stops on the line following the function call.
   - To execute the next source line and step into functions, type **step**.
   - To execute the next instruction, type **nexti**.
   - To execute the next instruction and step into functions, type **stepi**.

4.6.3.2.4 Stopping or Interrupting a Running Program

1. Connect GDB to the GDB Agent as described in Section 4.6.3.2.1 and load a program to the device.
2. To stop a running program and go to the GDB command prompt, press **Ctrl+C** (not supported on Windows).

4.6.4 Creating a New Project

1. Create a directory for your project, and copy one of the example project makefiles into the project directory.
2. Open the copied makefile and set the variable DEVICE to the target device.
3. Set the variable GCC_DIR to point to the directory where the GCC MSP430 package is installed.
4. Include all of the project source files (that is, the *.c files) as a dependency for the first target of the makefile.
5. Go to the project directory in a terminal. Type **make** to build the project or **make debug** to debug the project.

```
OBJECTS=blink.o
GCC_DIR =  ../../../bin
SUPPORT_FILE_DIRECTORY = ../../../include

# Please set your device here
DEVICE  = msp430X
CC      = $(GCC_DIR)/msp430-elf-gcc
GDB     = $(GCC_DIR)/msp430-elf-gdb
CFLAGS = -I $(SUPPORT_FILE_DIRECTORY) -mmcu=$(DEVICE) -O2 -g
LFLAGS = -L $(SUPPORT_FILE_DIRECTORY) -T $(DEVICE).ld

all: ${OBJECTS}
    $(CC) $(CFLAGS) $(LFLAGS) $? -o $(DEVICE).out
debug: all
    $(GDB) $(DEVICE).out
```


Copyright © 2020 Texas Instruments Incorporated

4.7 GDB Settings

The GDB Agent is a tool to connect GDB with the target hardware to debug software. The GDB Agent uses the MSP430 debug stack to connect to the hardware and provides an interface to GDB. On Windows, both a console and a GUI application version of the GDB Agent are provided. Only the console application is supported on Linux.

4.7.1 Console Application

If you use the console application, run it from a command terminal using following syntax:

Linux

INSTALL_DIR/bin/gdb_agent_console INSTALL_DIR/msp430.dat

Windows

INSTALL_DIR\bin\gdb_agent_console INSTALL_DIR\msp430.dat

The console application opens a TCP/IP port on the local machine. It displays the port number in the console. By default, this port number is 55000.

4.7.2 Optional Parameters for msp430.dat

Add the following lines to msp430.dat to enable or modify the specific debug options.

MSP430 low-power debugging

```
msp430_lowpowerdebug=true
```

MSP430 programming and erase options for GDB

The BSL or the protected memory can be unlocked at the start of the session using the `msp430_connectaction` keyword.

```
msp430_connectaction=[unlockbsl] [unlockprotected] [<other connect options>]
```

If protected memory is unlocked, it is erased on download if the download erase option is set to `erasefactory` , `erasemain` , or `erasemainandinfo` , or if the download erase option is set to `erasesegment` and the download image includes protected memory data.

`msp430_loadaction` controls whether a reset is done before or after download and also configures erase options for download.

```
msp430_loadaction =[resetbefore] [resetafter] [erasefactory|erasemain|erasemainandinfo|eraseuser|
erasesegment] [<other load options>]
```

Where:

`erasefactory` = MSP430_Erase(type = ERASE_TOTAL, ...) called at the start of each download

`erasemain` = MSP430_Erase(type = ERASE_MAIN, ...) called at the start of each download

`erasemainandinfo` = MSP430_Erase(type = ERASE_ALL, ...) called at the start of each download

`eraseuser` = MSP430_Erase(type = ERASE_USER, ...) called at the start of each download

`erasesegment` = MSP430_Erase(type = ERASE_SEGMENT, ...) called the first time that the segment is written to during each download

Another optional connect action is to reset or erase on connect:

```
msp430_connectaction=[reset] [erasefactory|erasemain|erasemainandinfo|eraseuser]
```

If this line is not present, the default is to not reset or erase on connect.

Copyright © 2020 Texas Instruments Incorporated

If the BSL or protected areas are unlocked, they are erased on connect if the `erasefactory` or `erasemainandinfo` options are set. They are erased on download if the `erasefactory` , `erasemainandinfo` , or `erasesegment` options are set and the image includes the BSL or protected segment.

No action is taken on the `auto run` or `launch` options as they are outside the scope of the GDB agent.

MSP430 verification options for GDB

`msp430_loadaction` includes a `verify` keyword. If this keyword is present, each write to flash is verified.

```
msp430_loadaction=[verify]
```

Add MSP430 breakpoints options for GDB

```
msp430_default_breakpoint = [software|hardware]
```

By default, the MSP430 GDB agent uses software breakpoints (type=BP_SOFTWARE) for all GDB `break` commands. If a software breakpoint fails, the GDB agent then attempts to set a hardware (type=BP_CODE) breakpoint. A hardware breakpoint can be set explicitly using the GDB `hbreak` command.

If the `msp430_default_breakpoint` option is set to `hardware`, the GDB agent uses type BP_CODE for all GDB breakpoints (both `break` and `hbreak` commands).

4.7.3 GUI Application

After you start the GUI application, configure the GUI and then start the GDB server. For more information, refer to the XDS GDB Agent online documentation.

1. Click the **Configure** button and, in the Select board configuration file window, select the msp430.dat file. If successfully configured, an MSP430 device is displayed in the **<Targets>** list. The TCP/IP port for the GDB server is displayed when the MSP430 device is selected from the list.
2. To start the GDB Agent, click the **Start** button when the MSP430 device is selected.

4.7.4 Attaching the Debugger

After starting the debugger and to attach to the GDB server, use the target remote **[<host ip address>]:<port>** command, where **<port>** is the TCP/IP port from above. If the GDB Agent runs locally, omit the host IP address.

4.7.5 Configuring the Target Voltage

To configure the target voltage for the device, open the file msp430.dat in a text editor. To change the voltage, modify the key msp430_vcc. By default, this value is set to 3.3 V.

4.7.6 Resetting the Target

To reset the target, use the **monitor reset** command.

4.7.7 Halting the Target

To halt the target, use the **monitor halt** command.

Copyright © 2020 Texas Instruments Incorporated

5 MSP430 GCC Features

The following MSP430-specific features are additions to the standard set of GCC features.

5.1 C/C++ Attributes

The following sections describe MSP430-specific additions to the function and data attributes available with GCC.

5.1.1 GCC Function Attribute Support

The following attributes may be applied to function declarations:

- **critical**

  Disable interrupts on entry, and restore the previous interrupt state on exit.
- **interrupt** or **interrupt(*x*)**

  Make the function an interrupt service routine for interrupt "x". This attribute can also be used without an argument. If no argument is used, the function is not linked to an interrupt, but the function will have properties that are associated with interrupts.

  To define an interrupt, use the following syntax:

```
void __attribute__ ((interrupt(INTERRUPT_VECTOR))) INTERRUPT_ISR (void)
```

  Example:

```
void __attribute__ ((interrupt(UNMI_VECTOR))) UNMI_ISR (void)
{ // isr }
```

  You can also use the following macro defined in the `iomacros.h` file, which is automatically included when using `msp430.h` from the MSP430 GCC support files:

```
#define __interrupt_vec(vec) __attribute__((interrupt(vec)))
```

  Example:

```
void __interrupt_vec(UNMI_VECTOR) UNMI_ISR (void)
{}
```

- **location(*address*)**

  Can be applied to declarations of functions or static/global data. It instructs the linker to place the section containing the function or data object at the specified address if possible.

  For accurate placement, objects with this location attribute should be in their own sections. This can be achieved by using the GCC command-line options `-f{function,data}-sections` or by applying the "section" attribute to the object's declaration.

  The memory region containing the specified address must be compatible with the type of section created for the object. For example, you cannot place read/write data in a read-only memory region. The linker deduces the memory region type from the flags set in the linker script for the memory region and the names of output sections that have been placed in that memory region. For example, if a memory region contains an output section called ".text", the linker assumes that entire region is executable.

  The linker renames sections containing objects that use the "location" attribute, giving them the prefix ".smi.location". These ".smi.location" sections are placed either in their own output sections in the linked executable or within other output sections, among input sections that do not rely on a specific ordering. More specifically, the linker places ".smi.location" sections with other input sections only under certain known output sections such as ".text" or ".data.*". This prevents ".smi.location" sections from being placed with sections such as ".crt_*", which rely on section ordering for proper operation of the program.

Copyright © 2020 Texas Instruments Incorporated

If the linker cannot place a ".smi.location" section within an output section due to an incompatibility, it places the ".smi.location" section in its own output section at the specified address. It then shifts the incompatible output section to be placed immediately after it.

The linker only attempts to place the first object with a location attribute in an input section. It emits a warning if any subsequent location attributes are ignored. Since the linker can only place location objects at a specific address by placing their input section at that address, it is not possible to accurately place more than one location object per input section.

To initialize data/bss variables that have been placed at specific locations, the linker creates a ".smi.location_init_array" section. The C Run-time (CRT) startup code uses this table to copy data or zero-initialize bss variables just before calling main().

- **naked**

  Do not generate a prologue or epilogue for the function.
- **reentrant**

  Disable interrupts on entry, and always enable them on exit.
- **retain**

  Can be applied to declarations of functions or static/global data. It instructs the linker to retain the section that contains this object in the linked executable file, even if symbols in that section appear unused. This prevents the section from being garbage collected. This attribute implies that the "used" attribute also applies.
- **wakeup**

  When applied to an interrupt service routine, wake the processor from any low-power state as the routine exits. When applied to other routines, this attribute is silently ignored.

5.1.2 GCC Data Attribute Support

The following attributes may be applied to variable declarations:

- **location**

  See the description of this attribute in Section 5.1.1.
- **noinit**

  Variables with this attribute are not initialized by the C runtime startup code or the program loader. Not initializing data in this way can reduce program startup times. A compiler warning will be provided if a variable marked with the noinit attribute is initialized to a constant value.
- **persistent**

  Variables with this attribute are not initialized by the C runtime startup code. Instead their value is set once when the application is loaded, and then never initialized again, even if the processor is reset or the program restarts. Persistent data is intended to be placed into Flash RAM, where its value will be retained across resets. The linker script used to create the application should ensure that persistent data is correctly placed. A compiler warning is provided if a variable marked with persistent is not initialized to a constant value.
- **retain**

  See the description of this attribute in Section 5.1.1.

Copyright © 2020 Texas Instruments Incorporated

5.1.3 GCC Section Attribute Support

The following attributes can be applied to functions or data to add a prefix to their default section name. This may change whether the linker places them in high or low memory. See Table 4-2 for details about how the linker handles sections with these prefixes.

- **lower**

  Adds the ".lower" prefix to the default section name.
- **upper**

  Adds the ".upper" prefix to the default section name.
- **either**

  Adds the ".either" prefix to the default section name.

5.2 Hints for Reducing the Size of MSP430 GCC Programs

In addition to size optimization (-Os) and link-time optimization (-flto), there are additional command-line options you can use and small changes you can make to reduce the code and data size of your program.

5.2.1 The -mtiny-printf Option

The -mtiny-printf option enables minimal implementations of the printf() and puts() library functions. These implementations reduce code size by removing some functionality. See Table 4-2 for more information.

5.2.2 The -ffunction-sections and -fdata-sections Options

The -ffunction-sections and -fdata-sections options instruct the GCC compiler to create a new output section for each function and data object. When used with the --gc-sections linker option, these options ensure that the linker can perform garbage collection of unused function and data objects in your program.

Typically, this reduces overall program size. However, if there is not a significant number of sections the linker can remove, these options may actually increase the size of your program and reduce performance. This is because the GCC compiler cannot perform certain optimizations when these options are enabled. We recommend that you experiment with these options to determine the overall effect they have on your program size.

5.2.3 Making Large Programs Fit Across Upper and Lower Memory

For MSP430X devices that have an "upper" memory region (memory above the 0xFFFF boundary), the large memory model (-mlarge) is supported.

If a program built for one of these MSP430X devices is too large to fit exclusively in the lower memory region using the small memory model, rebuilding the program for the large memory model (-mlarge) with -mcode-region=either and possibly -mdata-region=either can help it to fit.

There is a code size and performance penalty when using -mdata-region=either. MSP430X instructions must be generated to address data, even in cases where an MSP430 instruction would suffice, since the compiler must assume that data might be placed in the upper memory region. Therefore, -mdata-region=either should be used only if it is necessary—that is, if the program would not otherwise fit on the device.

Note that the -mlarge option alone forces the compiler to generate CALLA and RETA instructions for all subroutine calls and returns. While there is no code size penalty for using these instructions instead of CALL and RET, there is some performance overhead. There is no additional penalty to using -mcode-region=either if -mlarge is in use.

See Table 4-2 for details on how the "either" options shuffle code and data sections between upper and lower memory. For best results, use these options with the -ffunction-sections and -fdata-sections options, so that the sections available to be shuffled are smaller.

Copyright © 2020 Texas Instruments Incorporated

5.2.4 NOP Instructions Surrounding Interrupt State Changes

To prevent incorrect behavior when two adjacent instructions both change the global interrupt enable state (see Section 8.2), the C macros for modifying the interrupt state contain NOP instructions. The following macros are affected and are defined in `in430.h`, which is part of the msp430-gcc-support-files package:

- _set_interrupt_state
- _enable_interrupts
- _disable_interrupts
- _bis_SR_register
- _bic_SR_register

Macros that modify the status register (SR) might be used for purposes other than changing the interrupt state, so NOP instructions in these macros may not be needed. If code size is important, you can examine places your source code uses the _bic_SR_register and _bis_SR_register macros. If NOPs inserted by macros are not needed in some places, you can define your own macros that omit NOPs and use them where appropriate.

5.3 C Runtime Library (CRT) Startup Behavior

When your program starts running but before execution reaches main(), C Runtime (CRT) startup code initializes global/static data, zero-initializes bss, and calls any functions stored in .init_array (for example, global constructors for C++). The inclusion of functionality during this startup sequence is dynamic, so that functions are linked into the program only if they are required.

CRT functions are each stored in their own section. The linker script sorts these sections by name to enforce the order in which they are executed. You can cause your own functions to run before main() by placing them in a section with a name beginning with ".crt_####", where #### is a 4-digit decimal number, padded with leading 0s.

It is important to mark these functions with both the "naked" and "used" function attributes. The "naked" attribute removes the function prologue and epilogue, allowing the function to "fall-through" to the next CRT function instead of trying to execute a return instruction. The "used" attribute prevents compiler optimization from removing the function if it is not explicitly called. See Section 5.1.1

User-specified CRT functions may be used to disable the watchdog immediately after program start. This can prevent the watchdog from firing before main(), during initialization of large data or bss sections. For example:

```
#include <msp430.h>

static void __attribute__((naked, used, section(".crt_0042")))
disable_watchdog (void)
{
  WDTCTL = WDTPW | WDTHOLD;
}
```

The names of existing sections containing CRT startup code are show below. This list can be extracted by looking at the section names in "libcrt.a" and "crt0.o".

- .crt_0000start
- .crt_0100init_bss
- .crt_0200init_highbss
- .crt_0300movedata
- .crt_0400move_highdata
- .crt_0500run_preinit_array
- .crt_0600run_init_array
- .crt_0710run_smi_location_init_array
- .crt_0800call_main

The example above for the disable_watchdog() function would therefore run immediately after the system initializes and branches to "_start" and before bss initialization is performed.

Copyright © 2020 Texas Instruments Incorporated

5.4 Using printf with MSP430 GCC

The printf() function is commonly used to assist with debugging, however its behavior in MSP430-GCC is dependent on the debugging software being used.

When debugging within CCS, printf() output is shown in the CIO console.

When debugging with GDB and the GDB Agent, printf() is silently ignored. This is because the default implementation relies on the debugger understanding the TI C I/O protocol, which is currently unimplemented in GDB.

The default implementation can be overridden by defining the write() system call in your application. The write() function is called with the finalized string, ready to be printed, and must handle the process of outputting the string. The prototype for write() is:

```
int write (int fd, const char *buf, int len);
```

- `fd` is the file descriptor, which for printf will always be STDOUT.
- `buf` contains a pointer to the formatted string.
- `len` is the number of bytes from `buf` that should be written.

5.5 Link-time Optimization (LTO)

MSP430 GCC supports link-time optimization, which significantly reduces code size and improves performance in projects that consist of multiple source files and libraries. It can be enabled by passing the -flto flag to msp430-elf-gcc.

LTO is a feature of the GCC compiler, and does not affect the behavior of the linker. It is recommended that you compile all source files using the same options, and pass that same set of options when linking using msp430-elf-gcc. For the most straightforward operation, just compile and link all your source files using a single invocation of msp430-elf-gcc.

If the --save-temps option is passed, the optimized assembly code from your entire project will be stored in <output_filename>.ltrans0.s. The output assembly files from individual source files contain the LTO bytecode.

For more detailed information and tips on using LTO, see the GCC documentation on the -flto option.

5.6 The __int20 Type and Pointers in the Large Memory Model

For MSP430X devices in the large memory model, pointers and size_t are 20 bits in size. To handle this, GCC defines the `__int20` type, which uses 20 bits of space in registers, and 32 bits of space in memory.

This type is built into the compiler, so it can be used like any other type in source code. Using `unsigned __int20` designates the unsigned version of the type.

The __int20 type is not part of the ISO C specification, so using this type in any program compiled with flags that check for standards compliance may cause a warning or error to be emitted. A variation of the type that will not trigger these standards compliance messages is defined as `__int20__` (using a trailing double underscore).

Copyright © 2020 Texas Instruments Incorporated

6 Building MSP430 GCC From Sources

6.1 Required Tools

This document assumes that the required tools are installed on the system and that they are available on the system path.

- GNU make
- GCC and binutils
- bzip2 and tar
- curl, flex, bison, and texinfo

6.2 Building MSP430 GCC (Mitto Systems Limited)

The `README-build.sh` bash script included with the source-full package (and in the source-patches package) can be used to build the toolchain for Windows, Linux and macOS hosts. The script contains some distribution-specific instructions on how to install the pre-requisite tools from Section 6.1.

To build native Linux and macOS toolchains, follow the instructions in Section 6.2.1.

To build the toolchain for Windows hosts, follow both the instructions in Section 6.2.1 and then Section 6.2.2.

Note

If less than 2 GB of RAM is available during the build, the build may fail.

6.2.1 Building a Native MSP430 GCC Toolchain

Follow these steps to build Mitto MSP430 GCC for Linux and macOS:

1. Download the source-full tar archive (for example, msp430-gcc-7.3.0.9-source-full.tar.bz2) from the MSP430 GCC page.
2. Untar the file.
3. Change to the source-full directory.
4. Run `README-build.sh` to build the toolchain.
5. Build files are in the ./build folder.
6. Binaries/libs are in the ./install folder.

Note

An alternative to this process is to use the "source-patches" tar archive (for example, `msp430-gcc-7.3.0.9-source-patches.tar.bz2`) to apply patches to source tars as released by the upstream community.

Versions 7.3.0.9 and later include a script (`README-apply-patches.sh`), which downloads the upstream releases and applies the patches so the sources are ready for building. The `README-build.sh` script can then be used to build a native toolchain.

Copyright © 2020 Texas Instruments Incorporated

6.2.2 Building the MSP430 GCC Toolchain for Windows

Follow these instructions to build the toolchain for Windows hosts:

1. Begin by following the instructions in Section 6.2.1 to build a native toolchain.
2. Move the installation directory to a permanent location. (This is because the `README-build.sh` script deletes the "build" and "install" directories before starting the toolchain build.)
3. Install a cross-compiler for Windows:
   - On Ubuntu install the "mingw-w64" package as follows:

     ```
     > apt-get install mingw-w64
     ```
   - On Centos 7, first install the "Extra Packages for Enterprise Linux" (EPEL) repository, then install the mingw toolchain as follows:

     ```
     > yum install epel-release
     > yum install mingw64-gcc.x86_64 mingw64-gcc-c++.x86_64
     ```
4. Add the desired host platform to "configure_args_common" in `README-build.sh` . For 64-bit Windows this is usually "x86_64-w64-mingw32" and for 32-bit Windows it is "i686-w64-mingw32". For example:

   ```
   > configure_args_common='--target=msp430-elf --enable-languages=c,c++ --disable-nls
   --host=x86_64-w64-mingw32'
   ```

   Note

   You can confirm that a cross-compiler is available for the target host by running `${HOST}-gcc --version` . For example:

   ```
   > x86_64-w64-mingw32-gcc --version
   ```
5. Make sure the native toolchain installed earlier is on the PATH.
6. Run `README-build.sh` .

6.3 Building MSP430 GCC Stand-Alone Full Package

- MSP430 GCC Toolchain
  1. Download the MSP430 GCC Installer (Toolchain only) from http://www.ti.com/tool/msp430-gcc-opensource.
  2. Use the generated MSP430 GCC version (see Section 6.2).
- USB driver package (Windows only)
  1. Download "Stand-alone Driver Installer for USB Low-Level Drivers" from http://www.ti.com/tool/mspds.
- MSPDS OS package
  1. Download "MSP Debug Stack Open Source Package" from http://www.ti.com/tool/mspds.
- Build MSPDebugStack
  1. Extract "MSP Debug Stack Open Source Package".
  2. Follow the instructions in "README-build.sh".
- GDB agent
  1. Download the GDB Agent from the XDS Emulation Software (EMUPack) Download page.
- MSP430 support files for GCC
  1. Download "msp430-gcc-support-files.zip" from http://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/MSPGCC/latest/index_FDS.html.

Copyright © 2020 Texas Instruments Incorporated

7 MSP430 GCC and MSPGCC

The new GCC compiler for MSP low-power microcontrollers conforms to the MSP Embedded Application Binary Interface (EABI) (see MSP430 Embedded Application Binary Interface). This allows GCC to interoperate with the proprietary TI toolchain.

For example, assembly functions can be written in the same way, and libraries that are built with one compiler can be used as part of executables built with the other compiler. Aligning with the MSP EABI required breaking compatibility with the prior MSPGCC compiler. This document gives a brief overview of the ABI changes that are most likely to be noticed by and to affect a developer who is moving from MSPGCC to the newer GCC compiler for MSP.

7.1 Calling Convention

For developers writing assembly code, the most noticeable part of an ABI is the calling convention. Full specification of the calling convention is very detailed (see MSP430 Embedded Application Binary Interface), but developers writing assembly do not typically use most of it. There are three basic differences between MSPGCC and the GCC compiler for MSP in the calling convention that are important to be aware of:

- In MSPGCC, registers are passed starting with R15 and descending to R12. For example, if two integers are passed, the first is passed in R15 and the second is passed in R14. In contrast, the MSP430 EABI specifies that arguments are passed beginning with R12 and moving up to R15. So, in the same situation, registers R12 and R13 would hold the two arguments. In both cases, after the registers R12 through R15 are used, continued arguments are passed on the stack. If you are using stack-based arguments, you should consult the EABI specification.
- MSPGCC and the GCC compiler for MSP use different registers for the return value. MSPGCC places the return value in R15 (or R15 and consecutive lower registers if the value is larger than a word), while the EABI specifies that the return value is placed in R12.
- In MSPGCC, register R11 is considered a save on entry register and needs to be saved and restored by the callee if it is used in the called function. Conversely, the MSP EABI specifies that R11 is a save on call register, so it needs to be saved and restored by the calling function if its value will be needed after a function call. For comparison purposes, R4 to R10 are save on entry registers for both compilers, and R12 to R15 are save on call.

These are the key differences to be aware of when moving between the compilers. If you are writing assembly code that passes parameters on the stack or that passes structures by value, you should consult the MSP EABI document for additional information.

7.2 Other Portions of the ABI

Many other pieces make up the EABI, such as the object file format; debug information, and relocation information that is used when linking together files. However, in general, these pieces do not affect migration.

One other area to be aware of is that the details of data layout differ between ABIs. If you are relying on advanced data layout details such as layout of structures and bitfields, see MSP430 Embedded Application Binary Interface.

Copyright © 2020 Texas Instruments Incorporated

8 Appendix

8.1 GCC Intrinsic Support

The GCC Compiler supports the same intrinsics that the TI CGT for MSP430 does. These are:

- unsigned short **__bcd_add_short**(unsigned short op1, unsigned short op2);
- unsigned long **__bcd_add_long**(unsigned long op1, unsigned long op2);
- unsigned short **__bic_SR_register**(unsigned short mask); BIC mask, SR
- unsigned short **__bic_SR_register_on_exit**(unsigned short mask);
- unsigned short **__bis_SR_register**(unsigned short mask);
- unsigned short **__bis_SR_register_on_exit**(unsigned short mask);
- unsigned long **__data16_read_addr**(unsigned short addr);
- void **__data16_write_addr** (unsigned short addr, unsigned long src);
- unsigned char **__data20_read_char**(unsigned long addr);
- unsigned long **__data20_read_long**(unsigned long addr);
- unsigned short **__data20_read_short**(unsigned long addr);
- void **__data20_write_char**(unsigned long addr, unsigned char src);
- void **__data20_write_long**(unsigned long addr, unsigned long src);
- void **__data20_write_short**(unsigned long addr, unsigned short src);
- void **__delay_cycles**(unsigned long);
- void **__disable_interrupt**(void); AND **__disable_interrupts**(void);
- void **__enable_interrupt**(void); AND **__enable_interrupts**(void);
- unsigned short **__get_interrupt_state**(void);
- unsigned short **__get_SP_register**(void);
- unsigned short **__get_SR_register**(void);
- unsigned short **__get_SR_register_on_exit**(void);
- void **__low_power_mode_0**(void);
- void **__low_power_mode_1**(void);
- void **__low_power_mode_2**(void);
- void **__low_power_mode_3**(void);
- void **__low_power_mode_4**(void);
- void **__low_power_mode_off_on_exit**(void);
- void **__no_operation**(void);
- void **__set_interrupt_state**(unsigned short src);
- void **__set_SP_register**(unsigned short src);
- unsigned short **__swap_bytes**(unsigned short src);

Copyright © 2020 Texas Instruments Incorporated

8.2 NOP Instructions Required Between Interrupt State Changes

Incorrect execution of the MSP430 CPU can result from consecutive interrupt state changes (for example, EINT followed by DINT). The assembler detects such instruction patterns in code being assembled and emits warnings that NOP instructions might be required.

Since it is not always known what instruction will actually be executed after an EINT or DINT, the assembler warns if there is a NOP missing before/after every EINT or DINT, as appropriate for the device. The assembler also warns about instructions that modify the status register in a way that is unknown at assembly time, as these instructions might change the interrupt state.

Whether NOP instructions are required between interrupt state changes depends on the ISA the code is being assembled for. The assembler uses the following rules when deciding whether to warn about missing NOP instructions:

- Both the MSP430 and MSP430X ISA require a NOP after DINT.
- Only the MSP430X ISA requires a NOP before EINT.
- Only the MSP430X ISA requires a NOP after EINT.

See the user guide for your device family for more details.

9 References

1. Using the GNU Compiler Collection, Richard M. Stallman (http://gcc.gnu.org/onlinedocs/gcc.pdf). Refer to the *MSP430 Options* section.
2. GDB: The GNU Project Debugger, Free Software Foundation, Inc. (https://sourceware.org/gdb/current/onlinedocs/)

Revision History

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

Changes from June 30, 2019 to June 30, 2020	Page
• CCS v10.x is the current version	3
• Removed 32-bit Linux installer	29
• Added share directory, changed name of manifest file, and provided descriptions	30
• Python 2.7 is required to debug with GDB and GDB Agent	30
• Added -mmax-inline-shift and --symtab-meta options. Made corrections and additions to descriptions for -mhwmult, -mmcu, -msmall, and -my options	31
• Added -mQ assembler option and --relax/--no-relax linker options	31
• Explained how to set a hardware-assisted breakpoint on a function	36
• Using Ctrl+C to stop a running program is supported on Linux and macOS, but not Windows	36
• Software breakpoints are now used by default instead of hardware breakpoints in GDB Agent	37
• Added new section with descriptions of MSP430-specific GCC features	39
• Added location and retain function attributes and examples for interrupt function attributes	39
• Added location and retain data attributes	40
• Added information about code size and performance penalty for data placed in upper memory.	41
• Added section on startup behavior	42
• Added section on using printf	43
• Added section on link-time optimization	43
• Added section on the __int20 type and pointers in the large memory model	43

Copyright © 2020 Texas Instruments Incorporated

IMPORTANT NOTICE AND DISCLAIMER

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

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

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

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2020, Texas Instruments Incorporated