CodeWarrior™ Development Studio for Freescale™ 56800/E Digital Signal Controllers: DSP56F80x/DSP56F82x Family Targeting Manual

Revised: 20 June 2006

How to Contact Freescale

Corporate Headquarters	Freescale Corporation 7700 West Parmer Lane Austin, TX 78729 U.S.A.
World Wide Web	http://www.freescale.com/codewarrior
Technical Support	http://www.freescale.com/support

Table of Contents

1

Introduction

This manual explains how to use the CodeWarrior™ Integrated Development Environment (IDE) to develop code for the DSP56800 family of processors (DSP56F80x and the DSP56F82x).

This chapter contains the following sections:

- CodeWarrior IDE on page 13
- Freescale 56800/E Digital Signal Controllers on page 14
- References on page 16

CodeWarrior IDE

The CodeWarrior IDE consists of a project manager, a graphical user interface, compilers, linkers, a debugger, a source-code browser, and editing tools. You can edit, navigate, examine, compile, link, and debug code, within the one CodeWarrior environment. The CodeWarrior IDE lets you configure options for code generation, debugging, and navigation of your project.

Unlike command-line development tools, the CodeWarrior IDE organizes all files related to your project. You can see your project at a glance, so organization of your source-code files is easy. Navigation among those files is easy, too.

When you use the CodeWarrior IDE, there is no need for complicated build scripts of makefiles. To add files to your project or delete files from your project, you use your mouse and keyboard, instead of tediously editing a build script.

For any project, you can create and manage several configurations for use on different computer platforms. The platform on which you run the CodeWarrior IDE is called he host. host, you use the CodeWarrior IDE to develop code to target various platforms.

Note the two meanings of the term *target*:

- **Platform Target —** The operating system, processor, or microcontroller fin which/ on which your code will execute.
- **Build Target** — The group of settings and files that determine what your code is, as well as control the process of compiling and linking.

The CodeWarrior IDE lets you specify multiple build targets. For example, a project can contain one build target for debugging and another build target optimized for a particular operating system (platform target). These build targets can share files, even though each build target uses its own settings. After you debug the program, the only actions necessary to generate a final version are selecting the project's optimized build target and using a single Make command.

The CodeWarrior IDE's extensible architecture uses plug-in compilers and linkers to target various operating systems and microprocessors. For example, the IDE uses a GNU tool adapter for internal calls to DSP56800 development tools.

Most features of the CodeWarrior IDE apply to several hosts, languages, and build targets. However, each build target has its own unique features. This manual explains the features unique to the CodeWarrior Development Studio for Freescale 56800.

For comprehensive information about the CodeWarrior IDE, see the *CodeWarrior IDE User's Guide*.

NOTE For the very latest information on features, fixes, and other matters, see the *CodeWarrior Release Notes*, on the CodeWarrior IDE CD.

Freescale 56800/E Digital Signal Controllers

The Freescale 56800/E Digital Signal Controllers consist of two sub-families, which are named the DSP56F80x/DSP56F82x (DSP56800) and the MC56F8xxx/DSP5685x (DSP56800E). The DSP56800E is an enhanced version of the DSP56800.

The processors in the DSP56800 and DSP56800E sub-families are shown in Table 1.1 on page 14.

With this product the following Targeting Manuals are included:

- *Code Warrior Development Studio for Freescale 56800/E Digital Signal Controllers: DSP56F80x/DSP56F82x Family Targeting Manual*
- *Code Warrior Development Studio for Freescale 56800/E Digital Signal Controllers: MC56F8xxx/DSP5685x Family Targeting Manual*

NOTE Please refer to the Targeting Manual specific to your processor.

Table 1.1 Supported DSP56800x Processors for CodeWarrior Development Studio for Freescale 56800

DSP56800	DSP56800E
DSP56F801 (60 MHz)	DSP56852
DSP56F801 (80 MHz)	DSP56853

Table 1.1 Supported DSP56800x Processors for CodeWarrior Development Studio for Freescale 56800 (*continued*)

DSP56800	DSP56800E
DSP56F802	DSP56854
DSP56F803	DSP56855
DSP56F805	DSP56857
DSP56F807	DSP56858
DSP56F826	MC56F8013
DSP56F827	MC56F8014
	MC56F8023
	MC56F8025
	MC56F8036
	MC56F8037
	MC56F8122
	MC56F8123
	MC56F8145
	MC56F8146
	MC56F8147
	MC56F8155
	MC56F8156
	MC56F8157
	MC56F8165
	MC56F8166
	MC56F8167
	MC56F8322
	MC56F8323
	MC56F8335

Table 1.1 Supported DSP56800x Processors for CodeWarrior Development Studio for Freescale 56800 (*continued*)

DSP56800	DSP56800E
	MC56F8345
	MC56F8346
	MC56F8356
	MC56F8357
	MC56F8365
	MC56F8366
	MC56F8367

References

- Your CodeWarrior IDE includes these manuals:
  - *CodeWarrior™ IDE User’s Guide*
  - *CodeWarrior™ Development Studio IDE 5.6 Windows® Automation Guide*
  - *CodeWarrior™ Development Studio for Freescale 56800/E Digital Signal Controllers: DSP56F80x/DSP56F82x Family Targeting Manual*
  - *CodeWarrior™ Development Studio for Freescale 56800/E Digital Signal Controllers: MC56F8xxx/DSP5685x Family Targeting Manual*
  - *CodeWarrior™ Builds Tools Reference for Freescale 56800/E Digital Signal Controllers*
  - *CodeWarrior™ Development Studio IDE 5.5 User’s Guide Profiler Supplement*
  - *CodeWarrior™ Development Studio for Freescale™ DSP56800x Embedded Systems Assembler Manual*
  - *Codewarrior™ USB TAP Users Guide*
  - *Freescale™ 56800 Family IEEE - 754 Compliant Floating-Point Library User Manual*
  - *Freescale™ 56800E Family IEEE - 754 Compliant Floating-Point Library User Manual*
  - *CodeWarrior™ Development Studio HTI Host Target Interface (for Once™/ JTAG Communication) User’s Manual*
  - *DSP56800 to DSP56800E Porting Guide,* Freescale Semiconductors, Inc.

- *56F807 to 56F8300/56F8100 Porting User Guide*, Freescale Semiconductors Inc.
- To learn more about the DSP56800E processor, refer to the Freescale manual, *DSP56800E Family Manual*.

• To learn more about the DSP56800 processor, refer to the following manuals:

- *DSP56800 Family Manual.* Freescale Semiconductors, Inc.
- *DSP56F801 Hardware User Manual*. Freescale Semiconductors, Inc.
- *DSP56F803 Hardware User Manual*. Freescale Semiconductors, Inc.
- *DSP56F805 Hardware User Manual*. Freescale Semiconductors, Inc.
- *DSP56F807 Hardware User Manual*. Freescale Semiconductors, Inc.
- *DSP56F826 Hardware User Manual*. Freescale Semiconductors, Inc.
- *DSP56F827 Hardware User Manual*. Freescale Semiconductors, Inc.

• For more information on the various command converters supported by the CodeWarrior Development Studio for Freescale 56800 (DSP56F80x/DSP56F82x), refer to the following manuals:

- *Suite56™ Ethernet Command Converter User's Manual*, Freescale Semiconductors, Inc.
- *Suite56™ PCI Command Converter User's Manual*, Freescale Semiconductors, Inc.
- *Suite56™ Parallel Port Command Converter User's Manual*, Freescale Semiconductors, Inc.

To download electronic copies of these manuals or order printed versions, visit:

`http://www.freescale.com/`

2

Getting Started

This chapter explains how to install and run the CodeWarrior™ IDE on your Windows® operating system. This chapter also explains how to connect hardware for each of the communications protocols supported by the CodeWarrior debugger.

This chapter contains the following sections:

- System Requirements on page 19
- Installing and Registering the CodeWarrior IDE on page 20
- Installing DSP56800 Hardware on page 24

System Requirements

Table 2.1 on page 19 lists system requirements for installing and using the CodeWarrior IDE for DSP56800.

Table 2.1 Requirements for the CodeWarrior IDE

Category	Requirement
Host Computer Hardware	PC or compatible host computer with 1.0-GHz Pentium®-compatible processor, 512 megabytes of RAM, and a CD-ROM drive
Operating System	Microsoft® Windows® 2000/XP
Hard Drive	2.0 gigabytes of free space, plus space for user projects and source code

DSP56800 Hardware Requirements

You can use various DSP56800 hardware configurations with the CodeWarrior IDE. Table 2.2 on page 20 lists these configurations.

NOTE Each protocol in Table 2.2 on page 20 is selected from the **M56800 Target Settings** panel.

Table 2.2 DSP56800 Hardware Requirements

Target Connection	Boards Supported	Hardware Provided With Command Converter
Parallel port on-board Command Converter	All 56800 targets	• 25-pin parallel-port interface cable • Power supply, 9–12 Vdc, 500 mA with 2.5 mm receptacle (inside positive)
External Parallel Port Command Converter	All 56800 targets	• Freescale Parallel Port Command Converter • 25-pin parallel-port interface cable
PCI Command Converter	All 56800 targets	• 25-pin OCD ribbon cable • Target Interface Module • JTAG 14-pin ribbon interface cable

Installing and Registering the CodeWarrior IDE

Follow these steps:

1. To install the CodeWarrior software:

   a. Insert the CodeWarrior CD into the CD-ROM drive — the welcome screen appears.

NOTE If the Auto Install is disabled, run the program `Setup.exe` in the root directory of the CD.

   b. Click **Launch CodeWarrior Setup** — the install wizard displays welcome page.

   c. Follow the wizard instructions, accepting all the default settings.

   d. At the prompt to check for updates, click the **Yes** button — the CodeWarrior updater opens.

2. To check for updates:

NOTE If the updater already has Internet connection settings, you may proceed directly to substep f.

a. Click the **Settings** button — the **Updater Settings** dialog box appears.

b. Click the **Load Settings** button — the updater loads settings from your Windows control panel.

c. Modify the settings, as appropriate.

d. If necessary, enter the proxy username and the password.

e. Click the **Save** button — the **Updater Settings** dialog box disappears.

f. In the updater screen, click the **Check for Updates** button.

g. If updates are available, follow the on-screen instructions to download the updates to your computer.

h. When you see the message, “Your version … is up to date”, click the **OK** button — the message box closes.

i. Click the updater **Close** button — the installation resumes.

j. At the prompt to restart the computer, select the **Yes** option button.

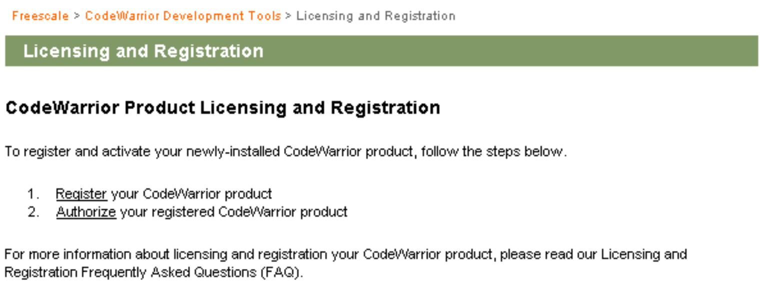
k. Click the **Finish** button — the computer restarts, completing installation.

3. To register the CodeWarrior software:

a. Select **Start> Programs>Freescale CodeWarrior> CW for DSC56800 R8.0>CodeWarrior IDE**.

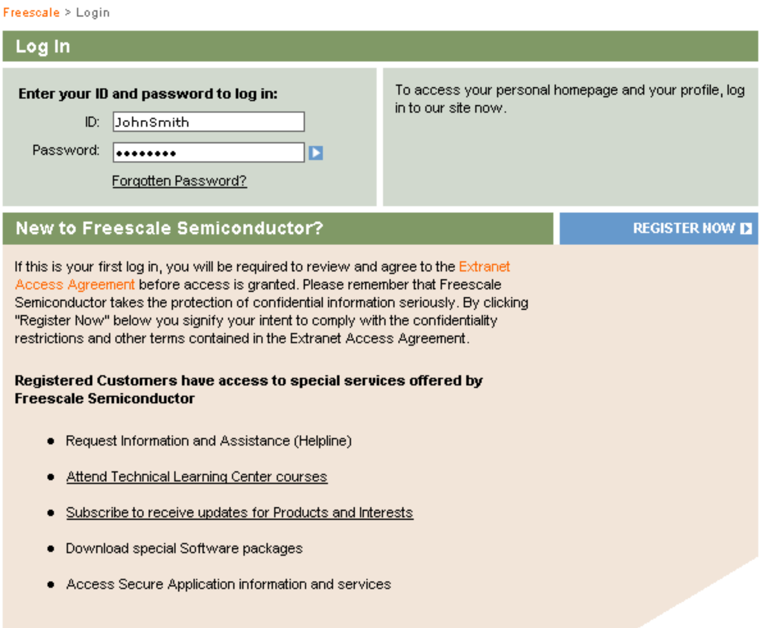
b. Select **Help > Register Product** — the Freescale registration page appears.

Figure 2.1 Freescale Registration Page

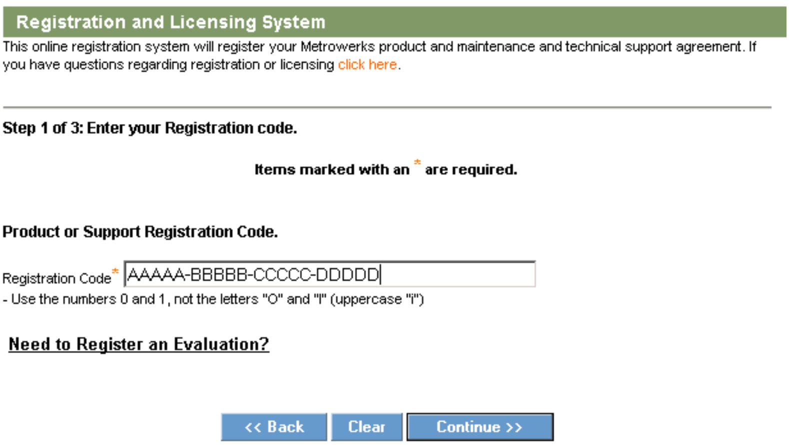


c. Click item number 1 — Register your CodeWarrior product.

d. Login or Register on the Freescale site.



e. Enter your Registration Code and click the Continue button.

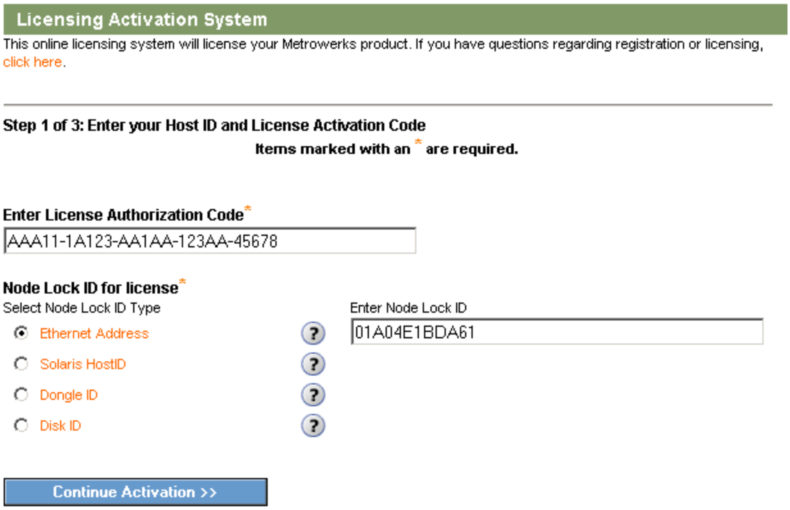


NOTE In the next screen you will be asked to confirm your Registration code by clicking Continue a second time. After registration is complete, you will receive an email with the activation code and directions on how to activate your product.

f. Click the Activation link in the email that you receive.

g. Login to the Freescale site.

h. Enter the License Authorization Code into the field.



i. For the default selection: Ethernet Address — determine your ethernet address:

Launch a Command Prompt window

Enter `ipconfig /all`

Copy the Physical Address value of the first Ethernet adapter listed

Paste value into the "Node Lock ID for license" text box (remove spaces or dashes)

j. Click Continue Activation.

k. Click Continue to confirm the Host ID and License Authorization Code.

The website will display your license keys along with instructions on installing the license. Copy and paste these keys into the top of the "license.dat" file located at the root of your CodeWarrior installation directory. Your product should now be fully licensed and operational.

Table 2.3 on page 24 lists the directories created during full installation.

To test your system, follow the instructions of the next section to create a project.

Table 2.3 Installation Directories, CodeWarrior IDE for DSP56800

Directory	Contents
(CodeWarrior_Examples)	Target-specific projects and code.
(Helper Apps)	Applications such as cwspawn.exe and cvs.exe.
bin	The CodeWarrior IDE application and associated plug-in tools.
ccs	Command converter server executable files and related support files.
DSP 56800x_EABI_Support	Default files used by the CodeWariior IDE for the DSP56800 stationery.
DSP56800x_EABI_Tools	Drivers to the CCS and command line tools, plus IDE default files for the DSP56800x stationery.
Freescale_Documentation	Documentation specific to the Freescale DSP56800 series.
Help	Core IDE and target-specific help files. (Access help files through the Help menu or F1 key.)
License	Licensing information.
Lint	Support for PCLint.
M56800 Support	Initialization files, Metrowerks Standard Library (MSL) and Runtime Library.
M56800x Support	Profiler libraries.
ProcessorExpert	Files for the Processor Expert.
Release_Notes	Release notes for the CodeWarrior IDE and each tool.
Stationery	Templates for creating DSP56800 projects. Each template pertains to a specific debugging protocol.

Installing DSP56800 Hardware

This section explains how to connect the DSP568xx hardware to your computer. Parallel port connections are explained in the *Kit Installation Guide* for each individual DSP568xxEVM board. All descriptions assume the default jumper settings, as explained in the *Hardware Manual* for your product, unless otherwise stated.

NOTE You can use the DSP56800 Simulator provided with the CodeWarrior IDE instead of installing additional DSP568xx hardware. However, the DSP56800 Simulator is a core simulator and will not give you product specific features (such as peripherals, specialized memory maps, etc.)

Using Parallel Port

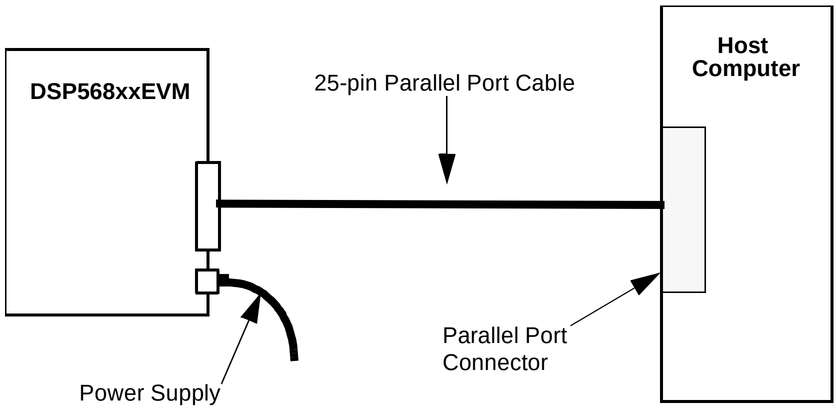
Connect the parallel port cable to your DSP568xxEVM board as described below.

Connecting via the on board Parallel Command Converter on DSP568xxEVM Board

1. Connect the 25-pin male connector at one end of a parallel port cable to the 25-pin female connector on your computer (Figure 2.2 on page 25).
2. Connect the 25-pin female connector at the other end of the parallel port cable to the 25-pin male connector on the DSP568xxEVM.
3. Plug the power supply into a wall socket.
4. Connect the power supply to the power connector on the DSP568xxEVM board.

   The green LED next to the power connector lights up.

Figure 2.2 Connecting Parallel Port Cable to DSP568xxEVM Board



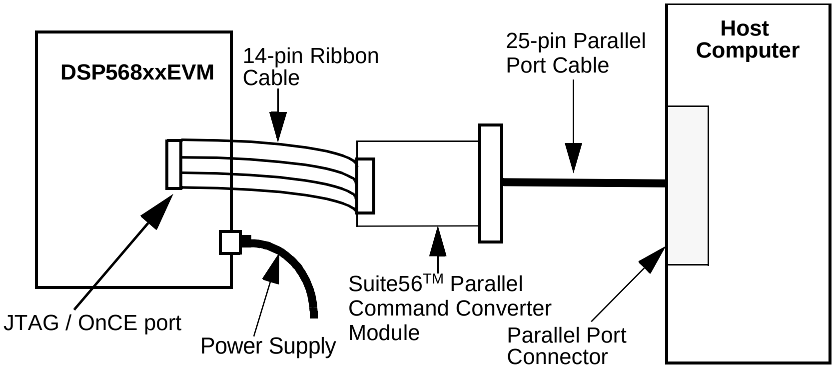
Connecting via the Suite56™ Parallel Port Command Converter Module and DSP568xxEVM Board

1. Enable the JTAG port.

   See the *Hardware Manual* or *Kit Installation Guide* for the jumpers that you need to change from the default configuration for your particular hardware.

2. Connect the 25-pin male connector at one end of a parallel port cable to the 25-pin female connector on your computer (Figure 2.3 on page 26).

Figure 2.3 Connecting Parallel Port Cable to Suite56TM Parallel Command Converter Module and DSP568xxEVM Board



3. Connect the 25-pin female connector at the other end of the parallel port cable to the 25-pin male connector on the Suite56TM Parallel Port Command Converter module.

4. Locate the 14-pin ribbon cable hanging from the Suite56TM Parallel Port Command Converter module. Connect the 14-pin female connector of the ribbon cable to the 14-pin JTAG male connector on the DSP568xxEVM board.

   Ensure that the red stripe on the ribbon cable corresponds to pin 1 on the DSP568xxEVM card.

5. Plug the power supply into a wall socket.

6. Connect the power supply to the power connector on the DSP568xxEVM board.

   The green LED next to the power connector lights up.

Installing the PCI Command Converter

Connect the PCI Command Converter and your Freescale DSP568xxEVM board to your computer as described below.

Installing the PCI Command Converter

Install the PCI Command Converter hardware:

1. Place your PCI Command Converter card on a static-proof mat.
2. Shut down your computer.

WARNING! Do not touch the components and connectors on the board or inside your computer without first being grounded. Otherwise, you could damage the hardware with static discharge.

3. Locate an empty card slot in your computer.
4. Insert the PCI Command Converter card in the empty card slot.

NOTE One end of the 25-pin cable has a 24-pin female connector. A ground cable is retrofitted to a wire of the 25-pin cable at the same end of the cable. The ground cable is crimped to a female disconnect terminal.

5. Connect the 24-pin female connector at one end of the 25-pin cable to the 24-pin female connector on the PCI Command Converter card (Figure 2.4 on page 29).
6. Connect the female disconnect terminal of the ground cable to the socket protruding from the PCI Command Converter card in your computer.
7. Connect the 25-pin female connector at the other end of the 25-pin cable to the 25-pin male connector on the OCDemon™ Wiggler.

Procedure for Manual Installation of PCI Command Converter Drivers

Windows® 2000/ Windows® XP

The required files are located in the following directory:

\Program Files\Freescale\CodeWarrior\ccs\drivers\pci

1. Copy the `*.inf` file to /winnt/inf.
2. Copy the `mac_mot.sys` file to /winnt/system32/drivers.
3. Copy the `windrvr.sys` file to /winnt/system32/drivers.
4. Install the `*.inf` file by right-clicking on this file and selecting the Install button.
5. From the command prompt, change to the following directory:

   \Program Files\Freescale\CodeWarrior\ccs\drivers\pci

6. Type the following:

```
wdreg -file mac_mot remove
wdreg remove
wdreg install
wdreg -file mac_mot install
```

7. Shut down your computer.
8. Turn on your computer.

Connecting the PCI Command Converter to the DSP568xxEVM Board

To connect the PCI Command Converter to your DSP568xxEVM board, follow the steps explained in "Installing the PCI Command Converter" on page 27 before performing the steps in this section.

Connect the PCI Command Converter to your DSP568xxEVM board:

1. Enable the JTAG port.

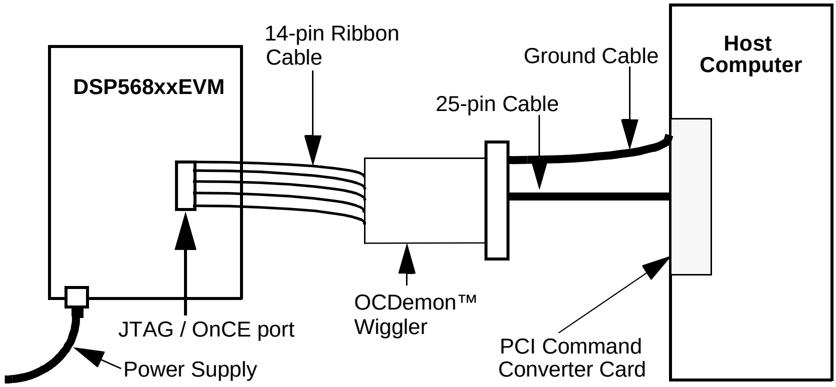
   See the *Hardware Manual* or *Kit Installation Guide* for the jumpers that you need change from the default configuration for your particular hardware.

2. Locate the 14-pin ribbon cable hanging from the OCDemon™ Wiggler. Connect the 14-pin female connector of the ribbon cable to the 14-pin JTAG male connector on the DSP568xxEVM board.

   Ensure that the red stripe on the ribbon cable corresponds to pin 1 on the DSP568xxEVM card.

3. Plug the power supply into a wall socket.
4. Connect the power supply to the power connector on the DSP568xxEVM board.
5. The green LED next to the power connector lights up. The board is now connected.

Figure 2.4 Attaching PCI Command Converter to DSP568xxEVM Board



3

Development Studio Overview

This chapter is for new users of the CodeWarrior™ IDE. This chapter contains the following sections:

- CodeWarrior IDE on page 31
- Development Process on page 32

If you are an experienced CodeWarrior IDE user, you will recognize the look and feel of the user interface. However, it is necessary that you become familiar with the DSP56800 runtime software environment.

CodeWarrior IDE

The CodeWarrior IDE lets you create software applications. It controls the project manager, the source-code editor, the class browser, the compiler, linker, and the debugger.

In the project manager, you can organize all the files and settings related to your project so that you can see your project at a glance and easily navigate among your source-code files. The CodeWarrior IDE automatically manages build dependencies.

A project can have multiple build targets. A build target is a separate build (with its own settings) that uses some or all of the files in the project. For example, you can have both a debug version and a release version of your software as separate build targets within the same project.

The CodeWarrior IDE has an extensible architecture that uses plug-in compilers and linkers to target various operating systems and microprocessors. The CodeWarrior CD includes a C compiler for the DSP56800 family of processors. Other CodeWarrior software packages include C, C++, and Java compilers for Win32, Linux, and other hardware and software combinations.

CodeWarrior Compiler for DSP56800

The CodeWarrior compiler for DSP56800 is an ANSI-compliant C compiler. This compiler is based on the same compiler architecture used in all CodeWarrior C compilers. When it is used together with the CodeWarrior linker for DSP56800, you can generate DSP56800 applications and libraries.

NOTE The CodeWarrior compiler for DSP56800 does not support C++.

CodeWarrior Assembler for DSP56800

The CodeWarrior assembler for DSP56800 has an easy-to-use syntax. The CodeWarrior IDE assembles any file with an `.asm` extension in the project. For information on features and syntax of the assembler, refer to the *Code Warrior Development Studio Freescale DSP56800x Embedded Systems Assembler Manual*. For opcode listings, refer to the *DSP56800 Family Manual*.

CodeWarrior Linker for DSP56800

The CodeWarrior linker for Freescale DSP56800 is in an Executable and Linker Format (ELF) linker. This linker lets you generate an ELF file (the default output file format) for your application and generate an S-record output file for your application.

CodeWarrior Debugger for DSP56800

The CodeWarrior debugger controls your program's execution and lets you see what happens internally as your program runs. Use the debugger to find problems in your program's execution.

The debugger can execute your program one statement at a time and suspend execution when control reaches a specified point. When the debugger stops a program, you can view the chain of function calls, examine and change the values of variables, inspect the contents of the processor's registers and see the contents of memory.

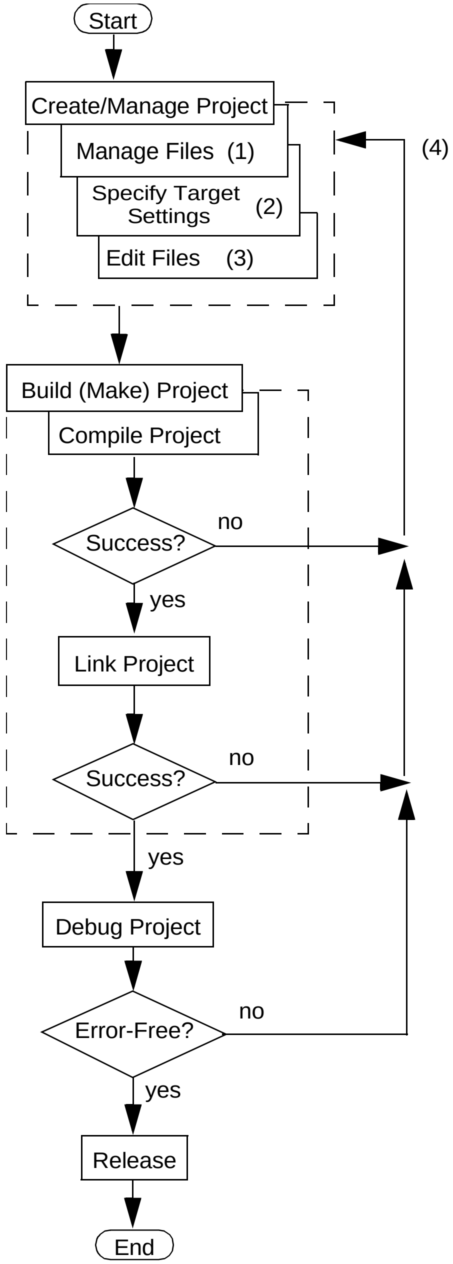
Metrowerks Standard Library

The Metrowerks Standard Library (MSL) is a set of standard C libraries for use in developing DSP56800 applications. These libraries are ANSI-compliant. Access the library sources for use in your projects. These libraries are a subset of the same ones used for all platform targets, but the libraries have been customized and the runtime adapted for use in DSP56800 development.

Development Process

The CodeWarrior IDE helps you manage your development work more effectively than you can with a traditional command-line environment. Figure 3.1 on page 33 depicts application development using the IDE.

Figure 3.1 CodeWarrior IDE Application Development



Notes:

(1) Use any combination: stationery (template) files, library files, or your own source files.

(2) Compiler, linker, debugger settings; target specification; optimizations.

(3) Edit source and resource files.

(4) Possible corrections: adding a file, changing settings, or editing a file.

Project Files versus Makefiles

The CodeWarrior IDE *project* is analogous to a collection of makefiles because you can have multiple builds in the same project. For example, you can have one project that maintains both a debug version and a release version of your code. You can build either or both of these versions as you wish. Different builds within a single project are called "build targets."

The CodeWarrior IDE uses the project window to list the files in a project. A project can contain various types of files, such as source-code files and libraries.

You can easily add or remove files from a project. You can assign files to one or more build targets within the same project. These assignments let you easily manage files common to multiple build targets.

The CodeWarrior IDE automatically handles the dependencies between files, and it tracks which files have changed since the last build. When you rebuild a project, only those files that have changed are recompiled.

The CodeWarrior IDE also stores compiler and linker settings for each build target. You can modify these settings by changing the options in the target settings panels of the CodeWarrior IDE or by using `#pragma` statements in your code.

Editing Code

The CodeWarrior IDE features a text editor. It handles text files in MS-DOS®/Windows,® and UNIX formats.

To open and edit a source-code file, or any other editable file in a project, use either of the following options:

- Double-click the file in the project window.
- Click the file. The file is highlighted. Drag the file to the Freescale CodeWarrior IDE window.

The editor window has excellent navigational features that allow you switch between related files, locate any particular function, mark any location within a file, or go to a specific line of code.

Compiling

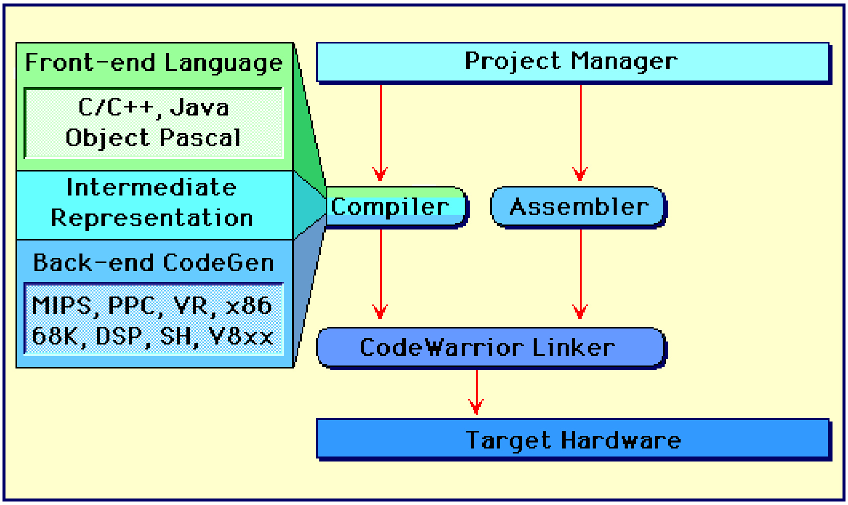
To compile any source-code file in the current build target, select the source-code file in the project window and then select **Project > Compile** from the menu bar of the Freescale CodeWarrior window.

To compile all the files in the current build target that were modified since they were last compiled, select **Project >Bring Up To Date** from the menu bar of the Freescale CodeWarrior window.

In UNIX and other command-line environments, object code compiled from a source-code file is stored in a binary file (a `.o` or `.obj` file). On Windows targets, the CodeWarrior IDE stores and manages object files internally in the data folder.

A proprietary compiler architecture is at the heart of the CodeWarrior IDE. This architecture handles multiple languages and platform targets. Front-end language compilers generate an intermediate representation (IR) of syntactically correct source code. The IR is memory-resident and language-independent. Back-end compilers generate code from the IR for specific platform targets. The CodeWarrior IDE manages the whole process (Figure 3.2 on page 35).

Figure 3.2 CodeWarrior Build System



As a result of this architecture, the CodeWarrior IDE uses the same front-end compiler to support multiple back-end platform targets. In some cases, the same back-end compiler can generate code from a variety of languages. Users derive significant benefit from this architecture. For example, an advance in the C/C++ front-end compiler means an immediate advance in all code generation. Optimizations in the IR mean that any new code generator is highly optimized. Targeting a new processor does not require compiler-related changes in the source code, so porting is much simpler.

All compilers are built as plug-in modules. The compiler and linker components are modular plug-ins. Freescale publishes this API, allowing developers to create custom or proprietary tools. For more information, go to Freescale Support at this URL:

http://www.Freescale.com/MW/Support

Once the compiler generates object code, the plug-in linker generates the final executable file. Multiple linkers are available for some platform targets to support different object-code formats.

Linking

To link object code into a final binary file, select **Project > Make** from the menu bar of the Freescale CodeWarrior window. The **Make** command brings the active project up to date, then links the resulting object code into a final output file.

The CodeWarrior IDE controls the linker through the use of linker command files. There is no need to specify a list of object files. The Project Manager tracks all the object files automatically. You can also use the Project Manager to specify link order.

The **Target>M56800 Target** settings panel lets you set the name of the final output file.

Debugging

To debug a project, select **Project** > **Debug** from the menu bar of the Freescale CodeWarrior window.

Viewing Preprocessor Output

To view preprocessor output, select the file in the project window and select **Project** > **Preprocess** from the menu bar of the Freescale CodeWarrior window. The CodeWarrior IDE displays a new window that shows you what your file looks like after going through the preprocessor.

You can use this feature to track down bugs caused by macro expansion or other subtleties of the preprocessor.

4

Tutorial

This chapter gives you a quick start at learning how to use the CodeWarrior™ Development Studio for Freescale™ DSC56800/E for the DSP56F80x/DSP56F82x Controllers.

CodeWarrior Development Studio for Freescale 56800 Tutorial

This chapter provides a tour of the software development environment of the CodeWarrior Development Studio for Freescale 56800. You will learn how to use the tools to program for DSP56800 boards.

This tutorial introduces you to many important elements of the CodeWarrior IDE that you will use when programming for DSP56800. However, the tutorial does not cover or explain all the features of the IDE.

You will learn how to create, compile, and link code that runs on DSP56800 systems.

If you are already familiar with the CodeWarrior software, read through the steps in this tutorial anyway. You will encounter the DSP56800 compiler and linker for the first time, as well as other features specific to DSP56800 application development.

This tutorial is divided into segments. In each segment, you will perform steps that introduce you to the critical elements of the CodeWarrior IDE programming environment. The segments are:

- Creating a Project on page 37
- Working with the Debugger on page 52

Creating a Project

You can create a DSP56800x project by using the:

- DSP56800x new project stationery wizard
- DSP56800x EABI stationery

To create a new project with the DSP56800x new project wizard, please see the sub-section "Creating a New Project with the DSP56800x New Stationery Project Wizard."

To create a new project with the DSP56800x EABI stationery, please see the sub-section "Creating a New Project with the DSP56800x EABI Stationery."

Creating a New Project with the DSP56800x New Stationery Project Wizard

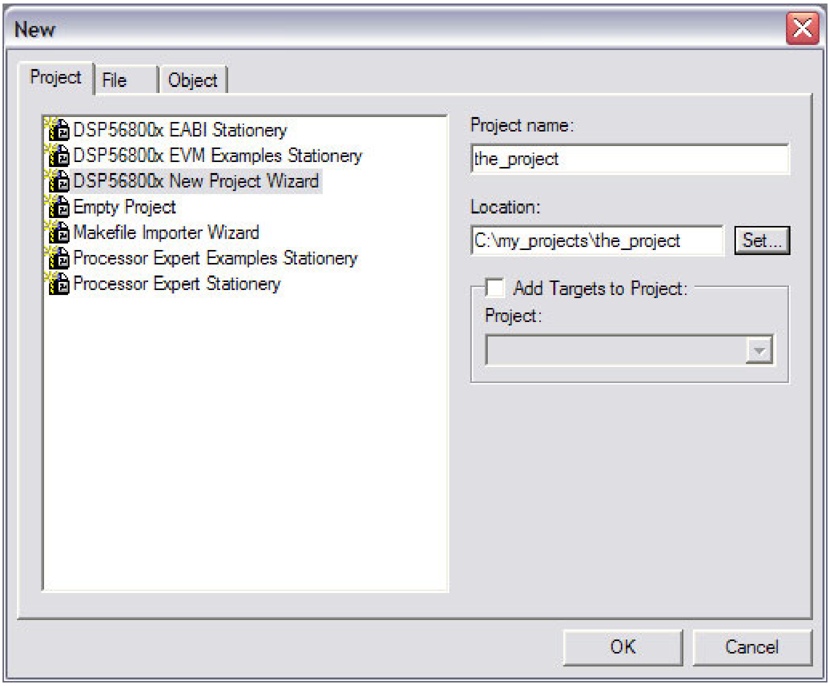
In this section of the tutorial, you work with the CodeWarrior IDE to create a project. with the DSP56800x New Stationery Project Wizard.

To create a project:

1. From the menu bar of the Freescale CodeWarrior window, select **File>New**.

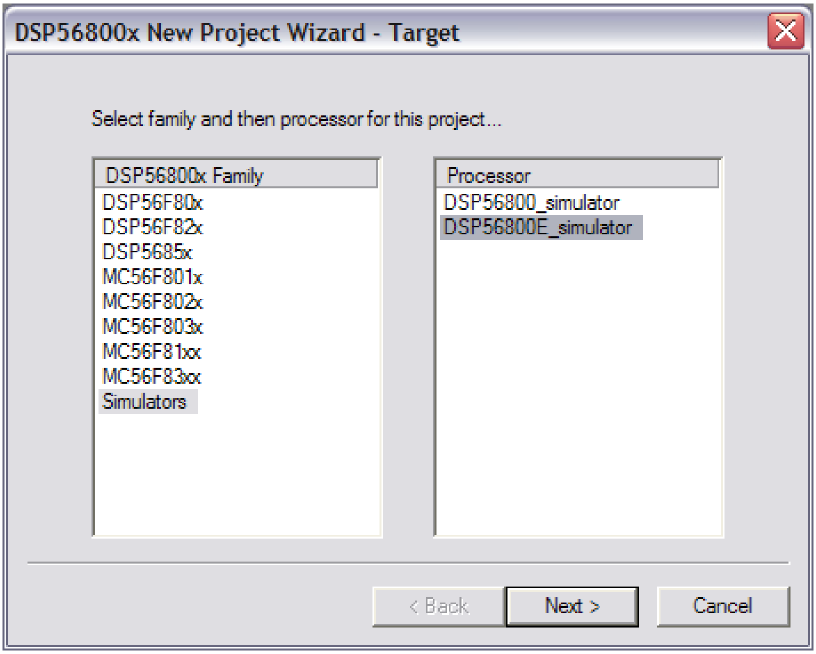
   The **New** dialog box appears.

Figure 4.1 New Dialog Box



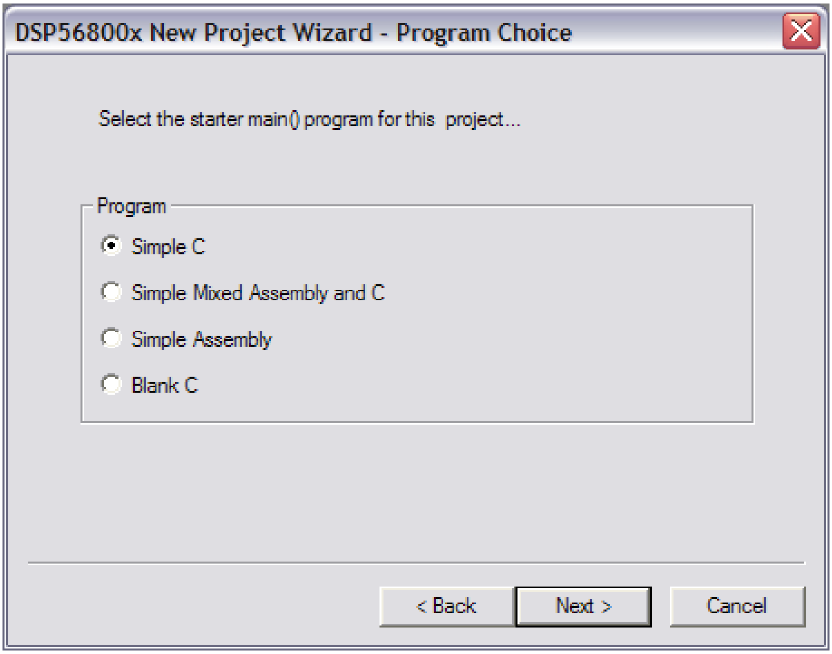
2. Select **DSP56800x New Project Wizard** (Figure 4.2 on page 39).
3. In the **Project Name** text box, type the project name. For example, the_project.
4. In the **Location** text box, type the location where you want to save this project or choose the default location.
5. Click **OK**. The **DSP56800x New Project Wizard — Target** dialog box (Figure 4.2 on page 39) appears.

Figure 4.2 DSP56800x New Project Wizard — Target Dialog Box



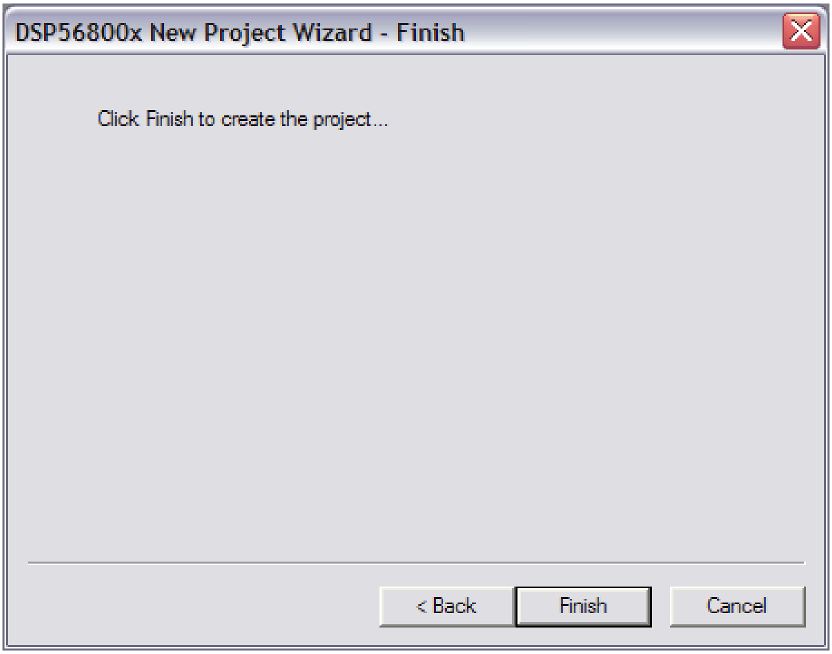
6. Select the target board and processor
   a. Select the family, such as Simulators, from the **DSP56800x Family** list.
   b. Select the processor, such as DSP56800_simulator, from the **Processors** list.
7. Click **Next**. The **DSP56800x New Project Wizard — Program Choice** dialog box (Figure 4.3 on page 40) appears.

Figure 4.3 DSP56800x New Project Wizard — Program Choice Dialog Box



8. Select the example main[] program for this project, such as Simple C.
9. Click **Next**. The **DSP56800x New Project Wizard — Finish** dialog box () appears.

Figure 4.4 DSP56800x New Project Wizard — Finish Dialog Box



10. Click **Finish** to create the new project.

NOTE For more details of the DSP56800x New Project Stationery Wizard, please see "DSP56800x New Project Wizard"..

Creating a New Project with the DSP56800x EABI Stationery

In this section of the tutorial, you work with the CodeWarrior IDE to create a project with the DSP56800x EABI Stationery.

You will start using a project stationery. A project stationery file is a template that describes a pre-built project, complete with source-code files, libraries, and all the appropriate compiler and linker settings. When you create a project based on stationery, the stationery is duplicated and becomes the basis of your new project.

You can create customized project stationery as well. Project stationery is a useful feature of the CodeWarrior IDE.

Practice working with a sample project as follows:

1. Launch the CodeWarrior IDE.

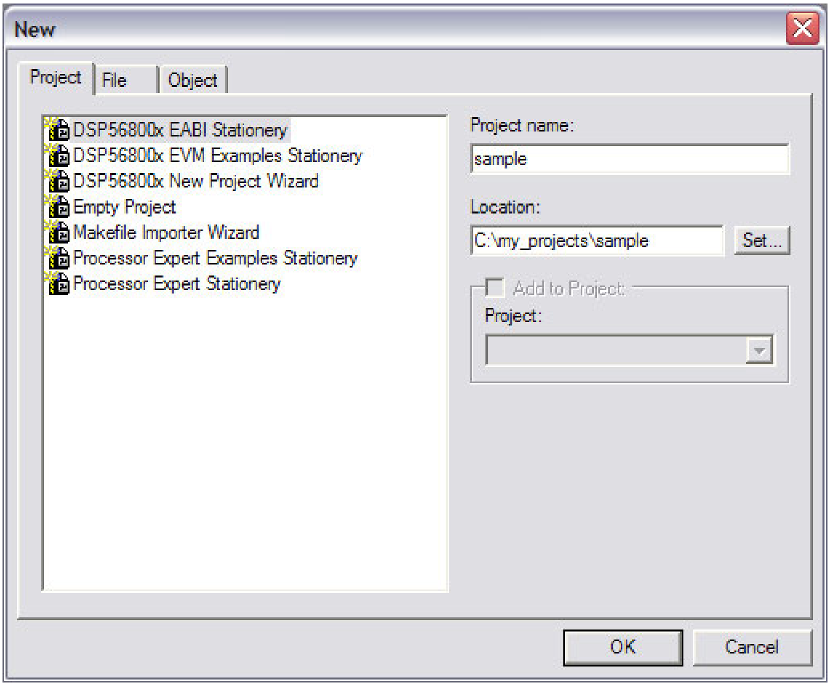
   The Freescale CodeWarrior window appears with a menu bar at the top.

To create a new project from project stationery:

1. From the menu bar of the Freescale CodeWarrior window, select **File > New**.

   The **New** window appears with a list of options in the **Project** tab (Figure 4.5 on page 42).

Figure 4.5 New Window



2. Select **DSP56800x EABI Stationery** in the **Project** tab.

NOTE To create a new project without using stationery, select **Empty Project** in the **New** window. This option lets you create a project from scratch. If you are a beginner, you should not try to use an **Empty Project** as it will not have any of the necessary target settings, startup files, or libraries included that are specific to the DSP56800 that allow you to quickly get up and running. This is why we include the DSP56800x_EABI Stationery, as it takes care of these tasks and minimizes the startup effort that is required.

---

3. Type a name in the **Project name** field (in this tutorial use “sample” as the name).

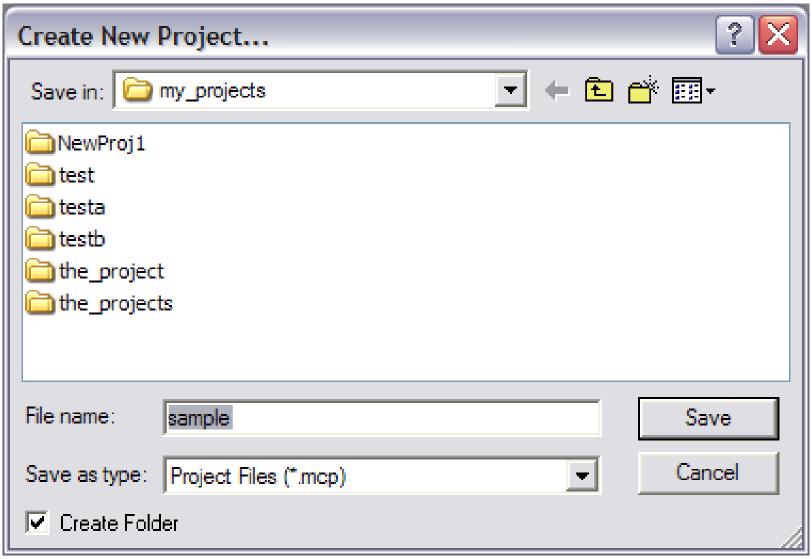
   The CodeWarrior IDE adds the `.mcp` extension automatically to your file when the project is saved. The `.mcp` extension allows any user to recognize the file as a Freescale CodeWarrior project file. In this tutorial, the file name is `sample.mcp`.

4. Set the location for the project.

   If you want to change the default location, perform the following steps:

   a. In the **New** window, click the **Set** button. The **Create New Project** dialog box (Figure 4.6 on page 43) appears:

Figure 4.6 Create New Project Dialog Box



   b. Use the standard navigation controls in the **Create New Project** dialog box to specify the path where you want the project file to be saved.

   c. Click the **Save** button. The CodeWarrior IDE closes the **Create New Project** dialog box.

   If you want to use the default location for your project, go to step 5.

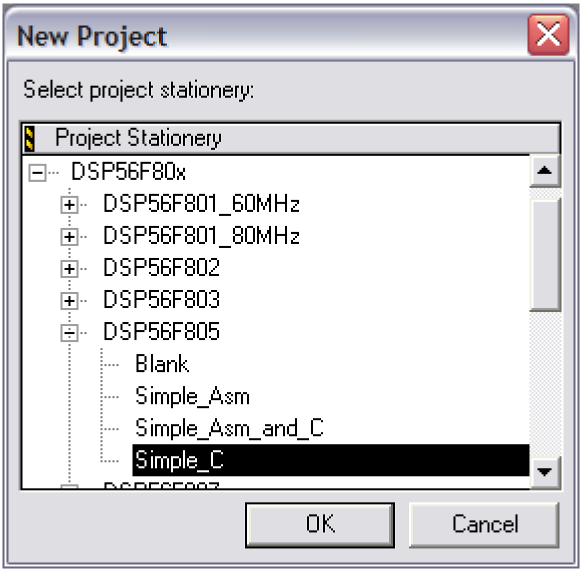
   In either case, the CodeWarrior IDE creates a folder with the same name as your project in the directory you select.

NOTE Enable the **Create Folder** checkbox in the **Create New Project** file dialog box to create a new folder for your project in the selected location.

5. Click **OK** in the **New** window.

   The **New Project** window appears (Figure 4.7 on page 44) with a list of board-specific project stationeries.

Figure 4.7 New Project Window



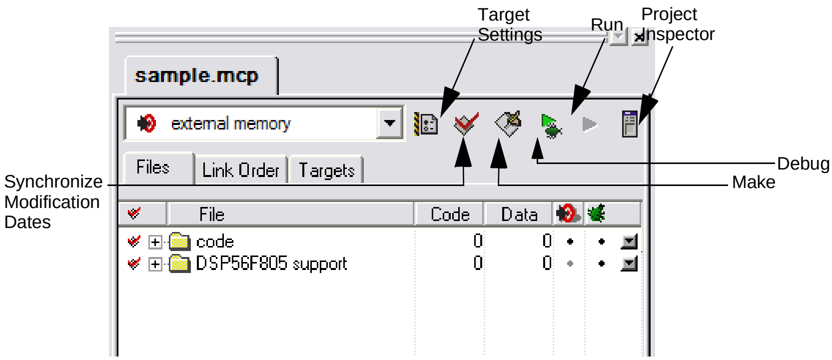
6. Select **DSP56F80x** as the Project Stationery for your target.

   Click the hierarchical control for the Project Stationery to expand the hierarchical view. Then, select **DSP56F805** and select **Simple_C** language from the hierarchical tree.

7. Click **OK** in the **New Project** window.

   A project window appears (Figure 4.8 on page 45). This window displays all the files and libraries that are part of the project stationery.

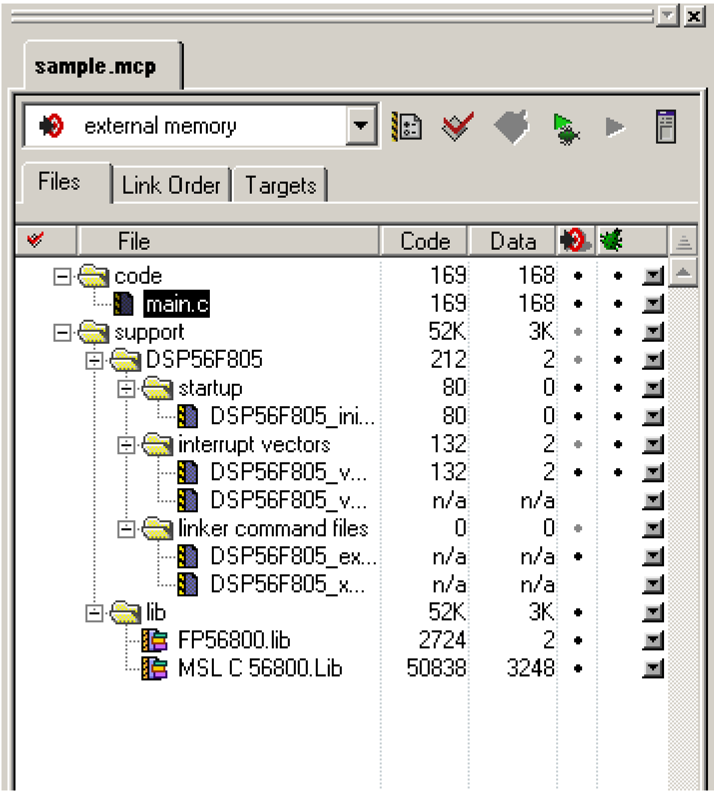
Figure 4.8 CodeWarrior Project Window



The project window is the central location from which you control development. You can use this window to:

- Add or remove source files
- Add libraries of code
- Compile code
- Generate debugging information and much more
- Confirm that the **Files** tab is selected in the project window (it should be selected by default).
- Click the hierarchical controls next to `'code' and 'support'` to expand and view their contents (Figure 4.9 on page 46).

Figure 4.9 CodeWarrior Project Window with Expanded Hierarchical Folders

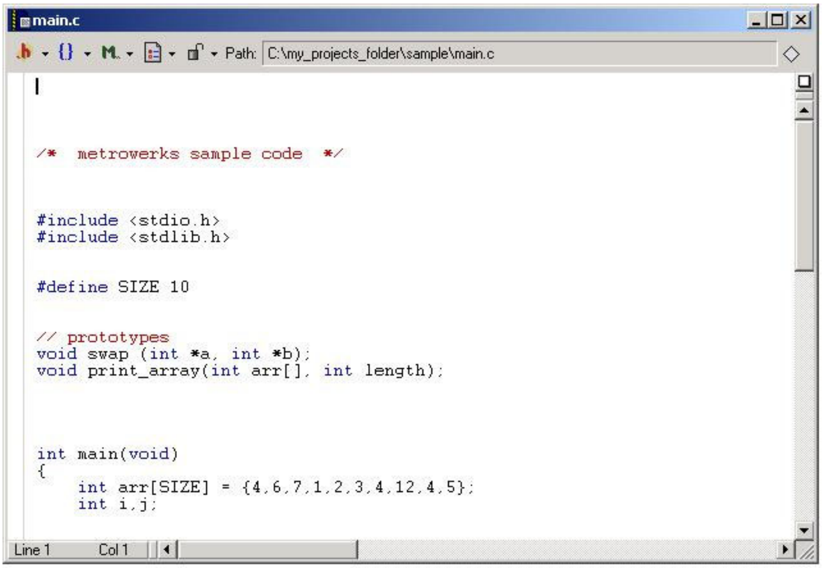


8. View a Source File

   a. Double-click the **main.c** file in the project window, the source code in the file is displayed in a CodeWarrior source-code editor window (Figure 4.10 on page 47).

NOTE For more details about the CodeWarrior editor and its features, see the *IDE User's Guide*.

Figure 4.10 CodeWarrior Editor Window



9. Examine the build target settings.

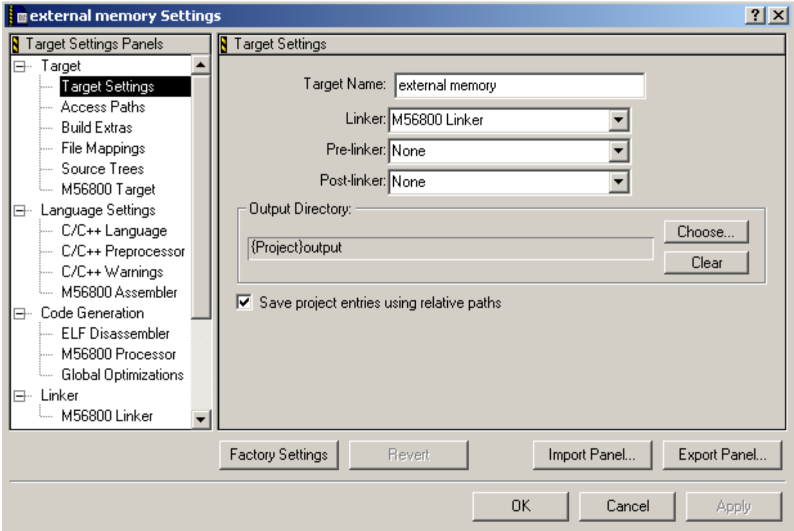
   The CodeWarrior IDE allows your project to have several different configurations contained within the project. These are called “build targets.” When you work with a new CodeWarrior project, you will want to examine your build target settings.

   a. To specify a build target, double-click the **Settings** icon in the Project window (see Figure 4.8 on page 45 for location of icons in the Project window).

      The **Target Settings** window {**external RAM (mode 3) Settings** in sample} appears (Figure 4.11 on page 48).

      This window contains several different panels. In Figure 4.11 on page 48, the **Target Settings *Panels*** is displayed in the **Target Settings** window.

Figure 4.11 Target Settings Window



b. If it is not already visible, click **Target** from the tree structure in the **Target Settings Panels pane to expand the hierarchical view**.

c. Click **Target Settings** from the hierarchical tree.

The **Target Settings** panel appears which displays all the options related to selecting a build target.

NOTE By selecting **M56800 Linker** from the **Linker** list box, the CodeWarrior IDE recognizes that the code you are writing is intended for DSP56800 processors, and populates the Target Settings Panel with the DSP56800 specific panels.

The **Target Settings** window is the location for all options related to the build target. Every panel and option is explained in the CodeWarrior documentation. Most of the general settings panels are explained in the *IDE User Guide*. DSP56800 target-specific panels are explained in this targeting manual.

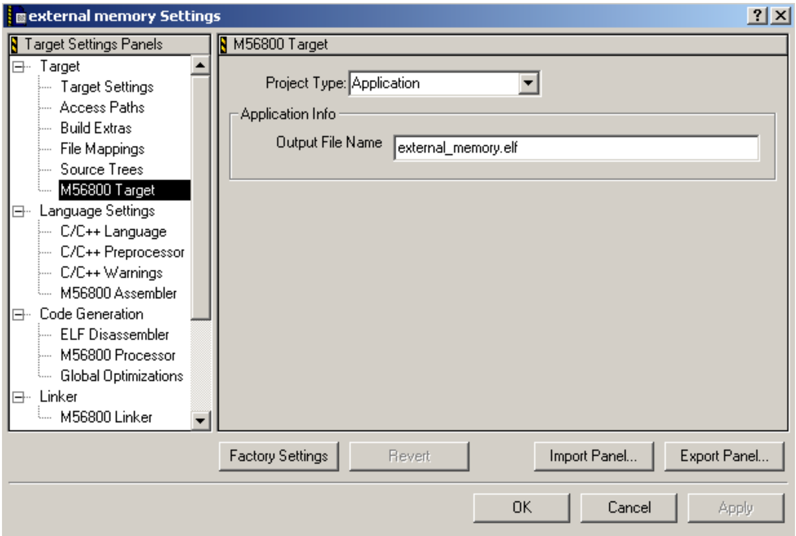
10. Set build target options:

a. In the **Target Settings Panels** panel, click **Target** in the tree structure to expand the hierarchical view.

b. Click **M56800 Target** from the hierarchical tree.

The **M56800 Target** panel appears (Figure 4.12 on page 49).

Figure 4.12 M56800 Target Settings Panel



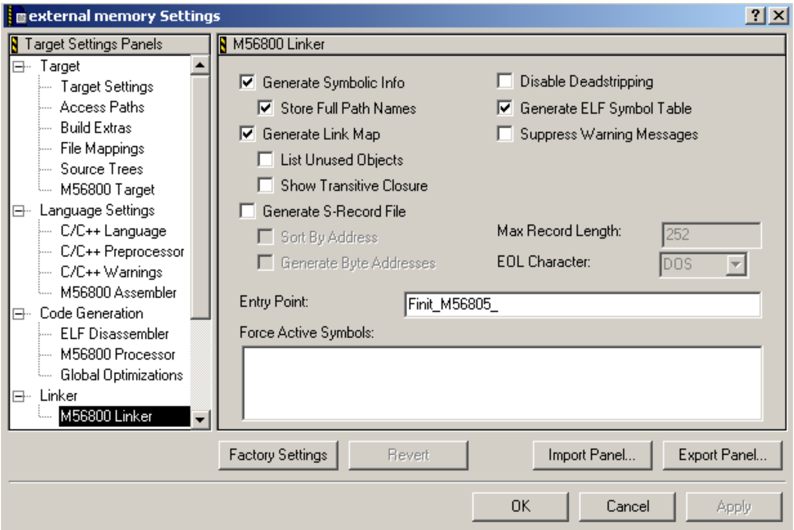
11. Set linker options.

   a. In the **Target Settings Panels** pane, click **Linker** in the tree structure to expand the hierarchical view.

   b. Click **M56800 Linker** from the hierarchical tree.

   The **M56800 Linker** panel appears (Figure 4.13 on page 50).

Figure 4.13 M56800 Linker Settings



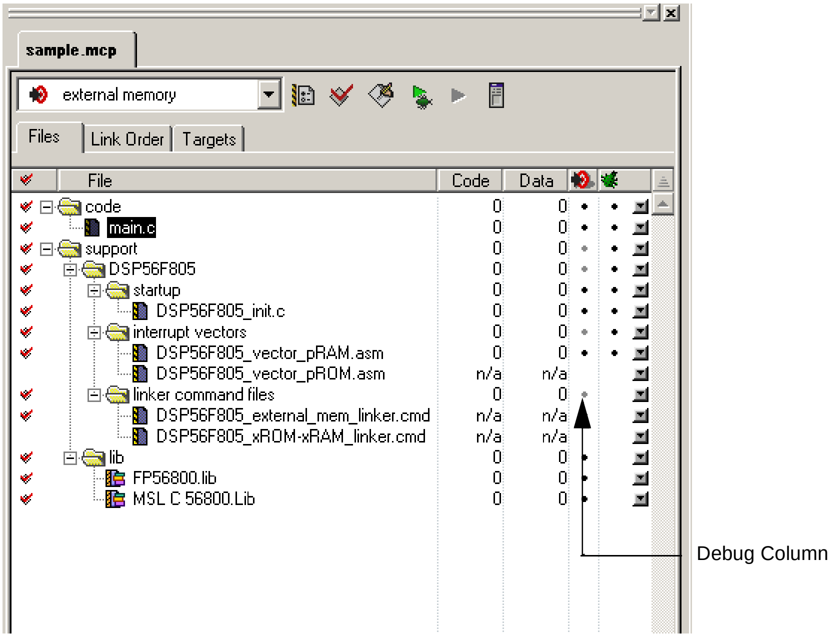
12. Examine the default settings and select the options according to your requirements. Close the **Target** window when you are finished by clicking the **OK** button.

13. Verify debugging information is generated.

    For the debugger to work, it needs certain information from the CodeWarrior IDE so that it can connect object code to source code. You must instruct the CodeWarrior IDE to produce this information.

    There is a debug-related column in the project window (Figure 4.14 on page 51). Every file, for which the IDE generates debugging information, has a dot in the Debug column. To enable symbolic information for a file, click the Debug column next to the file. A dot appears confirming that debugging information is generated for that file.

Figure 4.14 Turning on Debugging Per File



14. Compile the code using either of the following options:

- From the menu bar of the Freescale CodeWarrior window, select **Project > Make**.
- In the project window, click the **Make** icon.

The above step updates all files that need to be compiled and re-linked in the project. The IDE tracks these dependencies automatically.

NOTE The **Make** command in the menu bar of the Freescale CodeWarrior window compiles selected files, not all changed files. The **Bring Up To Date** command in the menu bar compiles all changed files, but does not link the project into an executable.

When you select the `Make` command, the IDE compiles all of the code. This may take some time as the IDE locates the files, opens them, and generates the object code. When the compiler completes the task, the linker creates an executable from the objects. You can see the compiler's progress in the project window and in the toolbar.

Editing the Contents of a Project

To change the contents of a project:

1. Add source files to the project.

   Most stationery projects contain source files that act as placeholders. Replace these placeholders with your own files.

   To add files, use one of the following options:

   - From the menu bar of the Freescale CodeWarrior window, select **Project > Add Files**.
   - Drag files from the desktop or Windows Explorer to the project window.

   To remove files:

   a. Select the files in the project window that you want to delete.

   b. Press the **Backspace** or **Delete** key.

2. Edit code in the source files.

   Use the IDE's source-code editor to modify the content of a source-code file. To open a file for editing, use either of the following options:

   - Double-click the file in the project window.
   - Select the file in the project window and press **Enter**.

   Once the file is open, you can use all of the editor's features to work with your code.

You have now been introduced to the major components of CodeWarrior Development Studio for Freescale 56800, except for the debugger. You are now familiar with the project manager, source code editor, and settings panels.

Working with the Debugger

In this section, you will explore the CodeWarrior debugger.

This tutorial assumes that you have already started the CodeWarrior IDE and have opened a sample project.

NOTE CodeWarrior IDE automatically enables the debugger and sets debugger-related settings within the project.

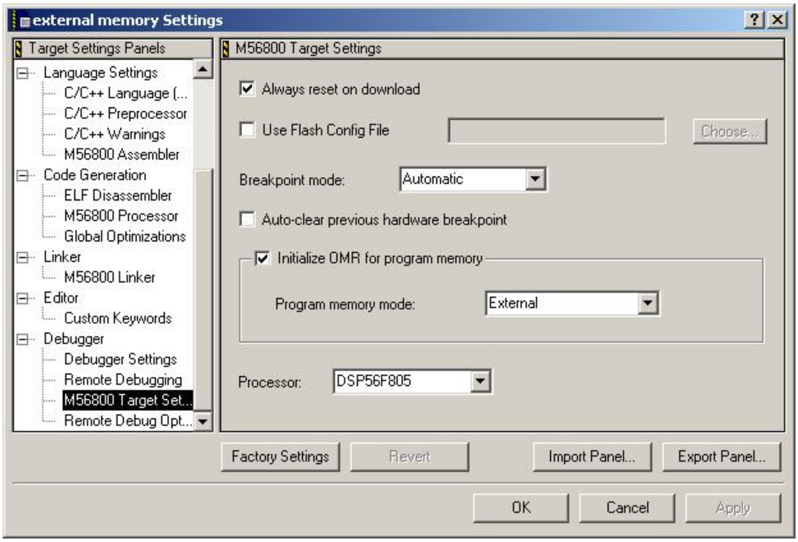
1. Access the **Target Settings** window **(**Figure 4.12 on page 49).
2. Set debugger options.

   a. In the **Target Settings Panels** pane, click **Debugger** in the tree structure to expand the hierarchical view.

b. Click **M56800 Target Settings** from the hierarchical tree

The **M56800 Target Settings** panel appears **(**Figure 4.15 on page 53).

Figure 4.15 Selecting Debugger Settings



3. Set protocol specific options:
   - Always reset on download

     Select this option to reset the board every time you download code to the board. If unchecked, the board is reset only before the initial download.

NOTE This option is not displayed if you select **Simulator** from the **Protocol** menu.

   - Breakpoint Mode

     From the pull-down menu, select **Software**.

   - Initialize OMR for Program Memory

     Enable **OMR For Program Memory** checkbox and select **External** memory.

4. Debug the project by using either of the following options:
   - From the Freescale CodeWarrior window, select **Project > Debug**.
   - Click the **Debug** button in the project window.

     This command instructs the IDE to compile and link the project. An ELF file is created in the process. ELF is the file format created by the CodeWarrior linker for

DSP56800. The ELF file contains the information required by the debugger and prepared by the IDE. When you debug the project on DSP hardware, the debugger displays the following message:

Resetting hardware. Please wait.

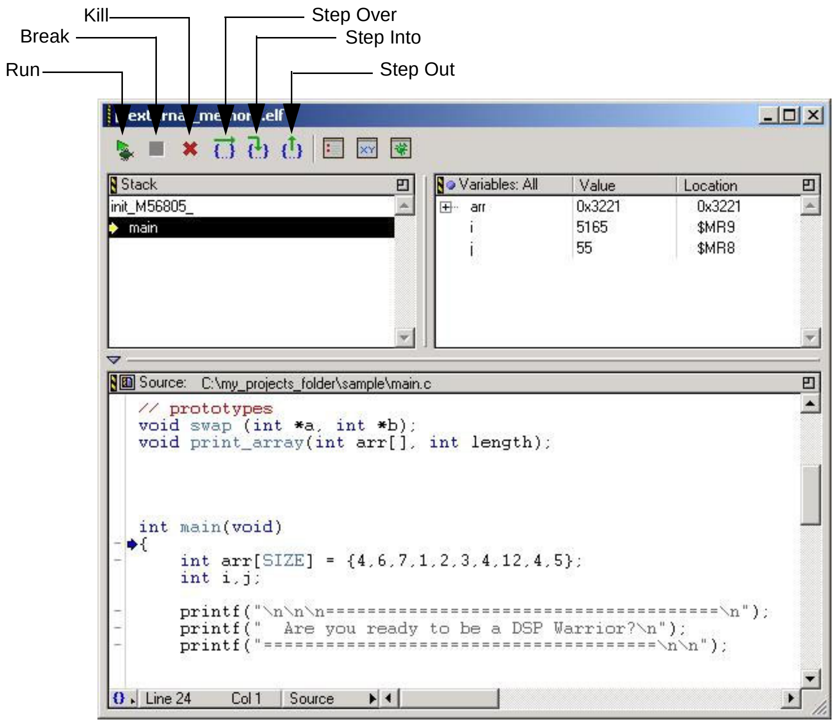
This reset step occurs automatically only once per debugging session. To reset the boards manually, press the **Reset** button on your board. Next, the debugger displays this message:

Download `external_memory.elf`

When the download to the board is complete, the IDE displays the **Program** window (**external_memory .elf** in sample) shown in Figure 4.16 on page 55.

NOTE Source code is shown only for files that are in the project folder or that have been added to the project in the project manager, and for which the IDE has created debug information. You must navigate the file system in order to locate sources that are outside the project folder and not in the project manager, such as library source files.

Figure 4.16 Program Window



5. Navigate through your code.

   The **Program** window has three panes:

   - **Stack pane**

     The **Stack** pane shows the function calling stack.

   - **Variables pane**

     The **Variables** pane displays local variables.

   - **Source pane**

     The **Source** panel displays source or assembly code.

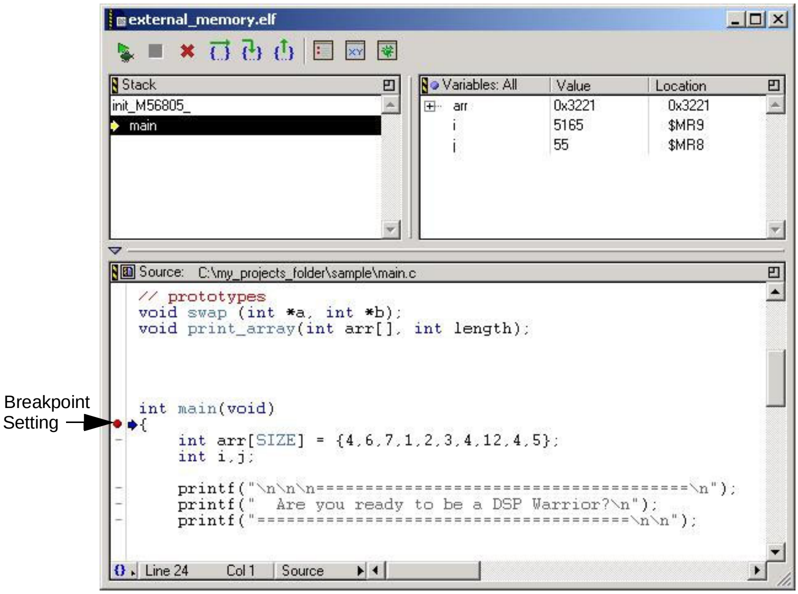
     The toolbar at the top of the **Program** window has buttons that allows you access to the execution commands in the **Debug** menu.

6. Set breakpoints.

a. Scroll through the code in the **Source** pane of the **Program** window until you come across the `main()` function.

b. Click the gray dash in the far left-hand column of the window, next to the first line of code in the `main()` function. A red dot appears (Figure 4.17 on page 56), confirming you have set your breakpoint.

Figure 4.17 Breakpoint in the Program Window



NOTE	To remove the breakpoint, click the red dot. The red dot disappears.

7. View and edit register values.

8. Registers are platform-specific. Different chip architectures have different registers.

   a. **From the menu bar of the Freescale CodeWarrior window,** select **View > Registers**.

   Expand the **General Purpose Registers** tree control to view the registers as in Figure 4.18 on page 57, or double-click on **General Purpose Registers** to view the registers as in Figure 4.19 on page 58.

Figure 4.18 General Purpose Registers for DSP56800

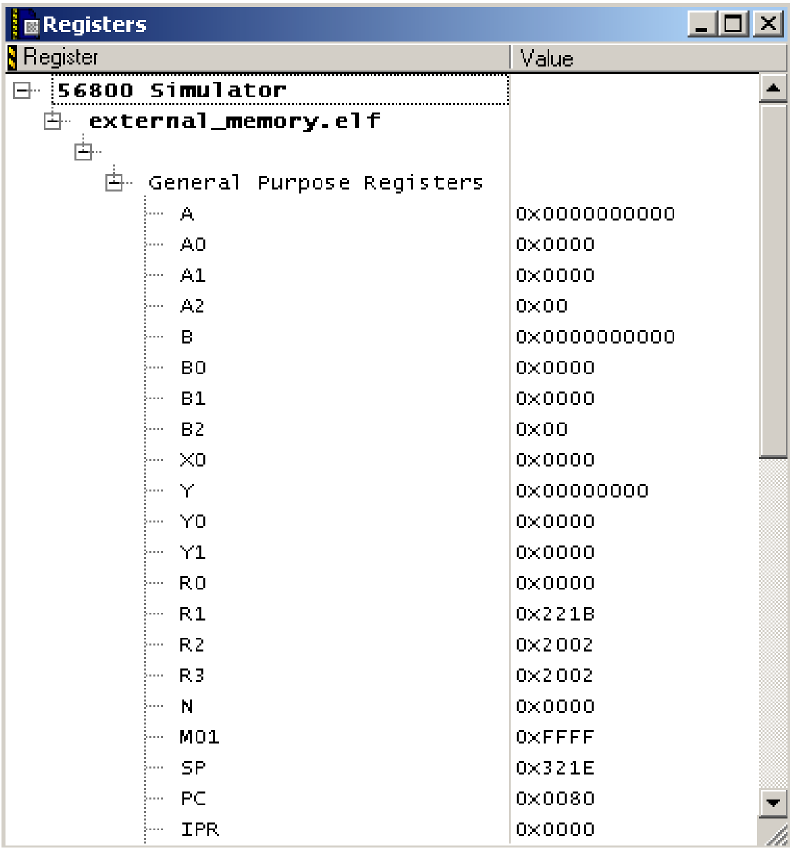
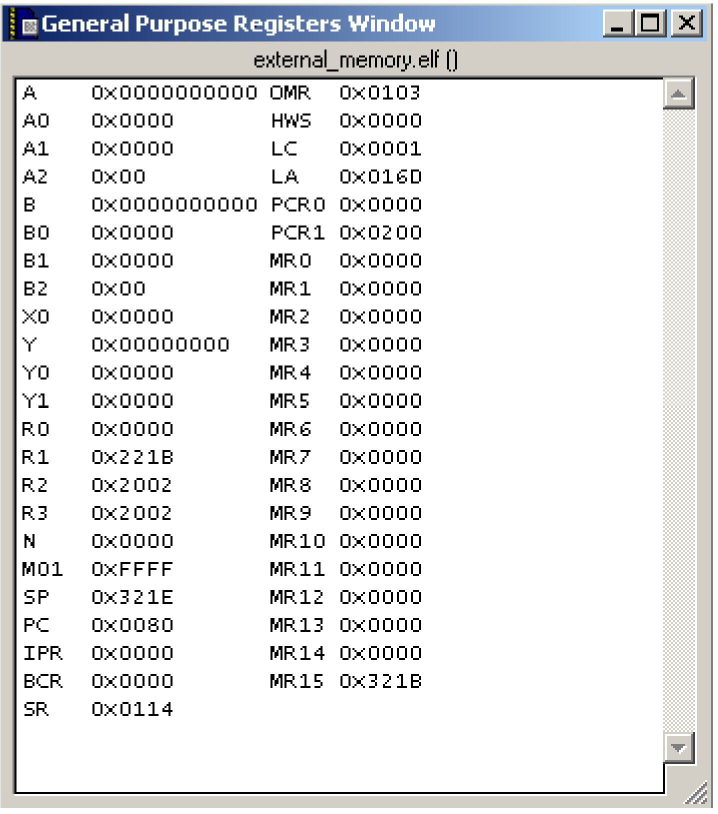


Figure 4.19 General Purpose Registers Window



b. To edit values in the register window, double-click a register value. Change the value as you wish.

9. View Data X:Memory.

All variables reside at a specific memory address determined at runtime.

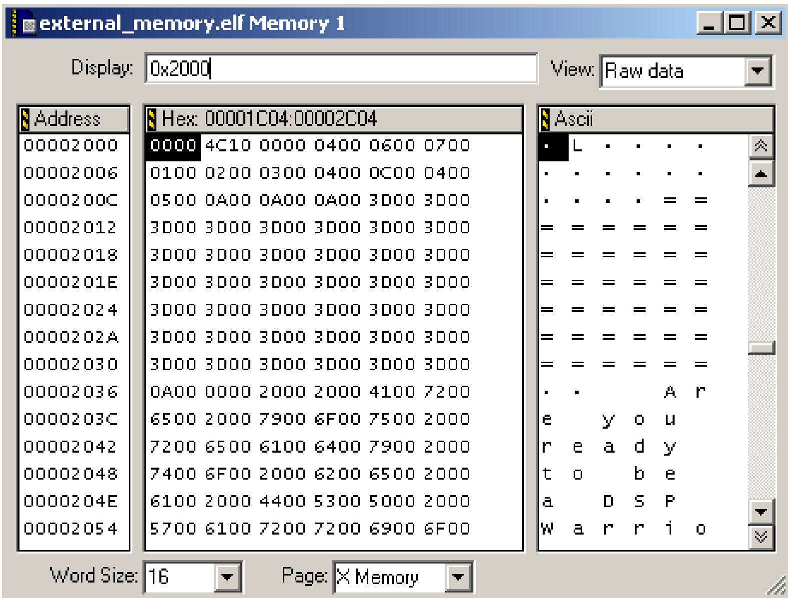
a. To view the memory address range of a variable, select **Data > View Memory** from the menu bar of the Freescale CodeWarrior window.

TIP If **Data > View Memory** is greyed out, make sure that you have the **Program** window as the highlighted window and that you either have the cursor in the **Source** pane or have a function selected in the **Stack** pane.

The **Memory** window appears (Figure 4.20 on page 59).

b. Locate the **Page** list box at the bottom of the **View Memory** window. Select **X Memory** from the **Page** list box.

Figure 4.20 View X:Memory Window



10. Enter the memory address in the **Display** field.

Enter a hexadecimal address in standard C hex notation, for example, 0x100.

The window displays the contents of X: memory.

If you are using the EVM hardware, type the address, `0x2000` in the **Display** text field and press **Enter**. You see the memory starting at that location. This is the beginning of the `.data` section. The memory address location for `.data` section (or any other section) are set through a combination of the Memory Segment on page 263 and Sections Segment on page 265 of the linker command file. Note that you see both the hexadecimal and ASCII values for X: memory. The contents of this window are editable as well.

11. View Program P:Memory.

a. To view the memory address range of a variable, select **Data > View Memory** from the menu bar of the Freescale CodeWarrior window.

TIP If **Data > View Memory** is greyed out, make sure that you have the **Program** window as the highlighted window and that you either have the cursor in the **Source** pane or have a function selected in the **Stack** pane.

The **Memory** window appears (Figure 4.21 on page 60).

b. Locate the **Page** list box at the bottom of the **View Memory**. Select **P Memory** from the **Page** list box.

c. Using the **View** list box, you have the option to view four types of P:Memory:

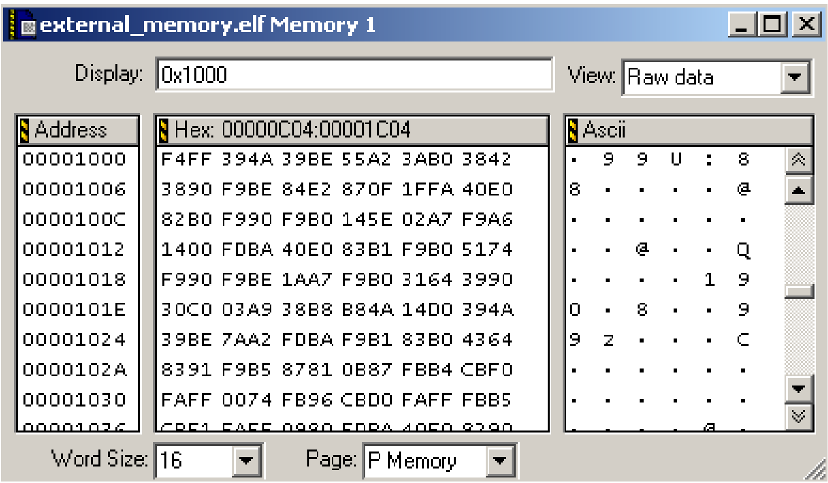
- Raw Data
- Disassembly
- Source
- Mixed

d. Enter the memory address in the **Display** field.

Enter a hexadecimal address in standard C hex notation, for example, 0x1000.

Figure 4.21 on page 60 shows Raw Data.

Figure 4.21 View P:Memory Window



12. Run the debugger.

Use either of the following options:

a. Select **Project > Run**.

b. Click the **Run** icon in the toolbar of the **Program** window.

In this simple example, the debugger will halt at a debug instruction after printing out messages to the console window. This debug instruction is the portion of the startup code which handles the program's exit.

13. Quit the application.

From the menu bar of the Freescale CodeWarrior window, select **Debug > Kill**. This stops the code execution and quits debugging.

Use either of the following options:

a. Select **Debug > Kill**

b. Click the **Kill** icon in the toolbar of the **Program** window.

This will stop code execution and close the **Program** window if the project is running. In this case, it will simply close the **Program** window, as we are currently halted.

References

You have completed the tutorial and used the basic elements of the CodeWarrior Development Studio for Freescale 56800.

Refer to the *IDE User Guide* to learn more about the features available to you.

5

Target Settings

Each build target in a CodeWarrior™ project has its own settings. This chapter explains the target settings panels for DSP56800 software development. The settings that you select affect the DSP56800 compiler, linker, assembler, and debugger.

This chapter contains the following sections:

Target Settings Overview

The target settings control:

- Compiler options
- Linker options
- Assembler options
- Debugger options
- Error and warning messages

When you create a project using stationery, the build targets, which are part of the stationery, already include default target settings. You can use those default target settings (if the settings are appropriate), or you can change them.

NOTE Use the DSP56800 project stationery when you create a new project.

Target Setting Panels

Table 5.1 on page 64 lists the target settings panels:

- Links identify the panels specific to DSP56800 projects. Click the link to go to the explanation of that panel.
- The Use column explains the purpose of generic IDE panels that also can apply to DSP56800 projects. For explanations of these panels, see the *IDE User Guide*.

Table 5.1 Target Setting Panels

Group	Panel Name	Use
Target	Target Settings on page 68	
	Access Paths	Selects the paths that the IDE searches to find files of your project. Types include absolute and project-relative.
	Build Extras	Sets options for building a project, including using a third-party debugger.
	File Mappings	Associates a filename extension, such as .c, with a plug-in compiler.
	Source Trees	Defines project -specific source trees (root paths) for your project.
	M56800 Target on page 70	
Language Settings	C/C++ Language (C only) on page 72	
	C/C++ Preprocessor on page 75	
	C/C++ Warnings on page 77	
	M56800 Assembler on page 82	
Code Generation	ELF Disassembler on page 83	
	M56800 Processor on page 86	
	Global Optimization	Configures how the compiler optimizes code.
Linker	M56800 Linker on page 88	

Table 5.1 Target Setting Panels (*continued*)

Group	Panel Name	Use
Editor	Custom Keywords	Changes colors for different types of text.
Debugger	Debugger Settings	Specifies settings for the CodeWarrior debugger.
	Remote Debugging on page 92	
	M56800 Target (Debugging) on page 94	
	Remote Debug Options on page 99	

Changing Target Settings

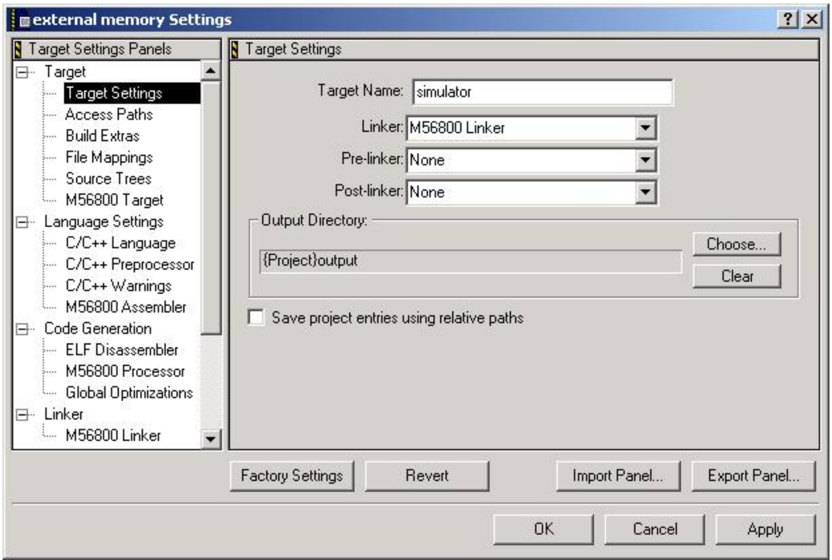
To change target settings:

1. Select **Edit > Target Name Settings.**

   Target is the name of the current build target in the CodeWarrior project.

   After you select this menu item, the CodeWarrior IDE displays the **Target Settings** window (Figure 5.1 on page 66).

Figure 5.1 Target Settings Window



The left side of the **Target Settings** window contains a list of target settings panels that apply to the current build target.

2. To view the Target Settings panel:

   Click on the name of the **Target Settings** panel in the **Target Settings** panels list on the left side of the **Target Settings** window.

   The CodeWarrior IDE displays the target settings panel that you selected.

3. Change the settings in the panel.
4. Click OK.

Exporting and Importing Panel Options to XML Files

The CodeWarrior IDE can export options for the current settings panel to an Extensible Markup Language (XML) file or import options for the current settings panel from a previously saved XML file.

Exporting Panel Options to XML File

1. Click the Export Panel button.
2. Assign a name to the XML file and save the file in the desired location.

Importing Panel Options from XML File

1. Click the Import Panel button.
2. Locate the XML file to where you saved the options for the current settings panel.
3. Open the file to import the options.

Saving New Target Settings in Stationery

To create stationery files with new target settings:

1. Create your new project from an existing stationery.
2. Change the target settings in your new project for any or all of the build targets in the project.
3. Save the new project in the Stationery folder.

Restoring Target Settings

After you change settings in an existing project, you can restore the previous settings by using any of the following methods:

- To restore the previous settings, click **Revert** at the bottom of the **Target Settings** window.
- To restore the settings to the factory defaults, click **Factory Settings** at the bottom of the window.

CodeWarrior IDE Target Settings Panels

Table 5.2 on page 68 lists and explains the CodeWarrior IDE target settings panels that can apply to DSP56800.

Table 5.2 Code Warrior IDE Target Settings Panels

Target Settings Panels	Description
Access Paths	Use this panel to select the paths that the CodeWarrior IDE searches to find files in your project. You can add several kinds of paths including absolute and project-relative. See *IDE User Guide*.
Build Extras	Use this panel to set options that affect the way the CodeWarrior IDE builds a project, including the use of a third-party debugger. See *IDE User Guide*.
File Mappings	Use this panel to associate a file name extension, such as.c, with a plug-in compiler. See *IDE User Guide*.
Source Trees	Use this panel to define project-specific source trees (root paths) for use in your projects. See *IDE User Guide*.
Custom Keywords	Use this panel to change the colors that the CodeWarrior IDE uses for different types of text. See *IDE User Guide*.
Global Optimizations	Use this panel to configure how the compiler optimizes the object code. See *IDE User Guide*.
Debugger Settings	Use this panel to specify settings for the CodeWarrior debugger.

DSP56800-Specific Target Settings Panels

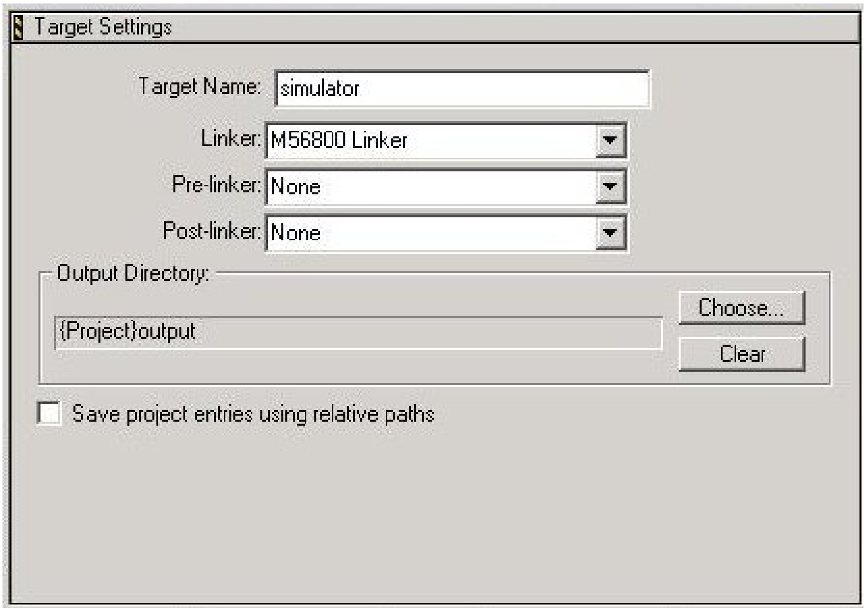
This section explains individual settings on DSP56800-specific target settings panels.

Target Settings

The **Target Settings** panel (Figure 5.2 on page 69), lets you set the name of your build target, as well as the linker and post-linker plug-ins to be used with the build target. By selecting a linker, you are specifying which family of processors to use. The other available panels in the **Target Settings** window change to reflect your choice.

Because the linker choice affects the visibility of other related panels, you must first set your build target before you can specify other options, like compiler and linker settings.

Figure 5.2 Target Settings Panel



Target Name

Use the **Target Name** field to set or change the name of a build target. When you use the **Targets** view in the project window, you see the name entered in the **Target Name** field.

The name you specify here is not the name of your final output file. It is instead a name for your personal use that you assign to the build target. You specify the name of the final output file in the **Output File Name** field of the **M56800 Target** panel.

Linker

Select a linker from the items listed in the **Linker** menu.

For DSP56800 projects, you must select the **M56800 Linker**. The selected linker defines the build targets. After you select a linker, only the panels appropriate for your build target (in this case, DSP56800) are available.

Pre-Linker

Some build targets have pre-linkers that perform additional work, such as data-format conversion, before the final executable file is built. CodeWarrior Development Studio for Freescale 56800 does not require a pre-linker, so set the **Pre-Linker** menu to **None**.

Post-Linker

Some build targets have post-linkers that perform additional work, such as data-format conversion, on the final executable file. CodeWarrior Development Studio for Freescale 56800 does not require a post-linker, so set the **Post-Linker** menu set to **None**.

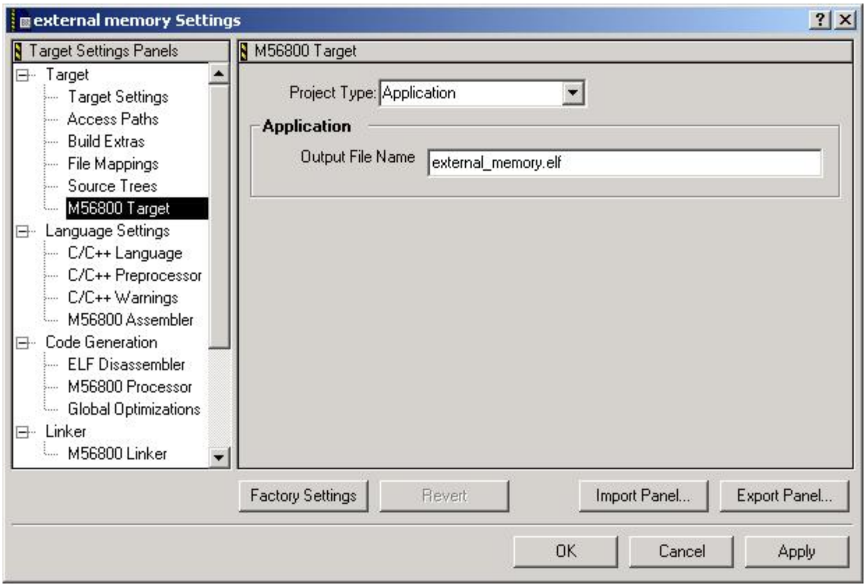
Output Directory

This field shows the directory to which the IDE saves the executable file that is built from the current project. The default output directory is the same directory in which the project file is located. If you want to save the executable file to a different directory, click the **Choose**. To erase the contents of this field, click **Clear**.

M56800 Target

The **M56800 Target** panel (Figure 5.3 on page 71) instructs the compiler and linker about the environment in which they are working, such as available memory and stack size. This panel is only available when the current build target uses the M56800 Linker.

Figure 5.3 M56800 Target Panel



The items in the **M56800 Target** panel are:

Project Type

The **Project Type** menu determines the kind of project you are creating. The available project types are **Application** and **Library**.

Use this menu to select the project type that reflects the kind of project you are building (Figure 5.3 on page 71).

Output File Name

The **Output File Name** field specifies the name of the executable file or library to create. This file is also used by the CodeWarrior debugger. By convention, application names must end with the extension "`.elf`" (without the quotes), and library names must end with the extension "`.lib`" (without the quotes).

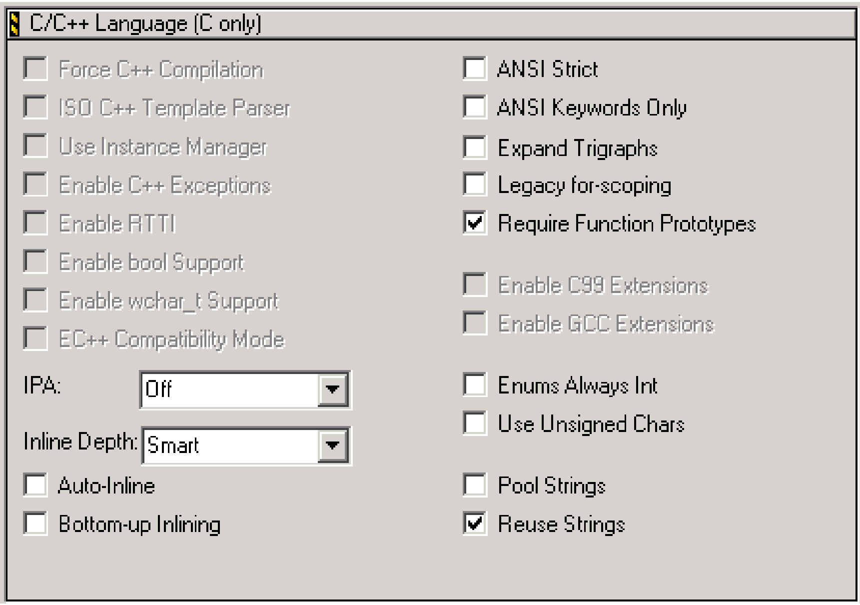
NOTE When building a library, ensure that you use the extension "`.lib`," as this is the default file-mapping entry for libraries.

If you wish to change an extension, you must add a file-mapping entry in the **File Mappings** settings panel.

C/C++ Language (C only)

Use the **C/C++ Language (C Only)** panel (Figure 5.4 on page 72) to specify C language features. Table 5.3 on page 73 explains the elements of this panel that apply to the processor, which supports only the C language.

Figure 5.4 C/C++ Language Panel (C only)



NOTE Always disable the following optionw, which do not apply to the DSP56800 compiler: Legacy for-scoping and Pool Strings

Table 5.3 C/C++ Language (C Only) Panel Elements

Element	Purpose	Comments
IPA list box	Specifies Interprocedural Analysis (IPA): Off — IPA is disabled File — inlining is deferred to the end of the file processing Program — Inlining is deferred until all files within the program are processed.	When the **Program** option is selected the **Disable Deadstripping** option on the linker preference panel must be enabled.
Inline Depth list box	Together with the ANSI Keyword Only checkbox, specifies whether to inline functions: Don't Inline — do not inline any Smart — inline small functions to a depth of 2 to 4 1 to 8 — Always inline functions to the number's depth Always inline — inline all functions, regardless of depth	If you call an inline function, the compiler inserts the function code, instead of issuing calling instructions. Inline functions execute faster, as there is no call. But overall code may be larger if function code is repeated in several places.
Auto-Inline checkbox	Checked — Compiler selects the functions to inline Clear — Compiler does not select functions for inlining	To check whether automatic inlining is in effect, use the __option(auto_inline) command.
Bottom-up Inlining checkbox	Checked — For a chain of function calls, the compiler begins inlining with the last function. Clear — Compiler does not do bottom-up inlining.	To check whether bottom-up inlining is in effect, use the __option(inline_bottom_up) command.

Table 5.3 C/C++ Language (C Only) Panel Elements (*continued*)

Element	Purpose	Comments
ANSI Strict checkbox	Checked — Disables CodeWarrior compiler extensions to C Clear — Permits CodeWarrior compiler extensions to C	Extensions are C++-style comments, unnamed arguments in function definitions, # not and argument in macros, identifier after #endif, typecasted pointers as lvalues, converting pointers to same-size types, arrays of zero length in structures, and the D constant suffix. To check whether ANSI strictness is in effect, use the __option(ANSI_strict) command.
ANSI Keywords Only checkbox	Checked — Does not permit additional keywords of CodeWarrior C. Clear — Does permit additional keywords.	Additional keywords are asm (use the compiler built-in assembler) and inline (lets you declare a C function to be inline). To check whether this keyword restriction is in effect, use the __option(only_std_keywords) command.
Expand Trigraphs checkbox	Checked — C Compiler ignores trigraph characters. Clear — C Compiler does not allow trigraph characters, per strict ANSI/ ISO standards.	Many common character constants resemble trigraph sequences, especially on the Mac OS. This extension lets you use these constants without including escape characters. NOTE: If this option is on, be careful about initializing strings or multi-character constants that include question marks. To check whether this option is on. use the __option(trigraphs) command.

Table 5.3 C/C++ Language (C Only) Panel Elements (*continued*)

Element	Purpose	Comments
Require Function Prototypes checkbox	Checked — Compiler does not allow functions that do not have prototypes. Clear — Compiler allows functions without prototypes.	This option helps prevent errors from calling a function before its declaration or definition. To check whether this option is in effect, use the __option(require_prototypes) command.
Enums Always Int checkbox	Checked — Restricts all enumerators to the size of a singed int. Clear — Compiler converts unsigned int enumerators to signed int, then chooses an accommodating data type, char to long int.	To check whether this restriction is in effect, use the __option(enumalwasysint) command.
Use Unsigned Chars checkbox	Checked — Compiler treats a char declaration as an unsigned char declaration. Clear — Compiler treats char and unsigned char declarations differently.	Some libraries were compiled without this option. Selecting this option may make your code incompatible with such libraries. To check whether this option is in effect, use the __option(unsigned_char) command.
Reuse Strings checkbox	Checked — Compiler stores only one copy of identical string literals, saving memory space. Clear — Compiler stores each string literal.	If you select this option, changing one of the strings affects them all.

C/C++ Preprocessor

The C/C++ Preprocessor (Figure 5.5 on page 76) panel controls how the preprocessor interprets source code. By modifying the settings on this panel, you can control how the preprocessor translates source code into preprocessed code.

More specifically, the C/C++ Preprocessor panel provides an editable text field that can be used to #define macros, set #pragmas, or #include prefix files.

Figure 5.5 The C/C++ Preprocessor Panel

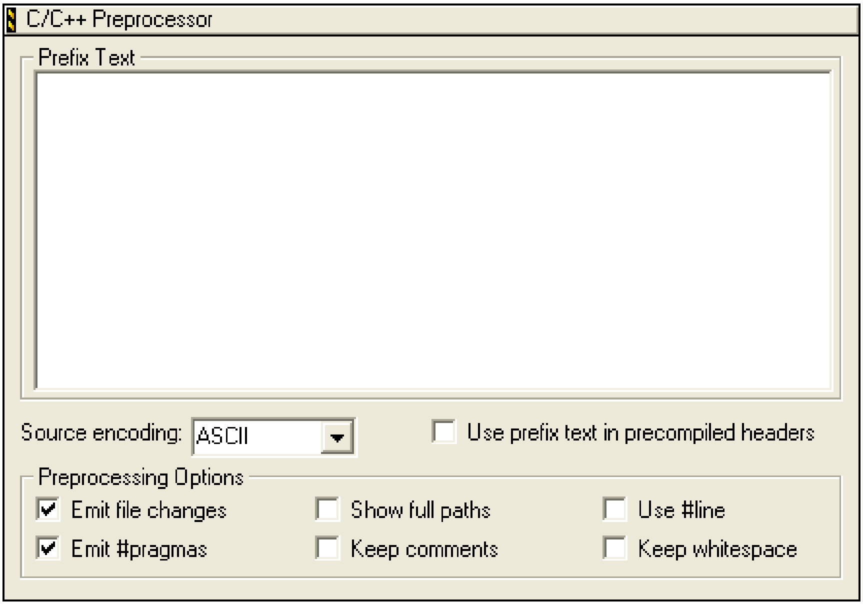


Table 5.4 on page 76 provides information about the options in this panel.

Table 5.4 C/C++ Language Preprocessor Elements

Element	Purpose	Comments
Source encoding	Allows you to specify the default encoding of source files. Multibyte and Unicode source text is supported.	To replicate the obsolete option “Multi-Byte Aware”, set this option to System or Autodetect. Additionally, options that affect the "preprocessing" request appear in this panel.
Use prefix text in precompiled header	Controls whether a *.pch or *.pch++ file incorporates the prefix text into itself.	This option defaults to “off” to correspond with previous versions of the compiler that ignore the prefix file when building precompiled headers. If any #pragmas are imported from old C/C++ Language (C Only) Panel settings, this option is set to “on”.

Table 5.4 C/C++ Language Preprocessor Elements (*continued*)

Element	Purpose	Comments
Emit file changes	Controls whether notification of file changes (or #line changes) appear in the output.	
Emit #pragmas	Controls whether #pragmas encountered in the source text appear in the preprocessor output.	This option is essential for producing reproducible test cases for bug reports.
Show full paths	Controls whether file changes show the full path or the base filename of the file.	
Keep comments	Controls whether comments are emitted in the output.	
Use #line	Controls whether file changes appear in comments (as before) or in #line directives.	
Keep whitespace	Controls whether whitespace is stripped out or copied into the output.	This is useful for keeping the starting column aligned with the original source, though we attempt to preserve space within the line. This doesn't apply when macros are expanded.

C/C++ Warnings

Use the C/C++ Warnings panel (Figure 5.6 on page 78) to specify C language features for the DSP56800. Table 5.5 on page 79 explains the elements of this panel.

NOTE The CodeWarrior compiler for DSP56800 does not support C++.

Figure 5.6 C/C++ Warnings Panel

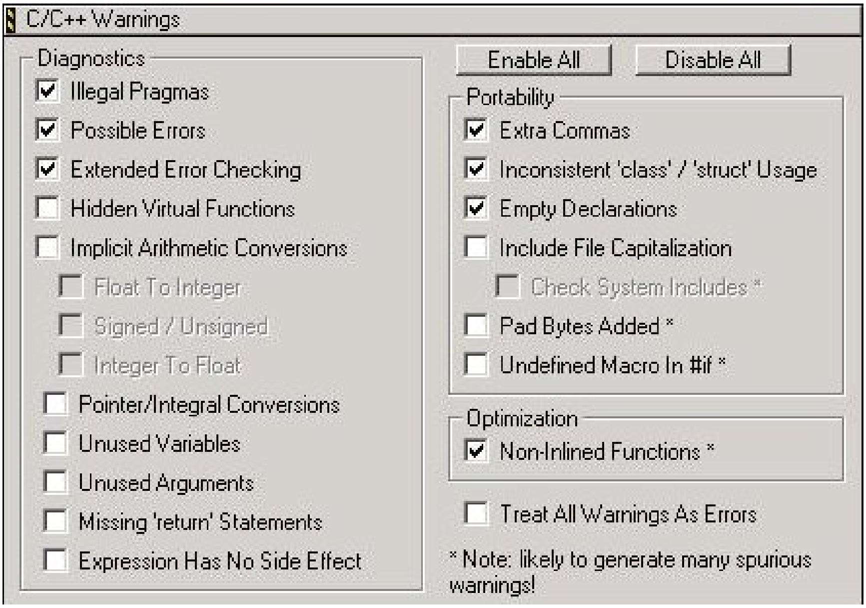


Table 5.5 C/C++ Warnings Panel Elements

Element	Purpose	Comments
Illegal Pragmas checkbox	Checked — Compiler issues warnings about invalid pragma statements. Clear — Compiler does not issue such warnings.	According to this option, the invalid statement **#pragma near_data off** would prompt the compiler response **WARNING: near data is not a pragma**. To check whether this option is in effect, use the __option(warn_illpragma) command.
Possible Errors checkbox	Checked — Compiler checks for common typing mistakes, such as == for =. Clear — Compiler does not perform such checks.	If this option is in effect, any of these conditions triggers a warning: an assignment in a logical expression; an assignment in a while, if, or for expression; an equal comparison in a statement that contains a single expression; a semicolon immediately after a while, if, or for statement. To check whether this option is in effect, use the __option(warn_possunwant) command.
Extended Error Checking checkbox	Checked — Compiler issues warnings in response to specific syntax problems. Clear — Compiler does not perform such checks.	Syntax problems are: a non-void function without a return statement, an integer or floating-point value assigned to an enum type, or an empty return statement in a function not declared void. To check whether this option is in effect, use the __option(extended_errorcheck) command.
Hidden Virtual Functions	Leave clear.	Does not apply to C.

Table 5.5 C/C++ Warnings Panel Elements (*continued*)

Element	Purpose	Comments
Implicit Arithmetic Conversions checkbox	Checked — Compiler verifies that operation destinations are large enough to hold all possible results. Clear — Compiler does not perform such checks.	If this option is in effect, the compiler would issue a warning in response to assigning a long value to a char variable. To check whether this option is in effect, use the __option(warn_implicitconv) command.
Pointer/ Integral Conversions	Checked — Compiler checks for pointer/integral conversions. Clear — Compiler does not perform such checks.	See #pragma warn_any_ptr_int_conv and #pragma warn_ptr_int_conv.
Unused Variables checkbox	Checked — Compiler checks for declared, but unused, variables. Clear — Compiler does not perform such checks.	The pragma **unused** overrides this option. To check whether this option is in effect, use the __option(warn_unusedvar) command.
Unused Arguments checkbox	Checked — Compiler checks for declared, but unused, arguments. Clear — Compiler does not perform such checks.	The pragma unused overrides this option. Another way to override this option is clearing the ANSI Strict checkbox of the C/C++ Language (C Only) panel, then not assigning a name to the unused argument. To check whether this option is in effect, use the __option(warn_unusedarg) command.
Missing ‘return’ Statements	Checked — Compiler checks for missing ‘return’ statements. Clear — Compiler does not perform such checks.	See #pragma warn_missingreturn.

Table 5.5 C/C++ Warnings Panel Elements (*continued*)

Element	Purpose	Comments
Expression Has No Side Effect	Checked — Compiler issues warning if expression has no side effect. Clear — Compiler does not perform such checks.	See #pragma warn_no_side_effect.
Extra Commas checkbox	Checked — Compiler checks for extra commas in enums. Clear — Compiler does not perform such checks.	To check whether this option is in effect, use the __option(warn_extracomma) command.
Inconsistent Use of 'class' and 'struct' Keywords checkbox	Leave clear.	Does not apply to C.
Empty Declarations checkbox	Checked — Compiler issues warnings about declarations without variable names. Clear — Compiler does not issue such warnings.	According to this option, the incomplete declaration **int ;** would prompt the compiler response **WARNING**. To check whether this option is in effect, use the __option(warn_emptydecl) command.
Include File Capitializatio n	Checked — Compiler issues warning about include file capitialization. Clear — Compiler does not perform such checks.	See #pragma warn_filenamecaps.
Pad Bytes Added	Checked — Compiler checks for pad bytes added. Clear — Compiler does not perform such checks.	See #pragma warn_padding.
Undefined Macro In #if	Checked — Compiler checks for undefined macro in #if. Clear — Compiler does not perform such checks.	See #pragma warn_undefmacro.

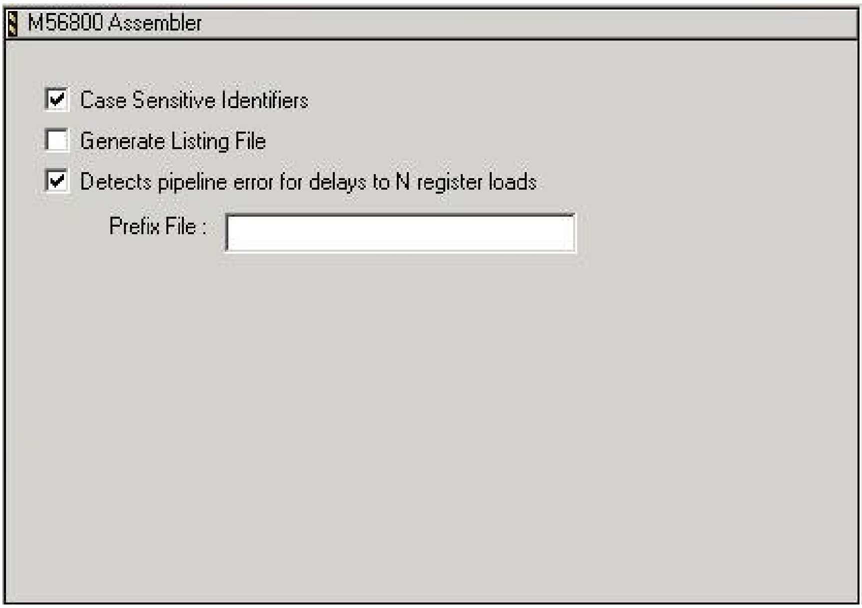
Table 5.5 C/C++ Warnings Panel Elements (*continued*)

Element	Purpose	Comments
Non-Inlined Functions checkbox	Checked — Compiler issues a warning if unable to inline a function. Clear — Compiler does not issue such warnings.	To check whether this option is in effect, use the __option(warn_notinlined) command.
Treat All Warnings As Errors checkbox	Checked — System displays warnings as error messages. Clear — System keeps warnings and error messages distinct.	

M56800 Assembler

The **M56800 Assembler** panel (Figure 5.7 on page 82) determines the format used for the assembly source files and the code generated by the DSP56800 assembler.

Figure 5.7 M56800 Assembler Settings Panel



The items in this panel are:

Case Sensitive Identifiers

When this option is enabled, the assembler distinguishes lowercase characters from uppercase characters for symbols. For example, the identifier `flag` is the not the same as `Flag` when the option is enabled.

NOTE This option must be enabled when mixing assembler and C code.

Generate Listing File

The **Generate Listing File** option determines whether or not a listing file is generated when the CodeWarrior IDE assembles the source files in the project. The assembler creates a listing file that contains file source along with line numbers, relocation information, and macro expansions when the option is enabled. When the option is disabled, the assembler does not generate the listing file.

When a listing file is output, the file is created in the same directory as the assembly file it is listing with a `.lst` extension appended to the end of the file name.

Detects pipeline errors for delays to N register loads

Checking this option enables the assembler to generate error messages.

In the following instruction: `[move X:(Rn+offset),N]`, `N` is not available in the instruction following immediately. This option allows the assembler to flag error for pipeline dependencies.

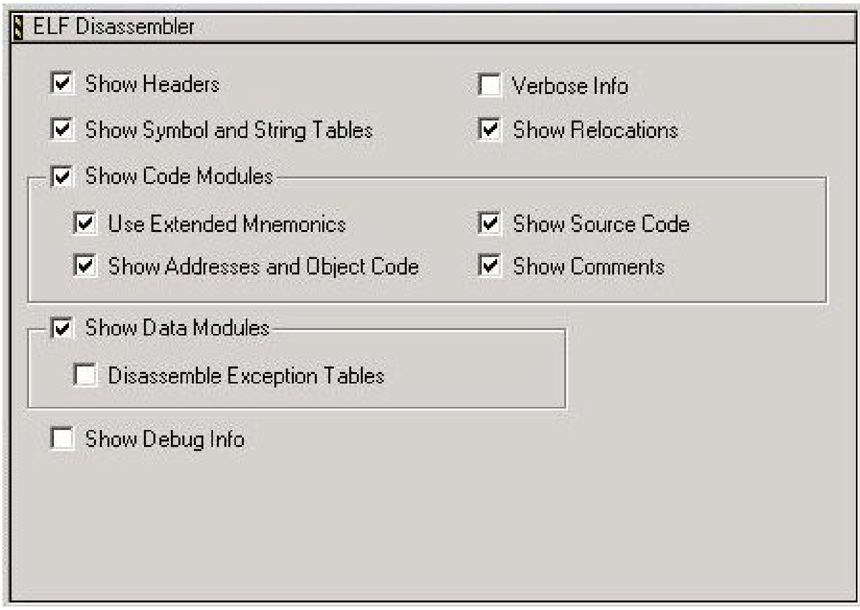
Prefix File

The **Prefix File** field contains the name of a file to be included automatically at the beginning of every assembly file in the project. This field lets you include common definitions without using an `include` directive in every file.

ELF Disassembler

The **ELF Disassembler** panel (Figure 5.8 on page 84) appears when you disassemble object files. To view the disassembly of a module, select **Project > Disassemble**.

Figure 5.8 ELF Disassembler Panel



The **ELF Disassembler** panel options are:

- **Show Headers**

  The **Show Headers** option determines whether the assembled file lists any ELF header information in the disassembled output.

- **Show Symbol and String Tables**

  The **Show Symbol and String Tables** option determines whether the disassembler lists the symbol and string table for the disassembled module.

- **Verbose Info**

  The **Verbose Info** option instructs the disassembler to show additional information in the ELF file. For the `.symtab` section, some of the descriptive constants are shown with their numeric equivalents. The sections `.line` and `.debug` are shown with an unstructured hex dump.

- Show Relocations

  The **Show Relocations** option shows relocation information for the corresponding text (`.rela.text`) or data (`.rela.data`) section.

- **Show Code Modules**

The **Show Code Modules** option determines whether the disassembler outputs the ELF code sections for the disassembled module.

If enabled, the **Use Extended Mnemonics, Show Source Code**, **Show Addresses and Object Code,** and **Show Comments** options become available.

– **Use Extended Mnemonics**

  The **Use Extended Mnemonics** option determines whether the disassembler lists the extended mnemonics for each instruction of the disassembled module.

  This option is available only if the **Show Code Modules** option is enabled.

– **Show Addresses and Object Code**

  The **Show Addresses** and **Object Code** option determines whether the disassembler lists the address and object code for the disassembled module.

  This option is available only if the **Show Code Modules** option is enabled.

– **Show Source Code**

  The **Show Source Code** option determines whether the disassembler lists the source code for the current module. Source code is displayed in mixed mode with line number information from the original C source.

  This option is available only if the **Show Code Modules** option is enabled.

– **Show Comments**

  The **Show Comments** option displays comments produced by the disassembler, in sections where comment columns are provided.

  This option is available only if the **Show Code Modules** option is enabled.

- **Show Data Modules**

  The **Show Data Modules** option determines whether or not the disassembler outputs any ELF data sections (such as `.data` and `.bss`) for the disassembled module.

  – **Disassemble Exception Tables**

    The **Disassemble Exception Tables** option determines whether or not the disassembler outputs any C++ exception tables for the disassembled module.

    This option is available when you select **Show Data Modules**.

NOTE **Disassemble Exception Tables** is not available for DSP56800, since it does not support C++.

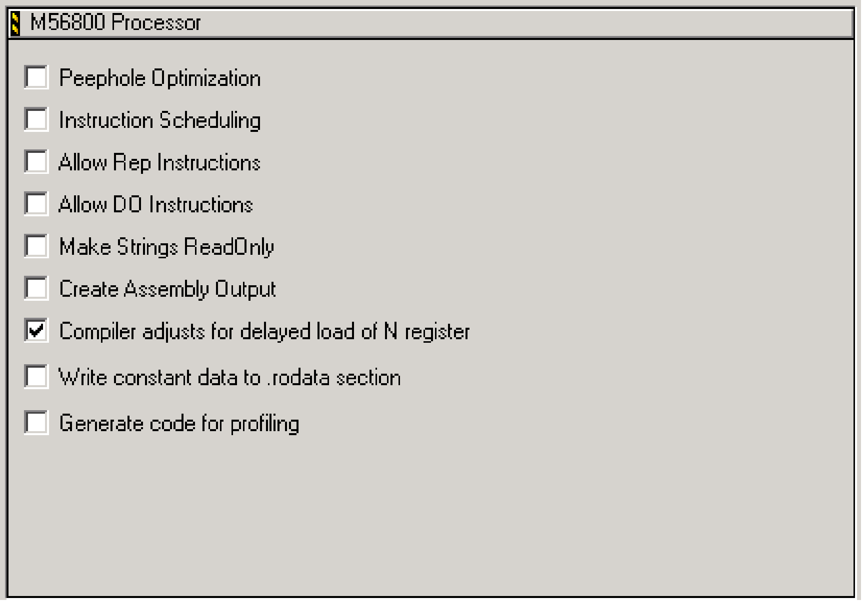
- **Show Debug Info**

The **Show Debug Info** option directs the disassembler to include DWARF symbol information in the disassembled output.

M56800 Processor

The **M56800 Processor** settings panel (Figure 5.9 on page 86) determines the kind of code the compiler creates. This panel is available only when the current build target uses the M56800 Linker.

Figure 5.9 M56800 Processor Settings Panel



The items in this panel are:

Peephole Optimization

This option controls the use of peephole optimizations. The peephole optimizations are small local optimizations that eliminate some compare instructions and optimize some address register updates for more efficient sequences.

Instruction Scheduling

This option determines whether the compiler rearranges instructions to take advantage of the M56800's scheduling architecture. This option results in faster execution speed, but is often difficult to debug.

NOTE Instruction Scheduling can make source-level debugging difficult because the source code might not correspond exactly to the underlying instructions. Disable this option when debugging code.

Allow REP Instructions

This option controls `REP` instruction usage. Such instructions are generally more efficient, but they prevent you from servicing any incoming interrupts inside a `REP` construct. If you are using interrupts or writing a time-critical real-time application, avoid using `REP` instructions.

Allow DO Instructions

This option controls the compiler's support for the `DO` instruction. Since the compiler never nests `DO` instructions, interrupt routines are always free to use those instructions.

Make Strings ReadOnly

This option determines whether you can specify a location to store string constants. If this option is disabled, the compiler stores string constants in the data section of the ELF file. If this option is enabled, the compiler stores string constants in the read-only `.rodata` section.

Create Assembly Output

This option allows the compiler to produce a `.asm` assembler-compatible file for each C source file in the project. The `.asm` file is located in the same path as the Project/Debug file and has the same name as the .c file containing `main`.

For example, `MyProgram.c` would produce the assembly output `MyProgram.asm`.

Compiler adjusts for delayed load of N-registers

When N-register (offset registers) are used consecutively, this option allows the compiler to send NOP instruction to resolve the restrictions in pipeline dependencies.

Write const data to .rodata section

This option allows the compiler to write all constant data to a read-only memory section (.rodata). You must add .rodata section in the linker command file. This option is overridden by the use_rodata pragma.

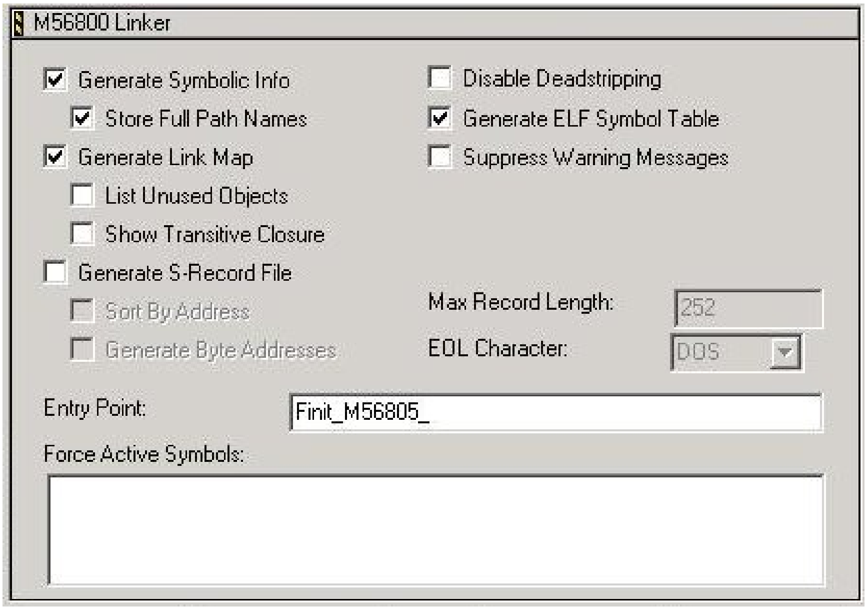
Generate code for profiling

This option allows the compiler to generate code for profiling. For more details about the profiler, see the “Profiler” on page 261.

M56800 Linker

The **M56800 Linker** panel (Figure 5.10 on page 88) controls the behavior of the linker. This panel is only available if the current build target uses the M56800 Linker.

Figure 5.10 M56800 Linker Settings Panel



The **M56800 Linker** panel options are:

- **Generate Symbolic Info**

  The **Generate Symbolic Info** option controls whether the linker generates debugging information. If the option is enabled, the linker generates debugging information. This information is included within the linked ELF file. This setting does not generate a separate file.

If you select **Project** > **Debug**, the CodeWarrior IDE enables the **Generate Symbolic Info** option for you.

If the **Generate Symbolic Info** option is not enabled, the **Store Full Path Names** option is not available.

NOTE If you decide to disable the **Generate Symbolic Info** option, you cannot debug your project using the CodeWarrior debugger. For this reason, the compiler enables this option by default.

– **Store Full Path Names**

The **Store Full Path Names** option controls how the linker includes path information for source files when generating debugging information.

If this option is enabled, the linker includes full path names to the source files. If this option is disabled, the linker uses only the file names. By default, this option is enabled.

This option is available only if you enable the Generate Symbolic Info on page 88 option.

- **Generate Link Map**

The **Generate Link Map** option controls whether the linker generates a link map. Enable this option to let the linker generate a link map.

The file name for the link map adds the extension `.xMAP` to the generated file name. The linker places the link map in the same folder as the output `.elf` file.

For each object and function in the output file, the link map shows which file provided the definition. The link map also shows the address given to each object and function, a memory map of where each section resides in memory, and the value of each linker-generated symbol.

Although the linker aggressively strips unused code and data when the CodeWarrior IDE compiles the relocatable file, it never deadstrips assembler relocatable files or relocatable files built with other compilers. If a relocatable file was not built with the CodeWarrior C compiler, the link map lists all of the unused but unstripped symbols.

– **List Unused Objects**

The **List Unused Objects** option controls whether the linker includes unused objects in the link map. Enable the option to let the linker include unused objects in the link map. The linker does not link unused code in the program.

Usually, this option is disabled. However, you might want to enable it in certain cases. For example, you might discover that an object you expect to be used is not actually used. This option is not available unless you enable the **Generate Link Map** option.

– **Show Transitive Closure**

The **Show Transitive Closure** option recursively lists in the link map file all of the objects referenced by main( ). Listing 5.1 on page 90 shows some sample code. To show the effect of the **Show Transitive Closure** option, you must compile the code.

Listing 5.1 Sample Code to Show Transitive Closure

```
void foot( void ){  int a = 100; }
void pad( void ){ int b = 101; }

int main( void ){
  foot();
  pad();
  return 1;
}
```

After you compile the source, the linker generates a link map file. Note that this option is not available unless you enable the **Generate Link Map** option.

Listing 5.2 Effects of Show Transitive Closure in the Link Map File

```
# Link map of Finit_sim_
  1] interrupt_vectors.text found in 56800_vector.asm
   2] sim_intRoutine (notype,local) found in 56800_vector.asm
   2] Finit_sim_ (func,global) found in 56800_init.asm
    3] Fmain (func,global) found in M56800_main.c
     4] Ffoot (func,global) found in M56800_main.c
     4] Fpad (func,global) found in M56800_main.c
    3] F__init_sections (func,global) found in Runtime 56800.lib
initsections.o
     4] Fmemset (func,global) found in MSL C 56800.lib mem.o
      5] F__fill_mem (func,global) found in MSL C 56800.lib
mem_funcs.o
  1] Finit_sim_ (func,global) found in 56800_init.asm
```

- **Disable Deadstripping**

  The **Disable Deadstripping** option prevents the linker from removing unused code and data.

- **Generate ELF Symbol Table**

The **Generate ELF Symbol Table** option instructs the linker to generate an ELF symbol table, as well as a list of relocations in the ELF executable file.

- **Suppress Warning Messages**

  The **Suppress Warning Messages** option controls whether the linker displays warnings. If this option is disabled, the linker displays warnings in the Message window. If this option is disabled, the linker does not display warnings.

- **Generate S-Record File**

  The **Generate S-Record File** option controls whether the linker generates an S-record file based on the application object image. The S-record files have the extension `.s`.

  In the case of the DSP56800, the linker generates three different S-record files. Their contents are:

  – `{output file name}.S`

    S-record file containing both P and X memory contents.

  – `{output file name}.p.S`

    S-record file containing P memory contents only.

  – `{output file name}.x.S`

    S-record file containing X memory contents only.

  The linker places the S-record files in the output folder, which is a sub-folder of the project folder.

  The linker generates the following S3 type S-records:

  – Sort by Address

    This option enables the compiler to sort S-records generated by the linker using byte address. This option is not available unless you enable the Generate S-Record File on page 91 option.

  – Generate Byte Addresses

    This option enables the linker to generate S-records in bytes. This option is not available unless you enable the Generate S-Record File on page 91 option.

  – Max Record Length

    The **Max Record Length** field specifies the maximum length of the S-record generated by the linker. This field is available only if the Generate S-Record File on page 91 option is enabled. The maximum value for an S-record length is 256 bytes.

NOTE Most programs that load applications onto embedded systems have a maximum length for S-records. The CodeWarrior debugger can handle S-records as large as 256 bytes. If you are using something other than the CodeWarrior debugger to load your embedded application, you need to determine its maximum length.

– **EOL Character**

  The **EOL Character** list box defines the end-of-line character for the S-record file. This list box is available only if you enable the Generate S-Record File on page 91 option.

- **Entry Point**

  The starting point for a program is set in the **Entry Point** field in the M56800 settings panel. The **Entry Point** field specifies the function that the linker first uses when the program runs.

  The default function found in this field is located within the startup code that sets up the DSP56800 environment before your code executes. This function and its corresponding startup code will be different depending upon which stationery you have selected. In the case of hardware targeted stationery, the startup code can be found in the stationery-generated project's startup folder.

  The `startup code` performs other tasks, such as clearing the hardware stack, creating an interrupt table, and getting the stack start and exception handler addresses.

  The final task performed by the startup code is to call your `main()` function.

- **Force Active Symbols**

  The **Force Active Symbols** field instructs the linker to include symbols in the link even if the symbols are not referenced. In essence, it is a way to make symbols immune to deadstripping. When listing multiple symbols, use a single space between them as a separator.

Remote Debugging

Use the Remote Debugging panel (Figure 5.11 on page 93, Figure 5.12 on page 93) to set parameters for communication between a DSP56800 board or Simulator and the CodeWarrior DSP56800 debugger. Table 5.6 on page 94 explains the elements of this panel.

NOTE Communications specifications also involve settings of the debugging M56800 Target panel (Figure 5.13 on page 95).

Figure 5.11 Remote Debugging Panel (56800 Simulator)

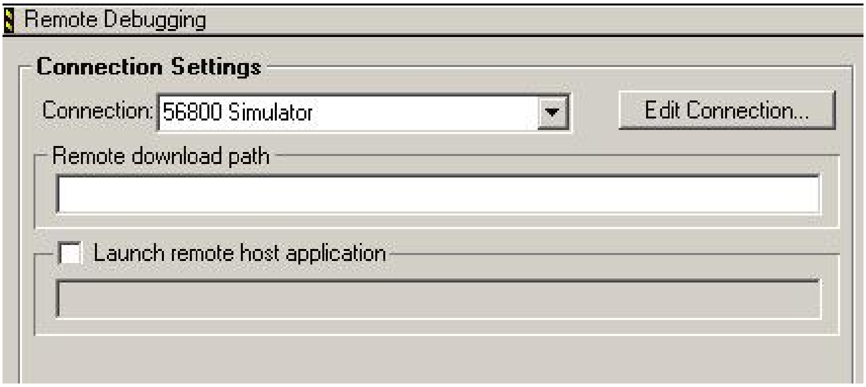


Figure 5.12 Remote Debugging Panel (Local Connection)

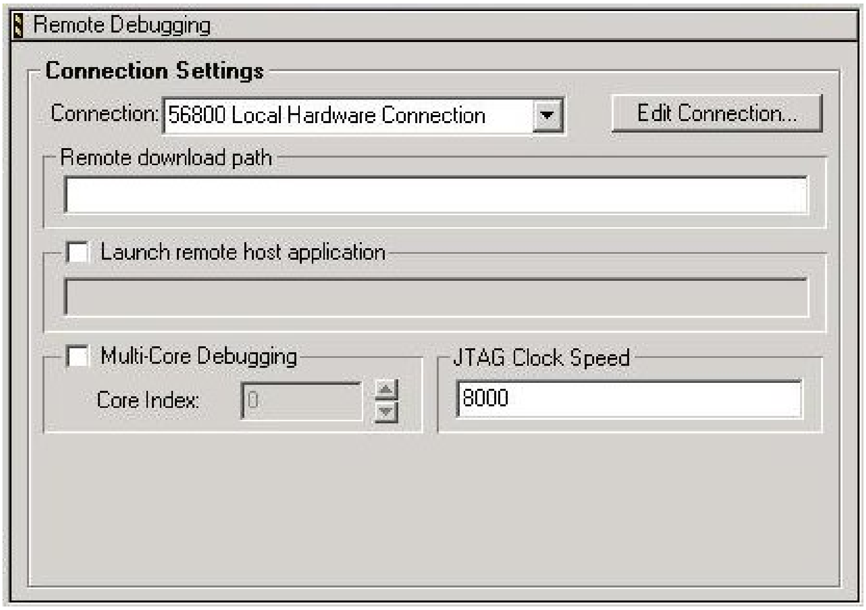


Table 5.6 Remote Debugging Panel Elements

Element	Purpose	Comments
Connection list box	Specifies the connection type: • 56800 Simulator — appropriate for testing code on the simulator before downloading code to an actual board. • 56800 Local Hardware Connection (CSS) — appropriate for using your computer's command converter server, connected to a DSP56800 board.	Selecting 56800 Simulator keeps the panel as Figure 5.11 on page 93 shows. Selecting 56800 Local Hardware Connection adds the JTAG Clock Speed text box to the panel, as Figure 5.12 on page 93 shows.
Remote Download Path text box		Not supported at this time.
Launch Remote Host Application checkbox		Not supported at this time.
Multi-Core Debugging	Allows debugging of multiple boards on a complex scan chain.	For more details, see Debugging on a Complex Scan Chain on page 245
JTAG Clock Speed text box	Specifies the JTAG lock speed for local hardware connection. (Default is 600 kilohertz.)	This list box is available only if the Connection list box specifies 56800 Local Hardware Connection (CSS). The HTI will not work properly with a clock speed over 800 kHz.

M56800 Target (Debugging)

The **M56800 Target Settings** panel lets you set communication protocols for interaction between the DSP56800 board and the CodeWarrior debugger.

Figure 5.13 M56800 Target Settings Panel

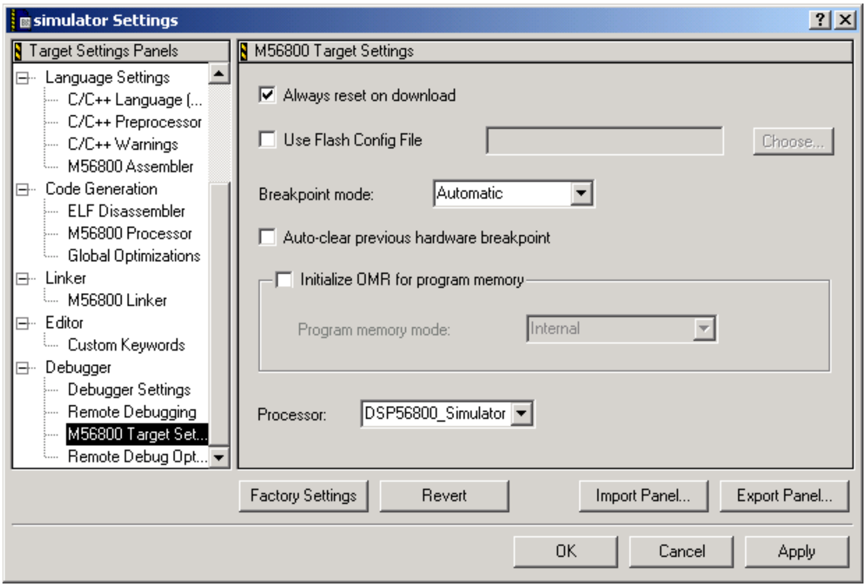


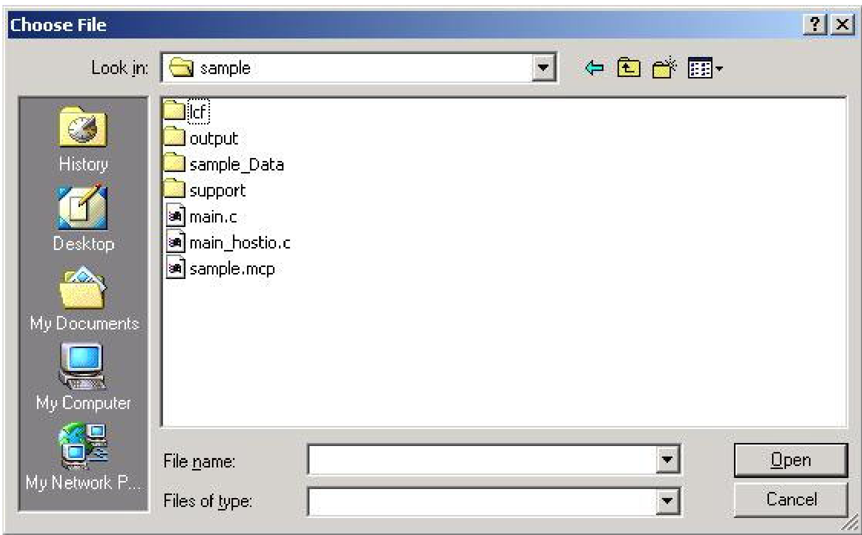
Table 5.7 Debugging M56800 Target Panel Elements

Element	Purpose	Comments
Always reset on download checkbox	Checked — IDE issues a reset to the target board each time you connect to the board. Clear — IDE does not issue a reset each time you connect to the target board.	
Use flash config file checkbox	Checked — When the **Use Flash Config File** option is enabled, you can specify the use of a flash configuration file (Listing 5.3 on page 98) in the text box . Clear — Debugger assumes no flash on the target.	If the full path and file name are not specified, the default location is the same as the project file. You can click the **Choose** button to specify the file. The **Choose File** dialog box appears (Figure 5.14 on page 97).
Breakpoint Mode checkbox	Specifies the breakpoint mode: • Automatic — CodeWarrior software determines when to use software or hardware breakpoints. • Software — CodeWarrior software always uses software breakpoints. • Hardware — CodeWarrior software always uses the available hardware breakpoints.	Software breakpoints contain debug instructions that the debugger writes into your code. You cannot set such breakpoints in flash, as it is read-only. Hardware breakpoints use the on-chip debugging capabilities of the DSP56800. The number of available hardware breakpoints limits these capabilities. Note, Breakpoint Mode only effects HW targets.

Table 5.7 Debugging M56800 Target Panel Elements (*continued*)

Element	Purpose	Comments
Auto-clear previous hardware breakpoint	Checked — Automatically clears the previous harware breakpoint. Clear — Does not Automatically clears the previous harware breakpoint.	
Initialize OMR for program memory checkbox	Checked — Choose the program memory mode (external or internal) at connect. Clear — OMR is unchanged.	
Processor list box	Specifies the processor	Currently this selects the register layout.

Figure 5.14 Choose File Dialog Box



Listing 5.3 Flash Configuration File Line Format

```
baseAddr  startAddr  endAddr  progMem  regBaseAddr  Terase
TME  Tnvs  Tpgs  Tprog  Tnvh  Tnvh1  Trcv
```

Each text line of the configuration file specifies a flash unit on the target. The prototype is shown in Listing 5.3 on page 98 and its arguments are shown in Table 5.8 on page 98.

Table 5.8 Flash Configuration File Line Format

Argument	Description
`baseAddr`	address where row 0 (zero) starts
`startAddr`	first flash memory address
`endAddr`	last flash memory address
`progMem`	0 = data (X:), 1 = program memory (P:)
`regBaseAddr`	location in data memory map where the control registers are mapped
Terase	erase time
TME	mass erase time
Tnvs	PROG/ERASE to NVSTR set up time
Tpgs	NVSTR to program set up time
Tprog	program time
Tnvh	NVSTR hold time
Tnvh1	NVSTR hold time(mass erase)
Trcv	recovery time

A sample flash configuration file for DSP56F803 and DSP56F805 is in Table 5.9 on page 98. Do not change the contents of this file.

Table 5.9 Sample Flash Configuration File for DSP56F803/5

0	0x0004	0x7dff	1	0x0f40	0x0002	0x0006	0x001A	0x0033	0x0066	0x001A	0x019A	0x0006

Table 5.9 Sample Flash Configuration File for DSP56F803/5 (*continued*)

0	0x8000	0x87ff	1	0x0f80	0x0002	0x0006	0x001A	0x0033	0x0066	0x001A	0x019A	0x0006
0	0x1000	0x1fff	0	0x0f60	0x0002	0x0006	0x001A	0x0033	0x0066	0x001A	0x019A	0x0006

NOTE You cannot use Flash ROM with the board set in development mode. Ensure the **Debugger sets OMR on launch** is not enabled if you are using this feature.

Remote Debug Options

Use the Remote Debug Options panel (Figure 5.15 on page 99) to specify different remote debug options.

Figure 5.15 Remote Debug Options

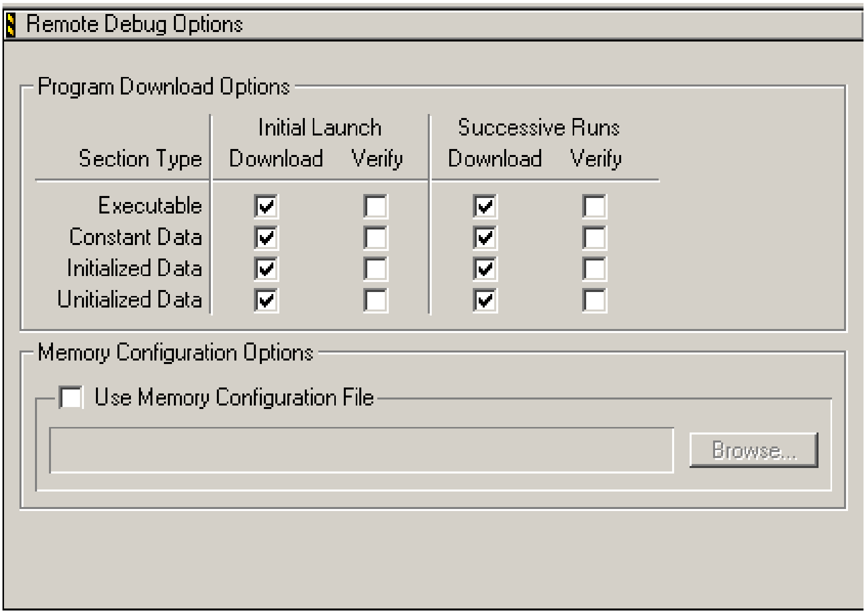


Table 5.10 Remote Debug Options Panel Elements

Element	Purpose	Comments
Program Download Options area	Checked Download checkboxes specify the section types to be downloaded on initial launch and on successive runs. Checked Verify checkboxes specify the section types to be verified (that is, read back to the linker).	Section types: • Executable — program-code sections that have X flags in the linker command file. • Constant Data — program-data sections that do not have X or W flags in the linker command file. • Initialized Data — program-data sections with initial values. These sections have W flags, but not X flags, in the linker command file. • Uninitialized Data — program-data sections without initial values. These sections have W flags, but not X flags, in the linker command file.
Use Memory Configuration File checkbox		Not supported at this time.

6

Processor Expert Interface

Your CodeWarrior™ IDE features a Processor Expert™ plug-in interface, for rapid development of embedded applications. This chapter explains Processor Expert concepts, and Processor Expert additions to the CodeWarrior visual interface. This chapter includes a brief tutorial exercise.

This chapter contains these sections:

Processor Expert Overview

The Processor Expert Interface (PEI) is an integrated development environment for applications based on DSP56800/E (or many other) embedded microcontrollers. It reduces development time and cost for applications. Its code makes very efficient use of microcontroller and peripheral capabilities. Furthermore, it helps develop code that is highly portable.

Features include:

- **Embedded Beans™ components** — Each bean encapsulates a basic functionality of embedded systems, such as CPU core, CPU on-chip peripherals, and virtual devices. To create an application, you select, modify, and combine the appropriate beans.
  - The **Bean Selector** window lists all available beans, in an expandable tree structure. The Bean Selector describes each bean; some descriptions are extensive.
  - The **Bean Inspector** window lets you modify bean properties, methods, events, and comments.
- **Processor Expert page** — This additional page for the CodeWarrior project window lists project CPUs, beans, and modules, in a tree structure. Selecting or double-clicking items of the page opens or changes the contents of related Processor Expert windows.
- **Target CPU window** — This window depicts the target microprocessor as a simple package or a package with peripherals. As you move the cursor over this picture's

pins, the window shows pin numbers and signals. Additionally, you can have this window show a scrollable block diagram of the microprocessor.

- **CPU Structure window** — This window shows the relationships of all target-microprocessor elements, in an expandable-tree representation.
- **CPU Types Overview** — This reference window lists all CPUs that your Processor Expert version supports.
- **Memory Map** — This window shows the CPU address space, plus mapping for internal and external memory.
- **Resource Meter** — This window shows the resource allocation for the target microprocessor.
- **Peripheral Usage Inspector** — This window shows which bean allocates each on-chip peripheral.
- **Installed Beans Overview** — This reference window provides information about all installed beans in your Processor Expert version.
- **Driver generation** — The PEI suggests, connects, and generates driver code for embedded-system hardware, peripherals, and algorithms.
- **Top-Down Design** — A developer starts design by defining application behavior, rather than by focussing on how the microcontroller works.
- **Extensible beans library** — This library supports many microprocessors, peripherals, and virtual devices.
- **Beans wizard** — This external tool helps developers create their own custom beans.
- **Extensive help information** — You access this information either by selecting Help from the Program Expert menu, or by clicking the Help button of any Processor Expert window or dialog box.

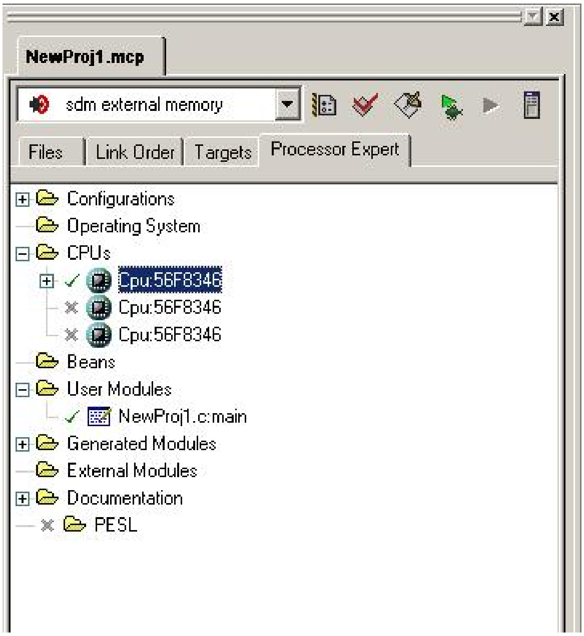
Processor Expert Code Generation

The PEI manages your CPU and other hardware resources, so that you can concentrate on virtual prototyping and design. Your steps for application development are:

1. Creating a CodeWarrior project, specifying the Processor Expert stationery appropriate for your target processor.
2. Configuring the appropriate CPU bean.
3. Selecting and configuring the appropriate function beans.
4. Starting code design (that is, building the application).

As you create the project, the project window opens in the IDE main window. This project window has a Processor Expert page (Figure 6.1 on page 103). The Processor Expert Target CPU window also appears at this time. So does the Processor Expert bean selector window, although it is behind the Target CPU window.

Figure 6.1 Project Window: Processor Expert Page



When you start code design, the PEI generates commented code from the bean settings. This code generation takes advantage of the Processor Expert CPU knowledge system and solution bank, which consists of hand-written, tested code optimized for efficiency.

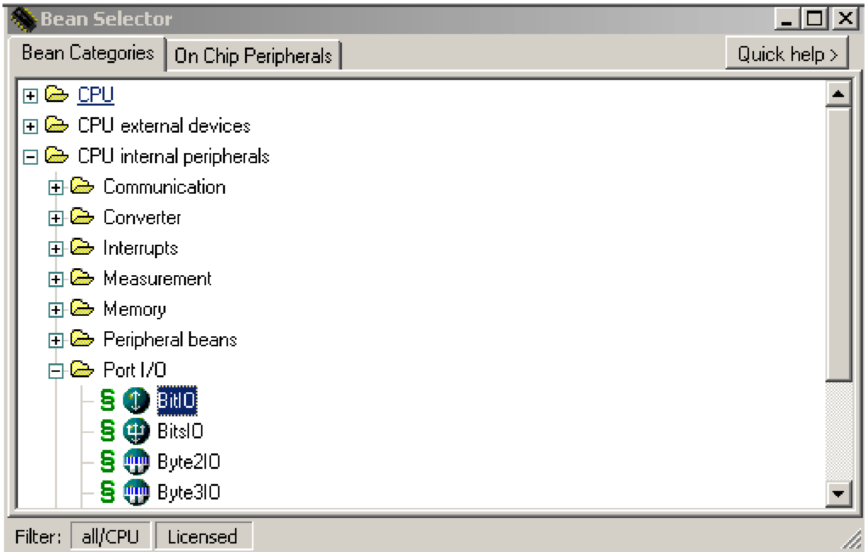
To add new functionalities, you select and configure additional beans, then restart code design. Another straightforward expansion of PEI code is combining other code that you already had produced with different tools.

Processor Expert Beans

Beans encapsulate the most-required functionalities for embedded applications. Examples include port bit operations, interrupts, communication timers, and A/D converters.

The Bean Selector (Figure 6.2 on page 104) helps you find appropriate beans by category: processor, MCU external devices, MCU internal peripherals, or on-chip peripherals. To open the bean selector, select **Processor Expert** > **View** > **Bean Selector**, from the main-window menu bar.

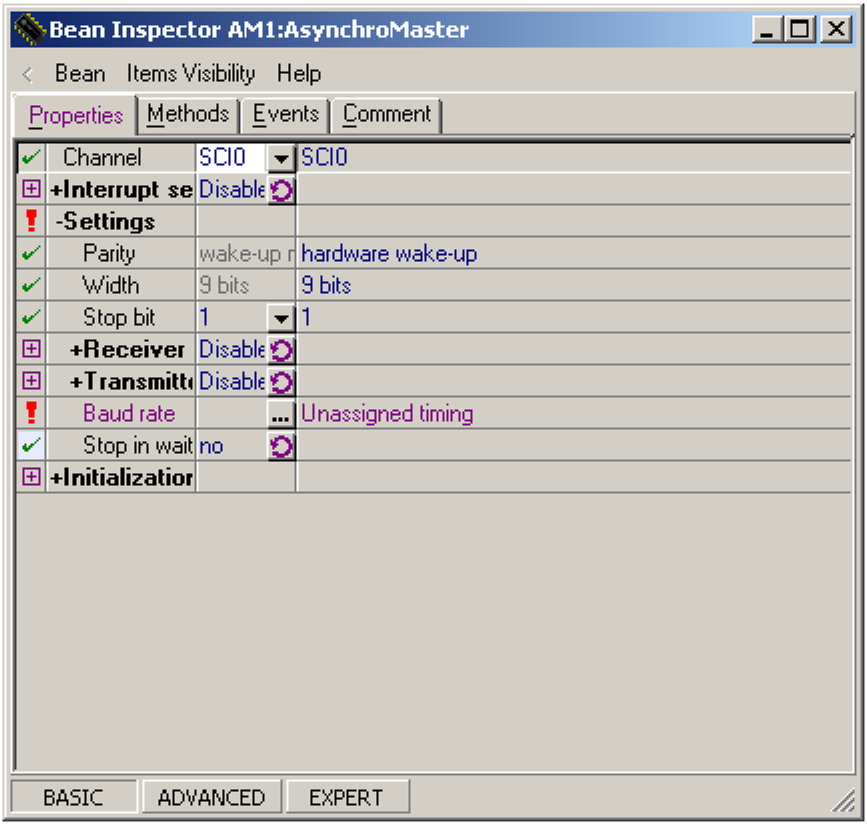
Figure 6.2 Bean Selector



The bean selector's tree structures list all available beans; double-clicking the name adds the bean to your project. Clicking the Quick Help button opens or closes an explanation pane that describes the highlighted bean.

Once you determine the appropriate beans, you use the Bean Inspector (Figure 6.3 on page 105) to fine tune each bean, making it optimal for your application.

Figure 6.3 Bean Inspector



Using the Bean Inspector to set a bean's initialization properties automatically adds bean initialization code to CPU initialization code. You use the Bean Inspector to adjust bean properties, so that generated code is optimal for your application.

Beans greatly facilitate management of on-chip peripherals. When you choose a peripheral from bean properties, the PEI presents all possible candidates. But the PEI indicates which candidates already are allocated, and which are not compatible with current bean settings.

Processor Expert Menu

Table 6.1 on page 106 explains the selections of the Processor Expert menu.

Table 6.1 Processor Expert Menu Selections

Item	Subitem	Action
Open Processor Expert	none	Opens the PEI for the current project. (Available only if the current project does not already involve the PEI.)
Code Design <Project>	none	Generates code, including drivers, for the current project. Access these files via the Generate Code folder, of the project-window Files page.
Undo Last Code Design	none	Deletes the most recently-generated code, returning project files to their state after the previous, error-free code generation.
View	Project Panel	Brings the Processor Expert page to the front of the CodeWarrior project window. (Not available if the project window does not include a Processor Expert page.)
	Bean Inspector	Opens the Bean Inspector window, which gives you access to bean properties.
	Bean Selector	Opens the Beans Selector window, which you use to choose the most appropriate beans.
	Target CPU Package	Opens the Target CPU Package window, which depicts the processor. As you move your cursor over the pins of this picture, text boxes show pin numbers and signal names.
	Target CPU Block Diagram	Opens the Target CPU Package window, but portrays the processor as a large block diagram. Scroll bars let you view any part of the diagram. As you move your cursor over modules, floating text boxes identify pin numbers and signals.
	Error Window	Opens the Error Window, which shows hints, warnings, and error messages.

Table 6.1 Processor Expert Menu Selections (*continued*)

Item	Subitem	Action
	Resource Meter	Opens the Resource Meter window, which shows usage and availability of processor resources.
View *(continued)*	Target CPU Structure	Opens the CPU Structure window, which uses an expandible tree structure to portray the processor.
	Peripherals Usage Inspector	Opens the Peripherals Usage Inspector window, which shows which bean allocates each peripheral.
	Peripheral Initialization Inspector	Opens the Peripherals Initialization Inspector window, which show the initialization value and value after reset for all peripheral register bits.
	Installed Beans Overview	Opens the Beans Overview window, which provides information about all beans in your project.
	CPU Types Overview	Opens the CPU Overview window, which lists supported processors in an expandable tree structure.
	CPU Parameters Overview	Opens the CPU Parameters window, which lists clock-speed ranges, number of pins, number of timers, and other reference information for the supported processors.
	Memory Map	Opens the Memory Map window, which depicts CPU address space, internal memory, and external memory.
Tools	<tool name>	Starts the specified compiler, linker or other tool. (You use the Tools Setup window to add tool names to this menu.)
	SHELL	Opens a command-line window.
	Tools Setup	Opens the Tools Setup window, which you use to add tools to this menu.
Help	Processor Expert Help	Opens the help start page.

Table 6.1 Processor Expert Menu Selections (*continued*)

Item	Subitem	Action
	Introduction	Opens the PEI help introduction.
	Benefits	Opens an explanation of PEI benefits.
	User Interface	Opens an explanation of the PEI environment.
	Tutorial	[None available for the DSP56800/E.]
	Quick Start	Opens PEI quick start instructions.
Help (continued)	Embedded Beans	Opens the first page of a description database of all beans.
	Embedded Beans Categories	Opens the first page of a description database of beans, organized by category.
	Supported CPUs, Compilers, and Debuggers	Opens the list of processors and tools that the PEI plug-in supports.
	PESL Library User Manual	Opens the Processor Expert System Library, for advanced developers.
	User Guide	Opens a .pdf guide that focuses on the DSP56800/E processor family.
	Search in PDF Documentation of the Target CPU	Opens documentation of the target processor, in a .pdf search window.
	Go to Processor Expert Home Page	Opens your default browser, taking you to the PEI home page.
	About Processor Expert	Opens a standard About dialog box for the PEI.
Update	Update Processor Exert Beans from Package	Opens the Open Update Package window. You can use this file-selection window to add updated or new beans (which you downloaded over the web) to your project.

Table 6.1 Processor Expert Menu Selections (*continued*)

Item	Subitem	Action
	Check Processor Expert Web for updates	Checks for updates available over the web. If any are available, opens your default browser, so that you can download them.
Bring PE Windows to Front	none	Moves PEI windows to the front of your monitor screen.
Arrange PE Windows	none	Restores the default arrangement of all open PEI windows.

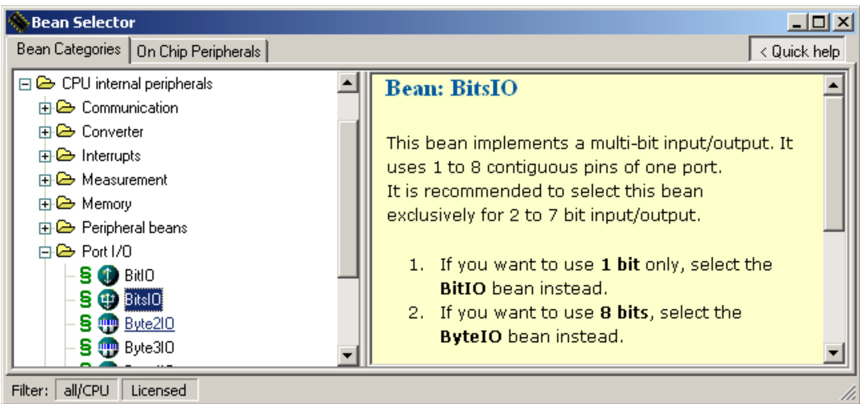
Processor Expert Windows

This section illustrates important Processor Expert windows and dialog boxes.

Bean Selector

The **Bean Selector** window (Figure 6.4 on page 109) explains which beans are available, helping you identify those most appropriate for your application project. To open this window, select **Processor Expert > View > Bean Selector**, from the main-window menu bar.

Figure 6.4 Bean Selector Window



The **Bean Categories** page, at the left side of this window, lists the available beans in category order, in an expandable tree structure. Green string bean symbols identify beans that have available licenses. Grey string bean symbols identify beans that do not have available licenses.

The **On-Chip Peripherals** page lists beans available for specific peripherals, also in an expandable tree structure. Yellow folder symbols identify peripherals fully available. Light blue folder symbols identify partially used peripherals. Dark blue folder symbols identify fully used peripherals.

Bean names are black; bean template names are blue. Double-click a bean name to add it to your project.

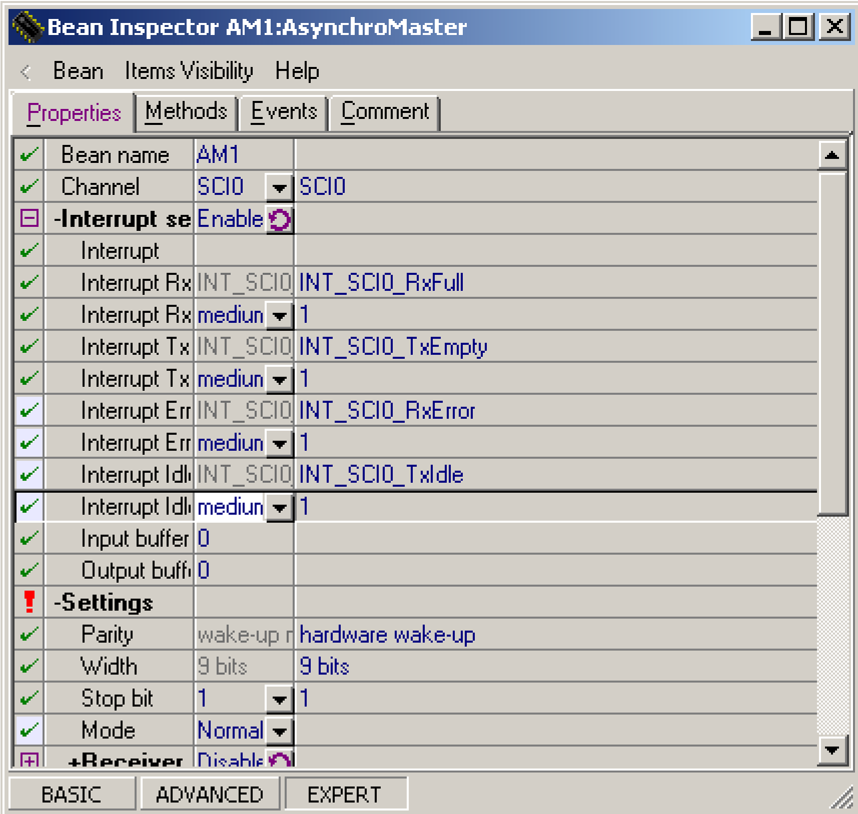
Click the Quick Help button to add the explanation pane to the right side of the window, as Figure 6.4 on page 109 shows. This pane describes the selected (highlighted) bean. Use the scroll bars to read descriptions that are long.

Click the two buttons at the bottom of the window to activate or deactivate filters. If the **all/CPU** filter is active, the window lists only the beans for the target CPU. If the license filter is active, the window lists only the beans for which licenses are available.

Bean Inspector

The **Bean Inspector** window (Figure 6.5 on page 111) lets you modify bean properties and other settings. To open this window, select **Processor Expert > View > Bean Inspector**, from the main-window menu bar.

Figure 6.5 Bean Inspector Window



This window shows information about the currently selected bean — that is, the highlighted bean name in the project-window Processor Expert page. The title of the Bean Inspector window includes the bean name.

The Bean Inspector consists of Properties, Methods, Events, and Comment pages. The first three pages have these columns:

- **Item names** — Items to be set. Double-click on group names to expand or collapse this list. For the Method or Event page, double-clicking on an item may open the file editor, at the corresponding code location.
- **Selected settings** — Possible settings for your application. To change any ON/OFF-type setting, click the circular-arrow button. Settings with multiple possible values have triangle symbols: click the triangle to open a context menu, then select the appropriate value. Timing settings have an ellipsis (...) button: click this button to open a setting dialog box.

- **Setting status** — Current settings or error statuses.

Use the comments page to write any notations or comments you wish.

NOTE If you have specified a target compiler, the Bean Inspector includes an additional Build options page for the CPU bean.
If your project includes external peripherals, the Bean Inspector includes an additional Used page. Clicking a circular-arrow button reserves a resource for connection to an external device. Clicking the same button again frees the resource.

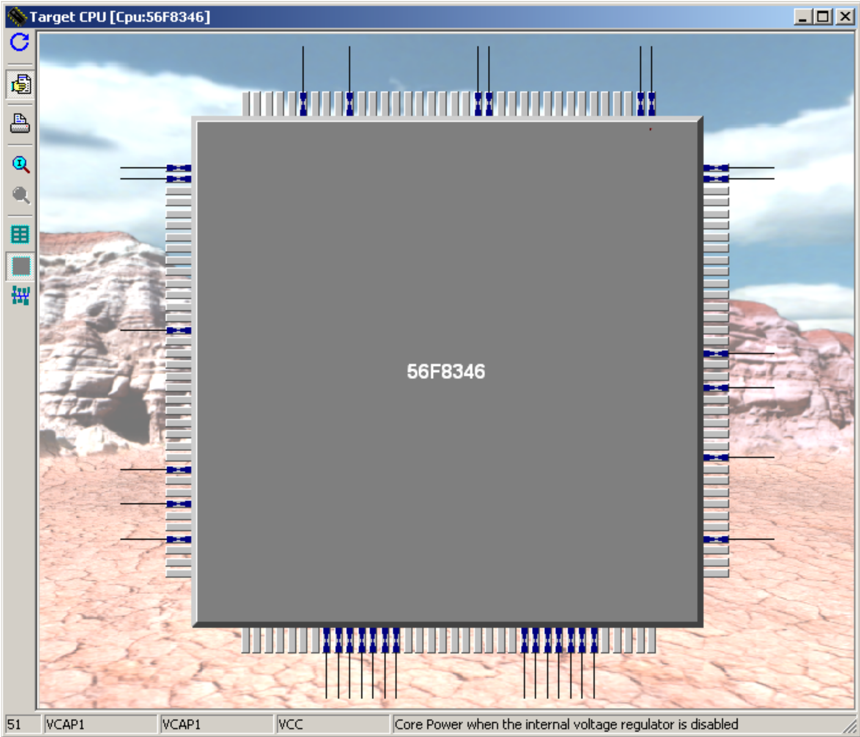
The Basic, Advanced, and Expert view mode buttons, at the bottom of the window, let you change the detail level of Bean Inspector information.

The Bean Inspector window has its own menu bar. Selections include restoring default settings, saving the selected bean as a template, changing the bean's icon, disconnecting from the CPU, and several kinds of help information.

Target CPU Window

The **Target CPU** window (Figure 6.6 on page 113) depicts the target processor as a realistic CPU package, as a CPU package with peripherals, or as a block diagram. To open this window, select **Processor Expert > View > Target CPU Package**, from the main-window menu bar. (To have this window show the block diagram, you may select **Processor Expert > View > Target CPU Block Diagram**, from the main-window menu bar.)

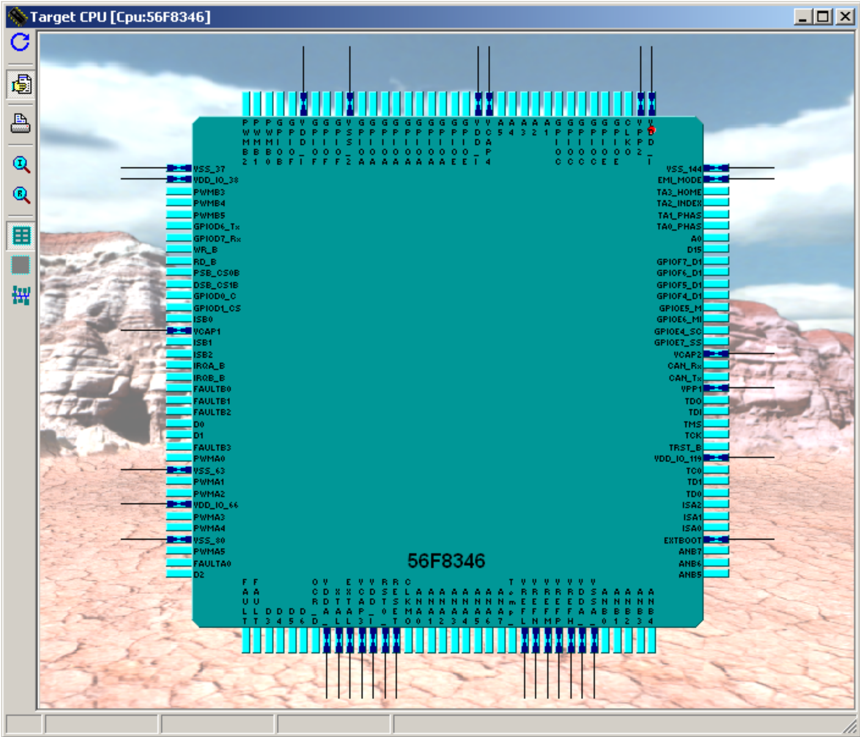
Figure 6.6 Target CPU Window: Package



Arrows on pins indicate input, output, or bidirectional signals. As you move your cursor over the processor pins, text boxes at the bottom of this window show the pin numbers and signal names.

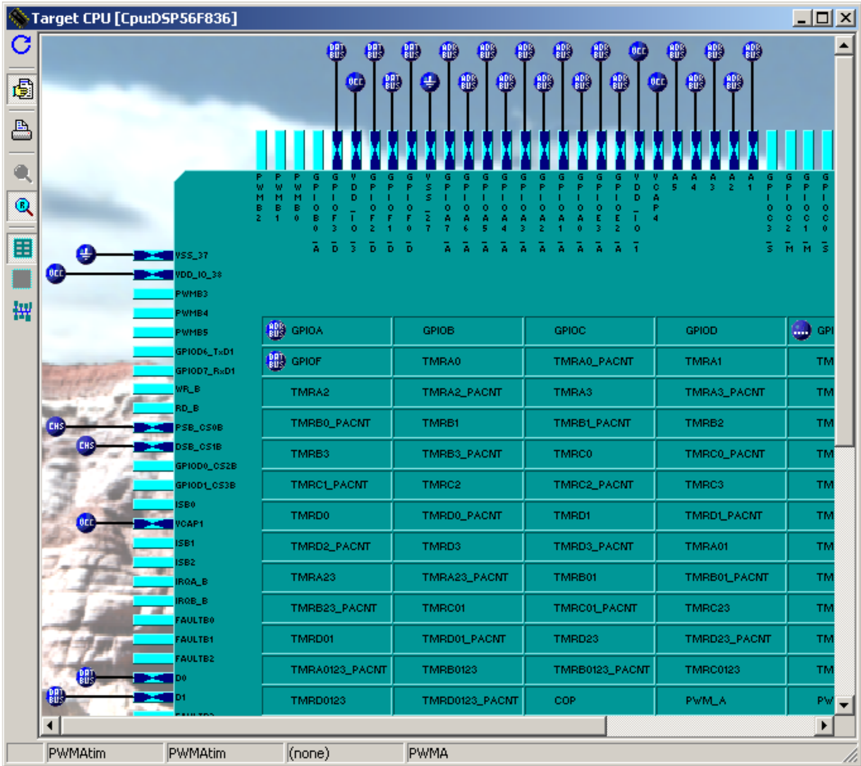
Use the control buttons at the left edge of this window to modify the depiction of the processor. One button, for example, changes the picture view the CPU package with peripherals. However, as Figure 6.7 on page 114 shows, it is not always possible for the picture of a sophisticated processor to display internal peripherals.

Figure 6.7 Target CPU Window: Package and Peripherals



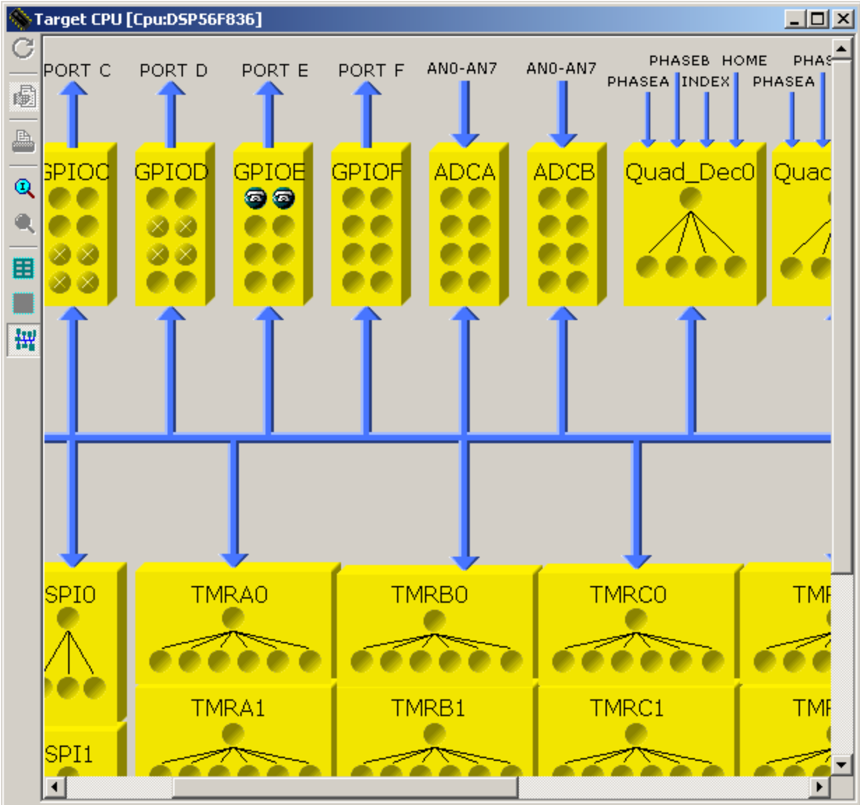
In such a case, you can click the **Always show internal peripheral devices** control button. Figure 6.8 on page 115 shows that this expands the picture size, as necessary, to allow the peripheral representations. This view also includes bean icons (blue circles) attached to the appropriate processor pins. Use the scroll bars to view other parts of the processor picture.

Figure 6.8 Target CPU Window: Peripherals and Bean Icons



Click the Show MCU Block Diagram to change the picture to a block diagram, as Figure 6.9 on page 116 shows. Use the scroll bars to view other parts of the diagram. (You can bring up the block diagram as you open the Target CPU window, by selecting **Processor Expert > View > Target CPU Block Diagram**, from the main-window menu bar.)

Figure 6.9 Target CPU Window: Block Diagram



Other control buttons at the left edge of the window let you:

- Show bean icons attached to processor pins.
- Rotate the CPU picture clockwise 90 degrees.
- Toggle default and user-defined names of pins and peripherals.
- Print the CPU picture.

NOTE As you move your cursor over bean icons, peripherals, and modules, text boxes or floating hints show information such as names, descriptions, and the allocating beans.

And note these additional mouse control actions for the Target CPU window:

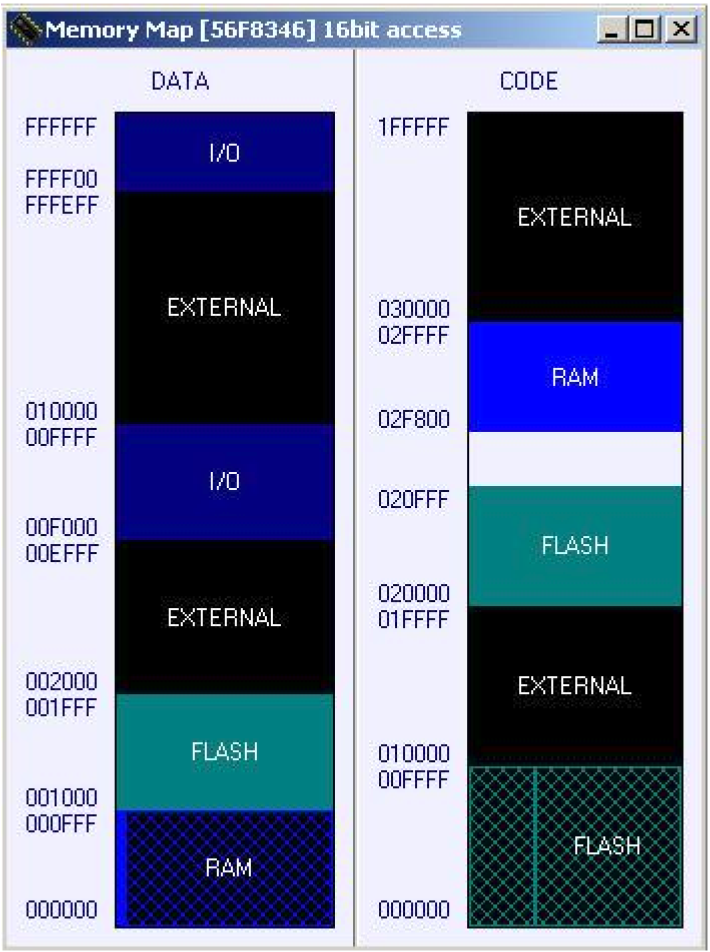
- Clicking a bean icon selects the bean in the project window's Processor Expert page.

- Double-clicking a bean icon open the Bean Inspector, displaying information for that bean.
- Right-clicking a bean icon, a pin, or a peripheral opens the corresponding context menu.
- Double-clicking an ellipsis (...) bean icon opens a context menu of all beans using parts of the peripheral. Selecting one bean from this menu opens the Bean Inspector.
- Right-clicking an ellipsis (...) bean icon opens a context menu of all beans using parts of the peripheral. Selecting one bean from this menu opens the bean context menu.

Memory Map Window

The **Memory Map** window (Figure 6.10 on page 118) depicts CPU address space, and the map of internal and external memory. To open this window, select **Processor Expert > View** > **Memory Map**, from the main-window menu bar.

Figure 6.10 Memory Map Window



The color key for memory blocks is:

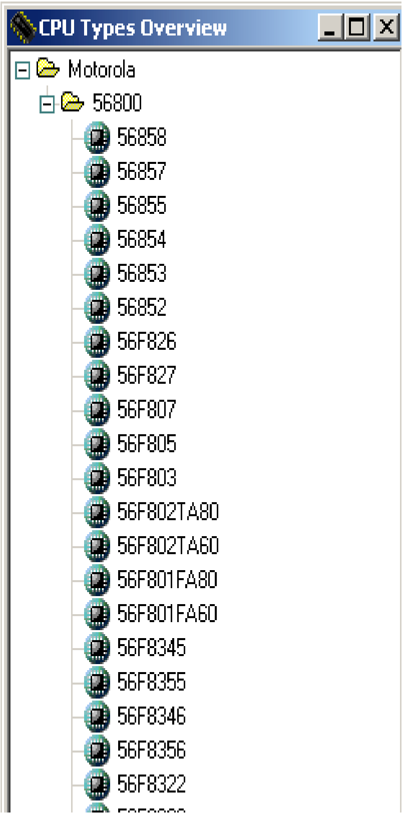
- White — Non-usable space
- Dark Blue — I/O space
- Medium Blue — RAM
- Light Blue — ROM
- Cyan — FLASH memory or EEPROM
- Black — External memory.

Pause your cursor over any block of the map to bring up a brief description.

CPU Types Overview

The **CPU Types Overview** window (Figure 6.11 on page 119) lists supported processors, in an expandable tree structure. To open this window, select **Processor Expert > View > CPU Types Overview**, from the main-window menu bar.

Figure 6.11 CPU Types Overview Window

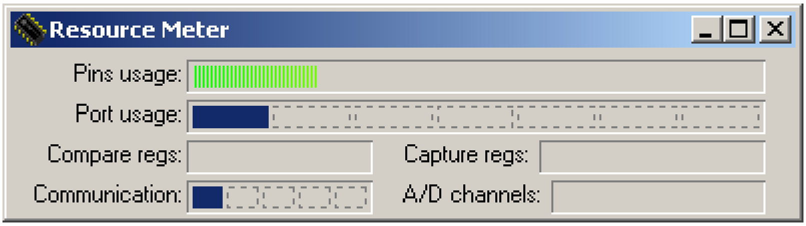


Right-click the window to open a context menu that lets you add the selected CPU to the project, expand the tree structure, collapse the tree structure, or get help information.

Resource Meter

The **Resource Meter** window (Figure 6.12 on page 120) shows the usage or availability of processor resources. To open this window, select **Processor Expert > View > Resource Meter**, from the main-window menu bar.

Figure 6.12 Resource Meter Window



Bars of this window indicate:

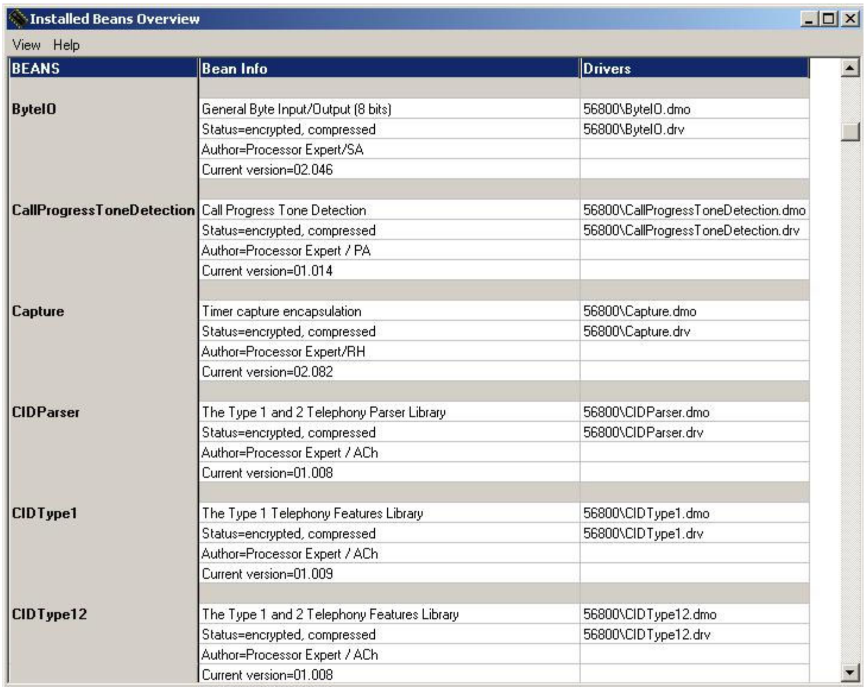
- The number of pins used
- The number of ports used
- Allocation of timer compare registers
- The number of timer capture registers used
- Allocation of serial communication channels
- Allocation of A/D converter channels.

Pausing your cursor over some fields of this window brings up details of specific resources.

Installed Beans Overview

The **Installed Beans Overview** window (Figure 6.13 on page 121) shows reference information about the installed beans. To open this window, select **Processor Expert > View > Installed Beans Overview**, from the main-window menu bar.

Figure 6.13 Installed Beans Overview Window

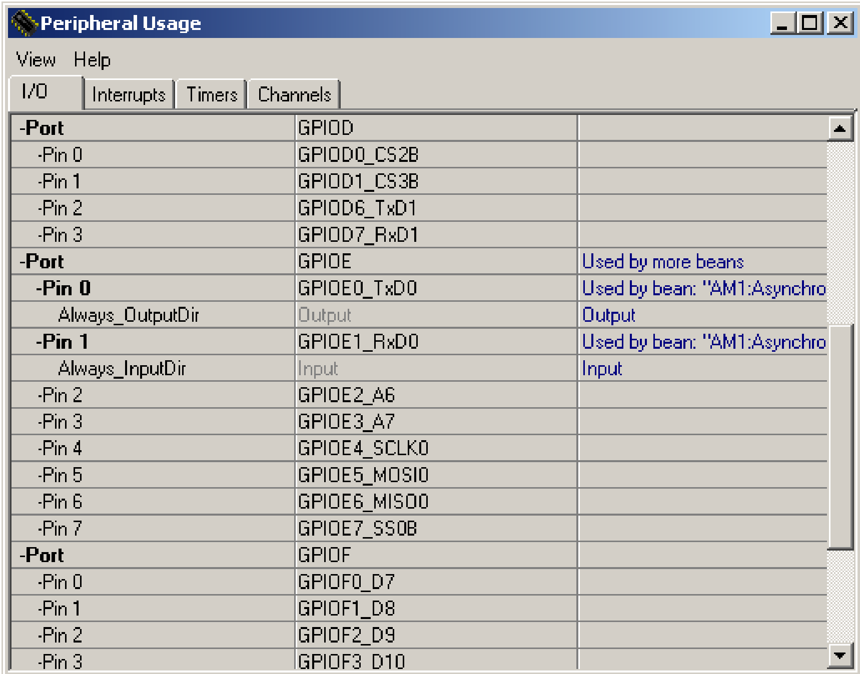


This window's View menu lets you change the display contents, such as showing driver status and information, restricting the kinds of beans the display covers, and so on.

Peripherals Usage Inspector

The **Peripherals Usage** window (Figure 6.14 on page 122) shows which bean allocates each peripheral. To open this window, select **Processor Expert > View > Peripherals Usage Inspector**, from the main-window menu bar.

Figure 6.14 Peripherals Usage Window



The pages of this window reflect the peripheral categories: I/O, interrupts, timers, and channels. The columns of each page list peripheral pins, signal names, and the allocating beans.

Pausing your cursor over various parts of this window brings up brief descriptions of items. This window’s View menu lets you expand or collapse the display.

Processor Expert Tutorial

This tutorial exercise generates code that flashes the LEDs of a MC56F8346E development board. Follow these steps:

1. Create a project:
   a. Start the CodeWarrior IDE, if it is not started already.
   b. From the main-window menu bar, select **File** > **New**. The **New** window appears.
   c. In the Project page, select (highlight) **Processor Expert Examples Stationery**.
   d. In the Project name text box, enter a name for the project, such as `LEDcontrol`.

e. Click the **OK** button. The **New Project** window replaces the **New** window.

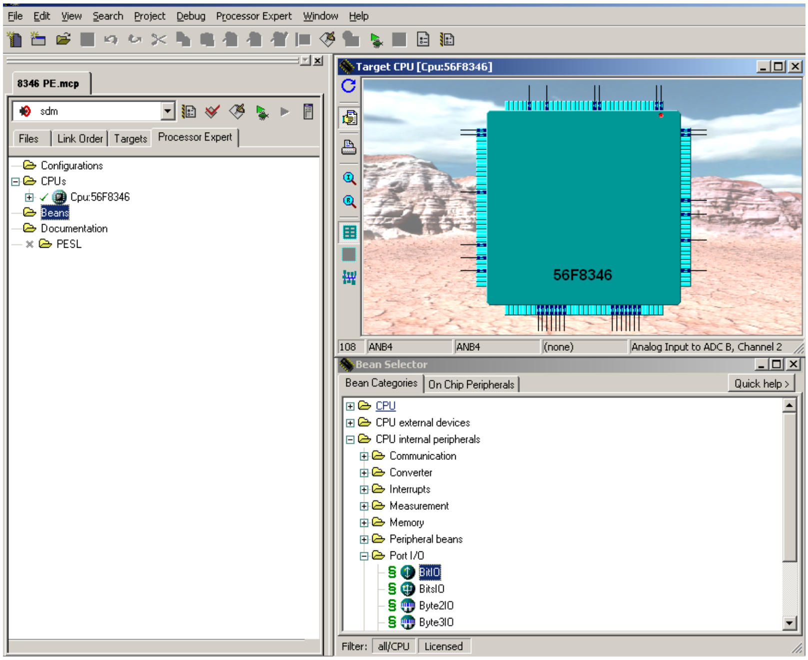
f. In the Project Stationery list, select **TestApplications** > **Tools** > **LED** > **56858**.

g. Click the **OK** button.

h. Click the **OK** button. The IDE:

- Opens the project window, docking it the left of the main window. This project window includes a Processor Expert page.
- Opens the **Target CPU** window, as Figure 6.15 on page 123 shows. This window shows the CPU package and peripherals view.
- Opens the **Bean Selector** window, behind the **Target CPU** window.

Figure 6.15 Project, Target CPU Windows



2. Select the sdm external memory target.

   a. Click the project window’s Targets tab. The Targets page moves to the front of the window.

   b. Click the target icon of the **sdm external memory** entry. The black arrow symbol moves to this icon, confirming your selection.

3. Add six BitIO beans to the project.

a. Click the project window's Processor Expert tab. The Processor Expert page moves to the front of the window.

b. Make the **Bean Selector** window visible:

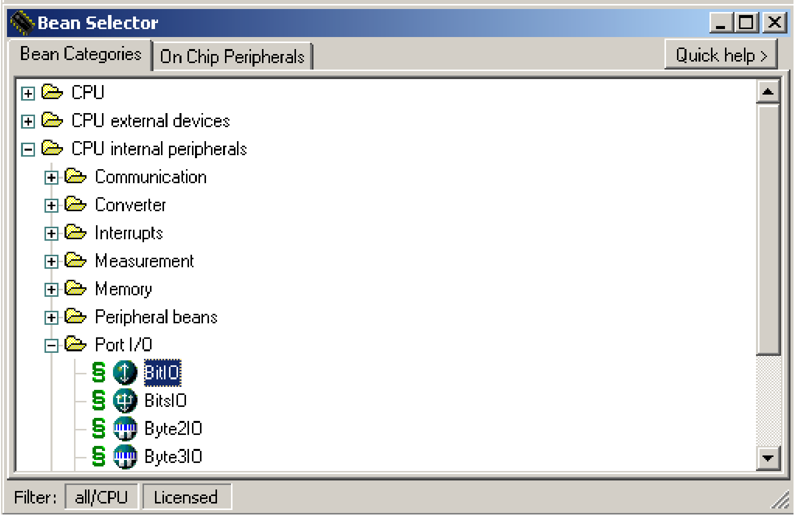
- Minimize the **Target CPU** window.
- Select **Processor Expert > View > Bean Selector**, from the main-window menu bar.

c. In the Bean Categories page, expand the entry **MCU internal peripherals**.

d. Expand the subentry **Port I/O**.

e. Double-click the **BitIO** bean name six times. (Figure 6.16 on page 124 depicts this bean selection.) The IDE adds these beans to your project; new bean icons appear in the project window's Processor Expert page.

Figure 6.16 Bean Selector: BitIO Selection



NOTE If new bean icons do not appear in the Processor Expert page, the system still may have added them to the project. Close the project, then reopen it. When you bring the Processor Expert page to the front of the project window, the page should show the new bean icons.

4. Add two ExtInt beans to the project.

   a. In the Bean Categories page of the Bean Selector window, expand the **Interrupts** subentry.

   b. Double-click the **ExtInt** bean name two times. The IDE adds these beans to your project; new bean icons appear in the Processor Expert page.

   c. You may close the **Bean Inspector** window.

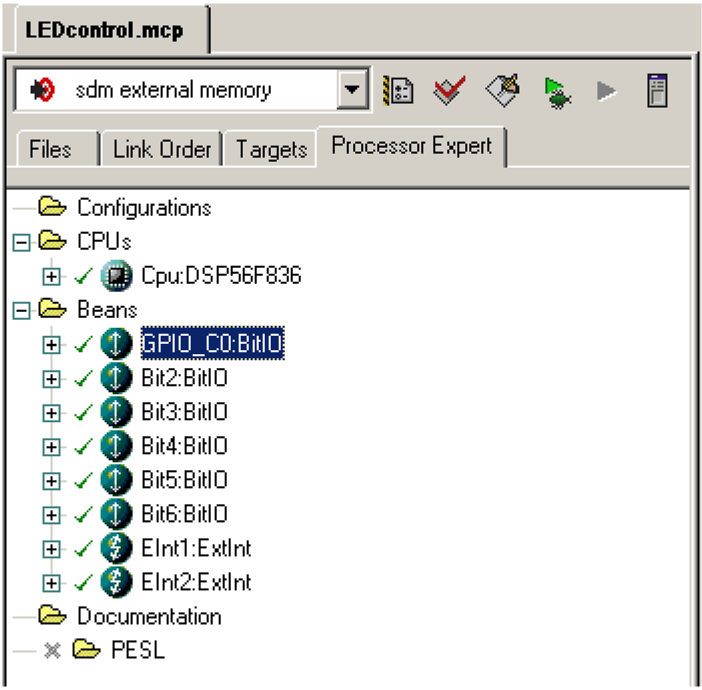
5. Rename the eight beans GPIO_C0 — GPIO_C3, GPIO_D6, GPIO_D7, IRQA, and IRQB.

   a. In the project window's Processor Expert page, right-click the name of the first BitIO bean. A context menu appears.

   b. Select **Rename Bean**. A change box appears around the bean name.

   c. Type the new name `GPIO_C0`, then press the Enter key. The list shows the new name; as Figure 6.17 on page 125 shows, this name still ends with `BitIO`.

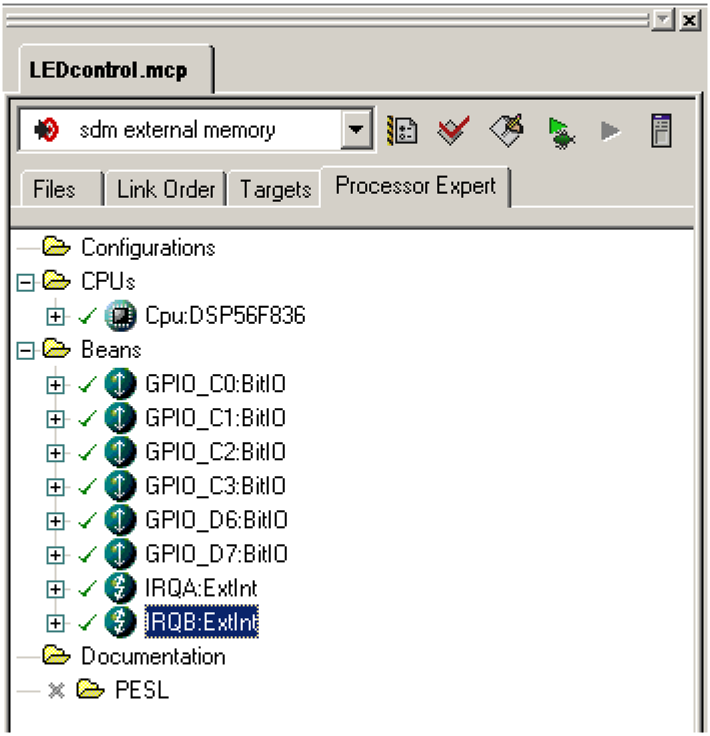
Figure 6.17 New Bean Name



   d. Repeat substeps a, b, and c for each of the other BitIO beans, renaming them `GPIO_C1`, `GPIO_C2`, `GPIO_C3`, `GPIO_D6`, and `GPIO_D7`.

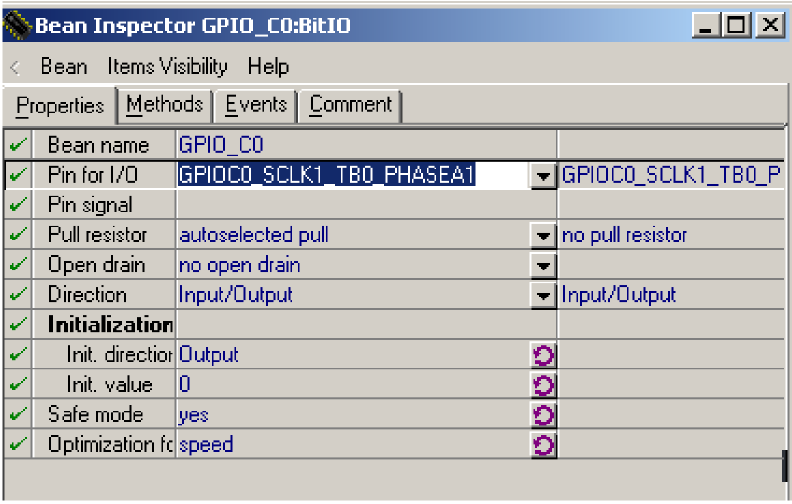
e. Repeat substeps a, b, and c for the two ExtInt beans, renaming them `IRQA` and `IRQB`. (Figure 6.18 on page 126 shows the Processor Expert page at this point.)

Figure 6.18 New Bean Names



6. Update pin associations for each bean.
   a. In the Processor Expert page, double-click the bean name `GPIO_C0`. The **Bean Inspector** window opens, displaying information for this bean.
   b. Use standard window controls to make the middle column of the Properties page about 2 inches wide.
   c. In the **Pin for I/O** line, click the triangle symbol of the middle-column list box. The list box opens.
   d. Use this list box to select **GPIOC0_SCLK1_TB0_PHASEA1**. (Figure 6.19 on page 127 depicts this selection.)

Figure 6.19 New Pin Association



e. In the project window's Processor Expert page, select the bean name `GPIO_C1`. The Bean Inspector information changes accordingly.

f. Use the **Pin for I/O** middle-column list box to select **GPIOC1_MOSI1_TB1_PHASEB1**.

g. Repeat substeps e and f, for bean `GPIO_C2`, to change its associated pin to **GPIOC2_MISO1_TB2_INDEX1**.

h. Repeat substeps e and f, for bean `GPIO_C3`, to change its associated pin to **GPIOC3_SSA_B_TB3_HOME1**.

i. Repeat substeps e and f, for bean `GPIO_D6`, to change its associated pin to **GPIOD6_TxD1**.

j. Repeat substeps e and f, for bean `GPIO_D7`, to change its associated pin to **GPIOD7_RxD1**.

k. In the project window's Processor Expert page, select the bean name `IRQA`. The Bean Inspector information changes accordingly.

l. Use the **Pin** middle-column list box to select **IRQA_B**.

m. Repeat substeps k and l, for bean `IRQB`, to change its associated pin to **IRQB_B**.

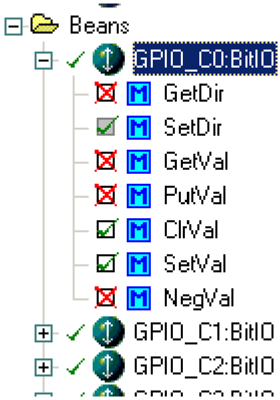
n. You may close the **Bean Inspector** window.

7. Enable BitIO SetDir, ClrVal, and SetVal functions.

   a. In the Processor Expert page, click the plus-sign control for the `GPIO_C0` bean. The function list expands: red X symbols indicate disabled functions, green check symbols indicate enabled functions.

   b. Double-click function symbols as necessary, so that only **SetDir**, **ClrVal**, and **SetVal** have green checks. (Figure 6.20 on page 128 shows this configuration.)

Figure 6.20 GPIO_C3 Enabled Functions



   c. Click the `GPIO_C0` minus-sign control. The function list collapses.

   d. Repeat substeps a, b, and c for beans `GPIO_C1`, `GPIO_C2`, `GPIO_C3`, `GPIO_D6`, and `GPIO_D7`.

8. Enable ExtInt OnInterrupt, GetVal functions.

   a. In the Processor Expert page, click the plus-sign control for the `IRQA` bean. The function list expands.

   b. Double-click function symbols as necessary, so that only **OnInterrupt** and **GetVal** have green check symbols.

   c. Click the `IRQA` minus-sign control. The function list collapses.

   d. Repeat substeps a, b, and c for bean `IRQB`.

9. Design (generate) project code.

   a. From the main-window menu bar, select **Processor Expert > Code Design ‘LEDcontrol.mcp**.’ (This selection shows the actual name of your project.) The IDE and PEI generate several new files for your project.

   b. You may close all windows except the project window.

10. Update file Events.c.

   a. Click the project window’s Files tab. The Files page moves to the front of the window.

b. Expand the **User Modules** folder.

c. Double-click filename **Events.c**. An editor window opens, displaying this file's text. (Listing 6.1 on page 129, at the end of this tutorial, shows this file's contents.)

d. Find the line `IRQB_OnInterrupt()`.

e. Above this line, enter the new line **`extern short IRQB_On;`**.

f. Inside `IRQB_OnInterrupt()`, enter the new line `IRQB_On ^= 1;`.

g. Find the line `IRQA_OnInterrupt()`.

h. Above this line, enter the new line **`extern short IRQA_On;`**.

i. Inside `IRQA_OnInterrupt()`, enter the new line `IRQA_On ^= 1;`.

NOTE Listing 6.1 on page 129 shows these new lines as bold italics.

j. Save and close file Events.c.

11. Update file LEDcontrol.c.

a. In the project window's Files page, double-click filename **LEDcontrol.c** (or the actual .c filename of your project). An editor window opens, displaying this file's text.

b. Add custom code, to utilize the beans.

NOTE Listing 6.2 on page 132 shows custom entries as bold italics. Processor Expert software generated all other code of the file.

c. Save and close the file.

12. Build and debug the project.

a. From the main-window menu bar, select **Project** > **Make**. The IDE compiles and links your project, generating executable code.

b. Debug your project, as you would any other CodeWarrior project.

This completes the Processor Expert tutorial exercise. Downloading this code to a DSP56836E development board should make the board LEDs flash in a distinctive pattern.

Listing 6.1 File Events.c

```
/*
** ###################################################################
**
**     Filename   : Events.C
**
**     Project    : LEDcontrol
```

```
**
**     Processor : DSP56F836
**
**     Beantype  : Events
**
**     Version   : Driver 01.00
**
**     Compiler  : Metrowerks DSP C Compiler
**
**     Date/Time : 3/24/2003, 1:18 PM
**
**     Abstract  :
**
**         This is user's event module.
**         Put your event handler code here.
**
**     Settings  :
**
**
**     Contents  :
**
**         IRQB_OnInterrupt - void IRQB_OnInterrupt(void);
**         IRQA_OnInterrupt - void IRQA_OnInterrupt(void);
**
**
**     (c) Copyright UNIS, spol. s r.o. 1997-2002
**
**     UNIS, spol. s r.o.
**     Jundrovska 33
**     624 00 Brno
**     Czech Republic
**
**     http      : www.processorexpert.com
**     mail      : info@processorexpert.com
**
** ###########################################################
*/
/* MODULE Events */

/*Including used modules for compilling procedure*/
#include "Cpu.h"
#include "Events.h"
#include "GPIO_C0.h"
#include "GPIO_C1.h"
#include "GPIO_C2.h"
#include "GPIO_C3.h"
#include "GPIO_D6.h"
#include "GPIO_D7.h"
```

```
#include "IRQA.h"
#include "IRQB.h"

/*Include shared modules, which are used for whole project*/
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"

/*
** ==========================================================
**     Event       :  IRQB_OnInterrupt (module Events)
**
**     From bean   :  IRQB [ExtInt]
**     Description :
**         This event is called when the active signal edge/level
**         occurs.
**     Parameters  : None
**     Returns     : Nothing
** ==========================================================
*/
#pragma interrupt called
extern short IRQB_On;
void IRQB_OnInterrupt(void)
{
  IRQB_On ^=1;
  /* place your IRQB interrupt procedure body here */
}


/*
** ==========================================================
**     Event       :  IRQA_OnInterrupt (module Events)
**
**     From bean   :  IRQA [ExtInt]
**     Description :
**         This event is called when the active signal edge/level
**         occurs.
**     Parameters  : None
**     Returns     : Nothing
** ===================================================================
*/
#pragma interrupt called
extern short IRQA_On;
void IRQA_OnInterrupt(void)
{
  IRQA_On ^= 1;
  /* place your IRQA interrupt procedure body here */
```

```
}


/* END Events */

/*
** ###################################################################
**
**     This file was created by UNIS Processor Expert 03.15 for
**     the Freescale DSP56x series of microcontrollers.
**
** #####################################################################
*/
```

Listing 6.2 File LEDcontrol.c

```
/*
** ###################################################################################
**     Filename  : LEDcontrol.C
**
**     Project   : LEDcontrol
**
**     Processor : DSP56F836
**
**     Version   : Driver 01.00
**
**     Compiler  : Metrowerks DSP C Compiler
**
**     Date/Time : 3/24/2003, 1:18 PM
**
**     Abstract  :
**
**         Main module.
**         Here is to be placed user's code.
**
**     Settings  :
**
**
**     Contents  :
**
**         No public methods
**
**
**     (c) Copyright UNIS, spol. s r.o. 1997-2002
**
**     UNIS, spol. s r.o.
**     Jundrovska 33
```

```
**      624 00 Brno
**      Czech Republic
**
**      http       : www.processorexpert.com
**      mail       : info@processorexpert.com
**
** ##################################################################
*/
/* MODULE LEDcontrol */

/* Including used modules for compilling procedure */
#include "Cpu.h"
#include "Events.h"
#include "GPIO_C0.h"
#include "GPIO_C1.h"
#include "GPIO_C2.h"
#include "GPIO_C3.h"
#include "GPIO_D6.h"
#include "GPIO_D7.h"
#include "IRQA.h"
#include "IRQB.h"
/* Include shared modules, which are used for whole project */
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"

/*
 * Application Description:
 *   LED program for the 56836 EVM.
 *
 *   Pattern: "Count" from 0 to 63, using LEDs to represent the bits of
the number.
 *
 *   Pressing the IRQA button flips LED order: commands that previously
went to LED1 go to LED6, and so forth.
 *   Pressing the IRQB button reverses the enabled/disabled LED states.
 *
 */

/* global used as bitfield, to remember currently active bits, used to
 * enable/disable all LEDs. */
long    num = 0;
short   IRQA_On,IRQB_On;

/* simple loop makes LED changes visible to the eye */
void wait(int);
voide wait(int count)
```

```
{
        int i;
        for (i=0; i<count; ++i);
}

/*set the given LED */
void setLED(int);
void setLED(int num)
{
        if (!IRQA_On)
        {
                num = 7-num;
        }
        if (!IRQB_On)
        {
                switch (num)
                {
                        case 1:  GPIO_C0_ClrVal(); break;
                        case 2:  GPIO_C1_ClrVal(); break:
                        case 3:  GPIO_C2_ClrVal(); break;
                        case 4:  GPIO_C3_ClrVal(); break;
                        case 5:  GPIO_D6_ClrVal(); break;
                        case 6:  GPIO_D7_ClrVal(); break;
                }
        }
        else
        {
                switch (num)
                {
                        case 1:  GPIO_C0_SetVal(); break;
                        case 2:  GPIO_C1_SetVal(); break;
                        case 3:  GPIO_C2_SetVal(); break;
                        case 4:  GPIO_C3_SetVal(); break;
                        case 5:  GPIO_D6_SetVal(); break;
                        case 6:  GPIO_D7_SetVal(); break;
                }
        }
}

/* clear the given LED */
void clrLED(int);
void clrLED(int num)
{
        if (!IRQA_On)
        {
                num = 7-num;
        }
        if (IRQB_On)
```

```
        {
                switch (num)
                {
                        case 1:  GPIO_C0_ClrVal(); break;
                        case 2:  GPIO_C1_ClrVal(); break;
                        case 3:  GPIO_C2_ClrVal(): break;
                        case 4:  GPIO_C3_ClrVal(); break;
                        case 5:  GPIO_D6_ClrVal(); break;
                        case 6:  GPIO_D7_ClrVal(); break;
                }
        }
        else
        {
                switch (num)
                {
                        case 1:  GPIO_C0_SetVal(); break;
                        case 2:  GPIO_C1_SetVal(); break;
                        case 3:  GPIO_C2_SetVal(); break;
                        case 4:  GPIO_C3_SetVal(); break;
                        case 5:  GPIO_D6_SetVal(); break;
                        case 6:  GPIO_D7_SetVal(); break;
                }
        }
}

#define CLEARLEDS    showNumberWithLEDs(0)
/* method to set each LED status to reflect the given number/bitfield
*/
void shwNumberWithLEDs(long);
void showNumberWithLEDs(long num)
{
        int i;
        for (i=0; i<6; ++i)
        {
                if ((num>>i) & 1
                        setLED(i+1);
                else
                        clrLED(i+1);
        }
}

/*  Pattern: "Count" from 0 to 63 in binary using LEDs to represent
bits of the current number. 1 = enabled LED, 0 = disabled LED. */
void pattern();
void pattern()
{
        long  i;
        int       j;
```

```
        for (i=0; i<=0b111111; ++i)
        {
                showNumberWithLEDs(i);
                wait(100000);
        }
}

void main(void)
{
  /*** Processor Expert internal initialization. DON'T REMOVE THIS
CODE!!! ***/
  PE_low_level_init();
  /*** End of Processor Expert internal initialization.            ***/

  /*Write your code here*/
#pragma  warn_possunwant off

  IRQA_On = IRQA_GetVal() ? 1 : 0;
  IRQB_On = IRQB_GetVal() ? 1 : 0;

  for(;;); {

        CLEARLEDS;
        pattern();

  }

#pragma  warn_possunwant reset
}

/* END LEDcontrol */
/*
** ###################################################################
/*
**     This file was created by UNIS Processor Expert 03.15 for
**     the Freescale DSP56x series of microcontrollers.
**
** ###############################################################
*/
```

7

C for DSP56800

This chapter explains the CodeWarrior™ compiler for DSP56800.

This chapter contains the following sections:

General Notes on C

Note the following on the DSP56800 processors:

- C++ language is not supported.
- Floating-point math functions (for example, sin, cos, and sqrt) are not supported.
- The sizeof function in C is not the same as the SIZEOF function in the linker. In C, the sizeof function returns a number of type SIZE_T, which the complier declares to be of type `unsigned long int`. The sizeof function in C returns the number of words, whereas the SIZEOF function in the linker returns the number of bytes.

Number Formats

This section explains how the CodeWarrior compilers implement integer and floating-point types for DSP56800 processors. Look at `limits.h` for more information on integer types and `float.h` for more information on floating-point types. Both `limits.h` and `float.h` are explained in the *MSL C Reference Manual*.

DSP56800 Integer Formats

Table 7.1 on page 138 shows the sizes and ranges of the data types for the DSP56800 compiler.

Table 7.1 Data Type Ranges

Type	Option Setting	Size (bits)	Range
bool	n/a	16	true or false
char	**Use Unsigned Chars** is disabled in the **C/C++ Language (C Only)** settings panel	16	-32,768 to 32,767
	Use Unsigned Chars is enabled	16	0 to 65,535
signed char	n/a	16	-32,768 to 32,767
unsigned char	n/a	16	0 to 65,535
short	n/a	16	-32,768 to 32,767
unsigned short	n/a	16	0 to 65,535
int	n/a	16	-32,768 to 32,767
unsigned int	n/a	16	0 to 65,535
long	n/a	32	-2,147,483,648 to 2,147,483,647
unsigned long	n/a	32	0 to 4,294,967,295

DSP56800 Floating-Point Formats

Table 7.2 on page 138 shows the sizes and ranges of the floating-point types for the DSP56800 compiler.

Table 7.2 DSP56800 Floating-Point Types

Type	Size (bits)	Range
float	32	1.17549e-38 to 3.40282e+38
short double	32	1.17549e-38 to 3.40282e+38

Table 7.2 DSP56800 Floating-Point Types (*continued*)

Type	Size (bits)	Range
`double`	32	`1.17549e-38` to `3.40282e+38`
`long double`	32	`1.17549e-38` to `3.40282e+38`

DSP56800 Fixed-Point Formats

Table 7.3 on page 139 shows the sizes and ranges of the fixed-point types for the DSP56800 compiler.

Table 7.3 DSP56800 Fixed-Point Types

Type	Declared As	Size (bits)	Range
fixed	`__fixed__`	16	`(-1.0 <= x < 1.0)`
short fixed	`__shortfixed__`	16	`(-1.0 <= x < 1.0)`
long fixed	`__longfixed__`	32	`(-1.0 <= x < 1.0)`

NOTE For compatibility reasons, preferably use DSP intrinsics instead of fixed-point types in Table 7.3 on page 139 for fractional arithmetic.

Calling Conventions, Stack Frames

The CodeWarrior IDE for Freescale DSP56800 stores data and calls functions in ways that might be different from other target platforms.

Calling Conventions

The registers `A`, `R2`, `R3`, `Y0`, and `Y1` pass parameters to functions. When a function is called, the parameter list is scanned from left to right. The parameters are passed in this way:

1. The first 32-bit value is placed in `A`.
2. The first two 16-bit values are placed in `Y0` and `Y1`, respectively.

3. The first two 16-bit addresses are placed in `R2` and `R3`.

   All remaining parameters are pushed onto the stack, beginning with the rightmost parameter. Multiple-word parameters have the least significant word pushed onto the stack first.

When calling a routine that returns a structure, the caller passes an address in `R0` which specifies where to copy the structure.

The registers `A`, `R0`, `R2`, and `Y0` are used to return function results as follows:

- 32-bit values are returned in `A`.
- 16-bit addresses are returned in `R2`.
- All 16-bit non-address values are returned in `Y0`.

Volatile and Non-Volatile Registers

Non-volatile Registers

Non-volatile registers are registers that can be saved across functions calls. These registers are also called saved over a call registers (SOCs).

Volatile Registers

Volatile registers are registers that cannot be saved across functions calls. These registers are also called non-SOC registers.

NOTE See Table 7.4 on page 140 for a list of volatile (non-SOC) and non-volatile (SOC) registers.

Table 7.4 Volatile and Non-Volatile Registers

Unit	Register Name	Size	Type	Comments
Arithmetic Logic Unit (ALU)	Y1	16	Volatile (non-SOC)	
	Y0	16	Volatile (non-SOC)	
	Y	32	Volatile (non-SOC)	

Table 7.4 Volatile and Non-Volatile Registers (*continued*)

Unit	Register Name	Size	Type	Comments
	X0	16	Volatile (non-SOC)	
	A2	4	Volatile (non-SOC)	
	A1	16	Volatile (non-SOC)	
	A0	16	Volatile (non-SOC)	
	A10	32	Volatile (non-SOC)	
	A	36	Volatile (non-SOC)	
	B2	4	Volatile (non-SOC)	
	B1	16	Volatile (non-SOC)	
	B0	16	Volatile (non-SOC)	
	B10	32	Volatile (non-SOC)	
	B	36	Volatile (non-SOC)	
Address Generation Unit (AGU)	R0	16	Volatile (non-SOC)	
	R1	16	Volatile (non-SOC)	
	R2	16	Volatile (non-SOC)	
	R3	16	Volatile (non-SOC)	

Table 7.4 Volatile and Non-Volatile Registers (*continued*)

Unit	Register Name	Size	Type	Comments
	N	16	Volatile (non-SOC)	
	SP	16	Volatile (non-SOC)	
	M01	16	Volatile (non-SOC)	In certain registers, values must be kept for proper C execution. Set to 0xFFFF for proper execution of C code.
Program Controller	PC	21	Volatile (non-SOC)	
	LA	16	Volatile (non-SOC)	
	HWS	16	Volatile (non-SOC)	
	OMR	16	Volatile (non-SOC)	In certain registers, values must be kept for proper C execution. For example, set the CM bit. (See “OMR Settings.”)
	SR	16	Volatile (non-SOC)	
	LC	16	Volatile (non-SOC)	
Page 0	MR0	16	Volatile (non-SOC)	
	MR1	16	Volatile (non-SOC)	
	MR2	16	Volatile (non-SOC)	

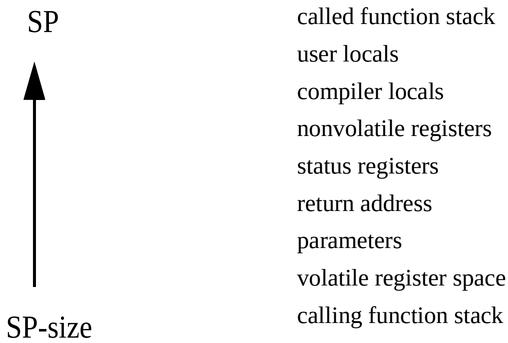
Table 7.4 Volatile and Non-Volatile Registers (*continued*)

Unit	Register Name	Size	Type	Comments
	MR3	16	Volatile (non-SOC)	
	MR4	16	Volatile (non-SOC)	
	MR5	16	Volatile (non-SOC)	
	MR6	16	Volatile (non-SOC)	
	MR7	16	Volatile (non-SOC)	
	MR8	16	Non-volatile (non-SOC)	
	MR9	16	Non-volatile (non-SOC)	
	MR10	16	Non-volatile (non-SOC)	
	MR11	16	Non-volatile (non-SOC)	
	MR12	16	Non-volatile (non-SOC)	
	MR13	16	Non-volatile (non-SOC)	
	MR14	16	Non-volatile (non-SOC)	
	MR15	16	Non-volatile (non-SOC)	

Stack Frame

The stack frame is generated as shown in Figure 7.1 on page 144. The stack grows upward, meaning that pushing data onto the stack increments the address in the stack pointer.

Figure 7.1 The Stack Frame



The stack pointer register (`SP`) is a 16-bit register used implicitly in all `PUSH` and `POP` instructions. The software stack supports structured programming, such as parameter passing to subroutines and local variables. If you are programming in both assembly-language and high-level language programming, use stack techniques. Note that it is possible to support passed parameters and local variables for a subroutine at the same time within the stack frame.

User Stack Allocation

The 56800 compilers build frames for hierarchies of function calls using the stack pointer register (SP) to locate the next available free X memory location in which to locate a function call's frame information. There is usually no explicit frame pointer register. Normally, the size of a frame is fixed at compile time. The total amount of stack space required for incoming arguments, local variables, function return information, register save locations (including those in pragma interrupt functions) is calculated and the stack frame is allocated at the beginning of a function call.

Sometimes, you may need to modify the SP at runtime to allocate temporary local storage using inline assembly calls. This invalidates all the stack frame offsets from the SP used to access local variables, arguments on the stack, etc. With the User Stack Allocation feature, you can use inline assembly instructions (with some restrictions) to modify the SP while maintaining accurate local variable, compiler temps, and argument offsets, i.e., these variables can still be accessed since the compiler knows you have modified the stack pointer.

The User Stack Allocation feature is enabled with the `#pragma check_inline_sp_effects [on|off|reset]` pragma setting. The pragma may be set on individual functions. By default the pragma is off at the beginning of compilation of each file in a project.

The User Stack Allocation feature allows you to simply add inline assembly modification of the SP anywhere in the function. The restrictions are straight-forward:

1. The SP must be modified by the same amount on all paths leading to a control flow merge point.
2. The SP must be modified by a literal constant amount. That is, address modes such as “(SP)+N” and direct writes to SP are not handled.
3. The SP must remain properly aligned.
4. You must not overwrite the compiler’s stack allocation by decreasing the SP into the compiler allocated stack space.

Point 1 above is required when you think about an if-then-else type statement. If one branch of a decision point modifies the SP one way and the other branch modifies SP another way, then the value of the SP is run-time dependent, and the compiler is unable to determine where stack-based variables are located at run-time. To prevent this from happening, the User Stack Allocation feature traverses the control flow graph, recording the inline assembly SP modifications through all program paths. It then checks all control flow merge points to make sure that the SP has been modified consistently in each branch converging on the merge point. If not, a warning is emitted citing the inconsistency.

Once the compiler determined that inline SP modifications are consistent in the control flow graph, the SP’s offsets used to reference local variables, function arguments, or temps are fixed up with knowledge of inline assembly modifications of the SP. Note, you may freely allocate local stack storage:

1. As long as it is equally modified along all branches leading to a control flow merge point.
2. The SP is properly aligned. The SP must be modified by an amount the compiler can determine at compile time.

A single new pragma is defined. `#pragma check_inline_sp_effects [on|off|reset]` will generate a warning if the user specifies an inline assembly instruction which modifies the SP by a run-time dependent amount. If the pragma is not specified, then stack offsets used to access stack-based variables will be incorrect. It is the user’s responsibility to enable `#pragma check_inline_sp_effects`, if they desire to modify the SP with inline assembly and access local stack-based variables. Note this pragma has no effect in function level assembly functions or separate assembly only source files (`.asm` files).

In general, inline assembly may be used to create arbitrary flow graphs and not all can be detected by the compiler.

For example:

```
REP #3
LEA (SP)+
```

This example would modify the SP by three, but the compiler would only see a modification of one. Other cases such as these might be created by the user using inline jumps or branches. These are dangerous constructs and are not detected by the compiler.

In cases where the SP is modified by a run-time dependent amount, a warning is issued.

Listing 7.1 Example 1 – Legal modification of SP Using Inline Assembly

```
#define EnterCritical() { asm(lea (SP)+);\
                          asm(move SR,X:(SP)+); \
                          asm(bfset  #0x0300,SR); \
                          asm(nop); \
                          asm(nop);}

#define ExitCritical()  { asm(lea (SP)-;\
                          asm(move X:SP) ,SR); \
                       asm(nop);\
                          asm(nop);}

#pragma check_inline_sp_effects on

int func()
{
         int a=1, b=1, c;

         EnterCritical();

         c = a+b;

         ExitCritical();

}
```

This case will work because there are no control flow merge points. SP is modified consistently along all paths from the beginning to the end of the function and is properly aligned.

Listing 7.2 Example 2 – Illegal Modification of SP using Inline Assembly

```
#define EnterCritical() { asm(lea (SP)+);\
                          asm(move SR,X:(SP)+); \
                          asm(bfset  #0x0300,SR); \
                          asm(nop); \
                          asm(nop);}

#define ExitCritical()  { asm(lea (SP)-;\
                          asm(move X:SP) ,SR); \
                       asm(nop);\
```

```
                                asm(nop);}

#pragma check_inline_sp_effects on

int func()
{
            int a=1, b=1, c;

            if (a)
{
                    EnterCritical();

                    c = a+b;

            }
            else {
                    c = b++;
            }

            ExitCritical();

            return (b+c);
}
```

This example will generate the following warning because the SP entering the ‘ExitCritical’ macro is different depending on which branch is taken in the if. Therefore, accesses to variables `a`, `b`, or `c` may not be correct.

```
Warning : Inconsistent inline assembly modification of SP in this
function.
M56800_main.c line 29    ExitCritical();
```

Listing 7.3 Example 3 – Modification of SP by a Run-time Dependent Amount

```
#define EnterCritical() { asm(move n,SP);\
                          asm(move SR,X:(SP)+); \
                          asm(nop); \
                          asm(nop);}

#define ExitCritical()  { asm(lea (SP)-;\
                          asm(move X:(SP) ,SR); \
                          asm(nop);\
                          asm(nop);}

#pragma check_inline_sp_effects on
```

```
int func()
{
          int a=1, b=1, c;

          if (a)
          {
                  EnterCritical();

                  c = a+b;

          }
          else {
                  EnterCritical();
                  c = b++;
          }



          return (b+c);
}
```

This example will generate the following warning:

```
Warning : Cannot determine SP modification value at compile time
M56800_main.c line 20     EnterCritical();
```

This example is not legal since the SP is modified by run-time dependent amount.

If all inline assembly modifications to the SP along all branches are equal approaching the exit of a function, it is not necessary to explicitly deallocate the increased stack space. The compiler “cleans up” the extra inline assembly stack allocation automatically at the end of the function.

Listing 7.4 Example 4 – Automatic Deallocation of Inline Assembly Stack Allocation

```
#pragma check_inline_sp_effects on
int func()
{
           int a=1, b=1, c;

           if (a)
           {
                     EnterCritical();

                     c = a+b;

           }
```

```
        else {
                EnterCritical();
                c = b++;
        }


        return (b+c);
}
```

This example does not need to call the 'ExitCritical' macro because the compiler will automatically clean up the extra inline assembly stack allocation.

Sections Generated by the Compiler

The compiler creates certain sections by default when compiling C source files. These default sections are all handled by the default LCF and are as follows:

- `.text`

  The compiler places executable code here by default.

- `.data`

  The compiler places initialized data here by default.

- `.bss`

  The compiler places uninitialized data here by default.

NOTE These sections are the sections generated by the compiler in the default case. Other user-defined sections can be generated through the use of the `#pragma define_section`.

If the project has the **Write constant data to .rodata section** checkbox enabled in the M56800 Processor portion of the Target Settings, then the compiler will generate the `.rodata` section for constant data. This option is overridden by the `#pragma use_rodata`.

NOTE The `.rodata` section is not handled by the default LCF. Thus, you need to add how you would like the LCF to place this section within the memory map. For more details on how to work with LCFs, see "ELF Linker."

By default, zero-initialized data is put into the `.bss` section by the compiler. This is done to reduce the load size of the application. The load size is reduced because instead of the debugger loading a sequence of zeros into the `.data` section (a loadable section), the compiler simply moves the zero-initialized data to the `.bss` section (not a loadable section) which is initialized to zero by the startup code. This behavior can be overridden by using the `#pragma explicit_zero_data` or by using the `#pragma use_rodata`, which put all constant data into a special `.rodata` section.

Table 7.5 on page 150 shows the memory map.

Table 7.5 Memory Map

Section	Size	Range (Hexadecimal)
`PROGRAM`	64K x 16 bit	`0000 - FFFF`
`DATA`	64K x 16 bit	`0000 - FFFF`

OMR Settings

The Operating Mode Register (OMR) is part of the program controller of the DSP56800 core. This register is responsible for the majority of how the core operates.

NOTE For general details about the OMR, see the *DSP56800 Family Manual*. For specific register details of your chip, see your chip manual.

The CodeWarrior compiler has some requirements about the value contained within this register and the mode in which the DSP56800 core operated. These requirements are described in Table 7.6 on page 150.

Table 7.6 OMR Bit Requirements

Bit Number	Bit Name	Requirements
`4`	Saturation or SA bit	This bit must be cleared for the compiled code to work properly.
`5`	Rounding or R bit	This bit must be cleared for the compiled code to work properly.
8	Condition code or CC bit	This bit must be set for the compiled code to work properly.

NOTE For general details about the OMR, see the *DSP56800 Family Manual*. For specific register details of your chip, see your chip manual.

Optimizing Code

Optimizations that are specific to DSP56800 development with the CodeWarrior IDE are:

- Page 0 Register Assignment on page 151
- Array Optimizations on page 151
- Multiply and Accumulate (MAC) Optimizations on page 152

Page 0 Register Assignment

The compiler uses page 0 address locations `X: 0x0030 - 0x003F` as *register variables*. Frequently accessed local variables are assigned to the page 0 *registers* instead of to stack locations so that load and store instructions are shortened. Addresses `X: 0x0030 - 0x0037` (page 0 registers `MR0-MR7`) are volatile registers and can be overwritten. The remaining registers (page 0 registers `MR8-MR15`) are treated as non-volatile and, if used by a routine, must be saved on entry and restored on exit.

Array Optimizations

Array indexing operations are optimized when optimizations are turned on in the **Global Optimizations** settings panel.

In Listing 7.5 on page 151, the `i` index is optimized out and the operation performs with address registers.

Listing 7.5 C Code Example for Array Optimizations

```
void main( void ) {
  short a[100], b[100];
  int i;

  // ... other code

  for ( i = 0; i < 100; i++ ) {
    ArrayA[i] = ArrayB[i]; }
  // ... other code
}
```

It is easier to understand the optimization process by viewing the assembler code mixed with C code, created both before (Listing 7.6 on page 152) and after (Listing 7.7 on page 152) optimizations are turned on.

Listing 7.6 Array Example Before Optimizations - Mixed View

```
for (i = 0;i < 100; i++ )
00001004: A7B20000  moves    #0,X:0x0032
00001006: A90B      bra      main+0x18 (0x1018)      ; 0x000812
 {
  a[i] = b[i];
00001007: 880F      move     SP,R0
00001008: DE40FF9D  lea      (R0+-99)
0000100A: BC32      moves    X:0x0032,N
0000100B: F044      move     X:(R0+N),X0
0000100C: 880F      move     SP,R0
0000100D: DE40FF39  lea      (R0+-199)
0000100F: BC32      moves    X:0x0032,N
00001010: D044      move     X0,X:(R0+N)
 }
```

The optimization level has been set to 3 (Listing 7.7 on page 152). Note that `i` is optimized out and the operation is now performed with address registers. This optimization is called induction.

NOTE With induction, the variable "i" is no longer used.

Listing 7.7 Array Example After Optimizations - Mixed View

```
for ( i = 0; i < 100; i++ )
00001008: A7B20000  moves    #0,X:0x0032
0000100A: A905      bra      START_+0x3 (0x101a)     ; 0x000810
 {
  a[i] = b[i];
0000100B: F016      move     X:(R2),X0
0000100C: D017      move     X0,X:(R3)
0000100D: DE02      lea      (R2)+
0000100E: DE03      lea      (R3)+
 }
```

Multiply and Accumulate (MAC) Optimizations

Multiply and Accumulate optimizations use address register calculations and perform arithmetic operations with a `MACR` instruction. The effect of these optimizations reflects in the source code examples in Listing 7.8 on page 153 and Listing 7.9 on page 153.

Listing 7.8 Sample Multiply and Accumulate Operation

```
void main( void )
{
  __fixed__ a[100], b[100];
  __fixed__ sum = 0;

  int i=0;

  for ( i = 0; i < 100; i++ ){
    sum += a[i] * b[i];
  }
}
```

The mixed view without optimizations is as follows:

Listing 7.9 Assembly Output for Multiply and Accumulate Operation

```
for ( i = 0; i < 100; i++ )
00001006: A7B20000  moves    #0,X:0x0032
00001008: A90E      bra      START_ (0x101f)         ; 0x000817
 {
  sum += a[i] * b[i];
00001009: 880F      move     SP,R0
0000100A: DE40FF39  lea      (R0+-199)
0000100C: BC32      moves    X:0x0032,N
0000100D: F344      move     X:(R0+N),Y1
0000100E: 880F      move     SP,R0
0000100F: DE40FF9D  lea      (R0+-99)
00001011: BC32      moves    X:0x0032,N
00001012: F144      move     X:(R0+N),Y0
00001013: B033      moves    X:0x0033,X0
00001014: 7C79      macr     +Y1,Y0,X0
00001015: 9033      moves    X0,X:0x0033
 }
```

The optimized version with level 3 optimizations (Listing 7.10 on page 153):

Listing 7.10 Assembly Output for Optimized Multiply and Accumulate Operation

```
for ( i = 0; i < 100; i++ )
0000100A: A7B20000  moves    #0,X:0x0032
0000100C: A908      bra      START_+0x5 (0x1021)     ; 0x000815
 {
  sum += a[i] * b[i];
0000100D: F316      move     X:(R2),Y1
0000100E: F117      move     X:(R3),Y0
```

```
0000100F: B033      moves    X:0x0033,X0
00001010: 7C79      macr     +Y1,Y0,X0
00001011: 9033      moves    X0,X:0x0033
00001012: DE02      lea      (R2)+
00001013: DE03      lea      (R3)+
 }
```

Compiler or Linker Interactions

This section explains important concepts about how the DSP56800 compiler and linker interact.

Deadstripping Unused Code and Data

The DSP56800 linker deadstrips unused code and data only from files compiled by the CodeWarrior C compiler. Assembler relocatable files and C object files built by other compilers are never deadstripped. Libraries built with the CodeWarrior C compiler only contribute the used objects to the linked program. If a library has assembly or other C compiler-built files, only those files that have at least one referenced object contribute to the linked program. Completely unreferenced object files are always ignored when deadstripping is enabled. Deadstripping is enabled by default in the **Linker > M56800 Linker Target Settings** panel.

Link Order

The DSP56800 linker always processes C and assembly source files, as well as archive files (`.a` and `.lib`) in the order specified under the **Link Order** tab in the project window. This is important in the case of symbol duplication. For example, if a symbol is defined in a source-code file and a library, the linker uses the definition which appears first in the link order.

If you want to change the link order, select the **Link Order** tab in the project window and drag your source or library file to the preferred location in the link order list. Files that appear at the top of the list are linked first.

8

Inline Assembly Language and Intrinsic Functions

This chapter explains the support for assembly language and intrinsic functions that is built into the CodeWarrior™ compiler. This chapter only covers the CodeWarrior IDE implementation of Freescale assembly language.

Working With DSP56800 Assembly Language

This section explains how to use the CodeWarrior compiler and assembler for assembly language programming, including assembly language syntax.

This chapter contains the following sections:

General Notes on Stand-Alone Assembly and Inline Assembly

The CodeWarrior IDE for the DSP56800 distinguishes between stand-alone assembly language and inline assembly language.

Stand alone assembly language files (files containing assembly language statements and having the file mapping suffix associated with the stand-alone assembler, usually `.asm`) are handled with an explicit stand-alone assembler plugin called the `asm_m56800.dll`. This plugin assembler supports a feature-rich assembly language syntax. The exact syntax of the assembly language statements and directives are found in the *DSP56800x_Assembly.pdf*.

Inline assembly language, on the other hand, is a DSP56800 instruction syntax handled directly by an internal compiler assembly language syntax parser and assembler. Inline assembly is normally distinguished by `asm { }` constructs within a C language function or as an explicit assembly language function in C, such as `asm int functionname()`. The inline assembler is meant for light duty enhancements or changes to instructions emitted by the compiler.

The following outlines a few of the key differences between stand-alone and inline assembly:

- Inline assembly statements are restricted to simple mnemonics and operand syntax as documented in the *DSP56800 Family* manual.
- Directives are not supported in inline assembly.
- Single and dual parallel move syntax is supported in both assemblers.
- Labels may be defined in inline assembly language, but their scope is restricted to the current function being compiled.
- Labels in the stand-alone assembler may be defined and exported (via the GLOBAL directive) in either X: or P: address space, therefore these labels are not scope limited.
- Data variables may not be defined in inline assembly language as the `ORG` directive is not supported in inline assembly (data requires `ORG X:` directive).
- Colons are required for any label definition in the inline assembler. The stand-alone assembler does not require a colon on labels as long as the label symbol name begins in the first character position.
- Mnemonics may begin at any character position on a line in the inline assembler. Mnemonics may not begin at the first character position in the stand-alone assembler.
- The stand-alone assembler allows semicolon comments. The inline assembler does not allow semicolon comments.

Inline Assembly Language Syntax for DSP56800

This section explains the inline assembly language syntax specific to DSP56800 development with the CodeWarrior IDE.

Function-level Inline Assembly Language

To specify that a block of code in your file should be interpreted as assembly language, use the `asm` keyword and standard DSP56800 instruction mnemonics.

To ensure that the C compiler recognizes the `asm` keyword, you must disable the **ANSI Keywords Only** option in the **C/C++ Language (C Only)** panel.

You can use the M56800 inline assembly language to specify that an *entire function* is in assembly language by using the syntax displayed in Listing 8.1 on page 157.

Listing 8.1 Function-level Syntax

```
asm <function header>
{
  <assembly instructions>
}
```

The function header is any valid C function header, and the local declarations are any valid C local declarations.

Statement-level Inline Assembly Language

The M56800 inline assembly language supports single assembly instructions as well as `asm` blocks, *within* a function using the syntax in Listing 8.2 on page 157. The inline assembly language statement is any valid assembly language statement.

Listing 8.2 Statement-level Syntax

```
asm { inline assembly statement
      inline assembly statement
      ...
}
asm ( inline assembly statement ;
      inline assembly statement ;
      ...
)
```

There are two different ways to represent statement-level assembly. In the first way, you use braces "{}" to contain the block. Within this type of block, the semicolon that separates statements is optional. In the second way, you use parentheses "()" to contain the block and the semicolon between statements is mandatory.

Adding Assembly Language to C Source Code

There are two ways to add assembly language statements in a C source code file. You can define a function with the `asm` qualifier, or you can use the inline assembly language.

The first method uses the `asm` keyword to specify that *all* statements in the function are in assembly language, as shown in Listing 8.3 on page 158 and Listing 8.7 on page 160. Note that if you are using this method, you must define local variables within the function.

Listing 8.3 Defining a Function with `asm`

```
asm long MyAsmFunction(void)
{
   /* Local variable definitions */
   /* Assembly language instructions */
}
```

The second method uses the `asm` qualifier as a statement to provide inline assembly language instructions, as shown in Listing 8.4 on page 158. Note that if you are using this method, you must *not* define local variables within the inline `asm` statement.

Listing 8.4 Inline Assembly with asm

```
long MyInlineAsmFunction(void)
{
    asm { move   x:(r0)+,x0 }
}
```

General Notes on Inline Assembly Language

Keep these points in mind as you write inline assembly language functions:

- All statements must either be a label:

  `[`*`LocalLabel`*`:]`

  Or an instruction:

  `( (`*`instruction`*`) [`*`operands`*`] )`

- Each statement must end with a new line
- Assembly language directives, instructions, and registers are not case-sensitive:

```
add   x0,y0
ADD   X0,Y0
```

Creating Labels for M56800 Inline Assembly

A label can be any identifier that you have not already declared as a local variable. A label must end with a colon.

Listing 8.5 Labels in M56800 Assembly

```
x1:   add  x0,y1,a
x2:   add  x0,y1,a
x3    add  x0,y1,a   //ERROR, MISSING COLON
```

Using Comments in M56800 Inline Assembly

Comments in inline assembly language can only be in the form of C and C++ comments. You cannot begin the inline assembly language comments with a semicolon (;) nor with a pound sign (#) - the preprocessor uses the pound sign. You can use the semicolon for comments in `.asm` sources. The proper comment format is shown in Listing 8.6 on page 159.

Listing 8.6 Comments Allowed in M56800 Inline Assembly Language

```
    move      x:(r3),y0       #  ERROR
    add       x0,y0           // OK
    move      r2,x:(sp)       ; ERROR
    adda      r0,r1,n         /* OK */
```

Calling Assembly Language Functions from C Code

You can call assembly language functions from C just like you would call any standard C function. You need to use standard C syntax for calling inline assembly language functions and stand-alone assembly language functions in `.asm` files.

Calling Inline Assembly Language Functions

You can call inline assembly language functions just like you would call any standard C function. Listing 8.7 on page 160 demonstrates how to create an inline assembly language function in a C source file. This example adds two 16-bit integers and returns the result.

Notice that you are passing two 16-bit addresses to the add_int function. You pick up those addresses in R3 and R2, and in Y0 pass back the result of the addition.

Listing 8.7 Sample Code - Creating an Inline Assembly Language Function

```
asm int add_int( int * i, int * j )
{
 move    x:(r2),y0
 move    x:(r3),x0
 add   x0,y0
 // int result returned in y0
 rts
}
```

Now you can call your inline assembly language function with standard C notation, as in Listing 8.8 on page 160.

Listing 8.8 Sample Code - Calling an Inline Assembly Language Function

```
  int x = 4, y = 2;

  y = add_int( &x, &y ); /* Returns 6 */
```

Calling Stand-alone Assembly Language Functions

In order for your assembly language files to be called from C code, you need to specify a `SECTION` mapping for your code so that it is linked appropriately. You must also specify a memory space location. Code is usually specified to program memory (P) space with the ORG directive.

When defining an assembly language function, use the GLOBAL directive to specify the list of symbols within the current section. You can then define the assembly language function.

An example of a complete assembly language function is shown in Listing 8.9 on page 160. In this function, two 16-bit integers are written to program memory. A separate function is needed to write to P: memory because C pointer variables cannot be employed. C pointer values only allow access to X: data memory.

The first parameter is a short value and the second parameter is the 16-bit address where the first parameter is written.

Listing 8.9 Sample Code - Creating an Assembly Language Function

```
                                  ;"my_assym.asm"
  SECTION user                    ;map to user defined section in CODE
```

```
  ORG P:                      ;put the following program in P
                                ;memory

  GLOBAL Fpmemwrite           ;This symbol is defined within the
                              ;current section and should be
                                 ;accessible by all sections

Fpmemwrite:
  MOVE    Y1,R0               ;Set up pointer to address
  NOP                            ;Pipeline delay for R0
  MOVE    Y0,P:(R0)+          ;Write 16-bit value to address
                              ;pointed to by R0 in P: memory and
                                ;post-increment R0
  rts                           ;return to calling function

  ENDSEC                        ;End of section
  END                           ;End of source program
```

NOTE	The compiler prepends the letter ‘F’ to every function label name. Therefore, when calling C functions from either Assembly Language or Inline Assembly, the ’F’ must be prepended.

You can now call your assembly language function from C, as shown in Listing 8.10 on page 161.

Listing 8.10 Sample Code - Calling an Assembly Language Function from C

```
void pmemwrite( short, short ); /* Write a value into P: memory */

void main( void )
{
  // ...other code

  // Write the value given in the first parameter to the address
  // of the second parameter in P: memory
  pmemwrite( (short)0xE9C8, (short)0x0010 );

  // other code...
}
```

Calling Functions from Assembly Language

Assembly programs can call C function or Assembly language functions. This section explains the compiler convention for:

- Calling C Functions from Assembly Language

  Functions written in C can be called from within assembly language instructions. For example, if you defined your C program function as:

```
void foot( void ) {
     /* Do something */
}
```

  You could then call your C function from assembly language as:

```
jsr   Ffoot
```

- Calling Assembly Language Functions from Assembly Language

  To call an assembly language function from assembly language, use the `jsr` instruction with the function name as defined in your assembly language source. For example, you can call your function in Listing 8.9 on page 160 as:

```
jsr   Fpmemwrite
```

Intrinsic Functions for DSP56800

This section explains issues related to DSP56800 intrinsic functions and using them with DSP56800 projects.

An Overview of Intrinsic Functions

CodeWarrior C for DSP56800 has intrinsic functions to generate inline assembly language instructions.

Intrinsic functions are used to target specific processor instructions. They can be helpful in accomplishing a few different things:

- Intrinsic functions let you pass in data to perform specific optimized computations. For example, some calculations may be inefficient if coded in C because the compiler has to follow ANSI C rules to represent data, and this may cause the program to jump to runtime math routines for certain computations. In such cases, it probably is better to code these calculations using assembly language instructions and intrinsic functions.
- Intrinsic functions can control small tasks. For example, with intrinsic functions you can set a bit in the operating mode register to enable saturation. This is more

convenient than using inline assembly language syntax and specifying the operation in an `asm` block, every time that the operation is required.

NOTE Support for intrinsic functions is not part of the ANSI C standard. They comprise an extension provided by the CodeWarrior compiler.

Fractional Arithmetic

Many of the intrinsic functions for Freescale DSP56800 use fractional arithmetic with implied fractional values. An implied fractional value is a symbol, which has been declared as an integer type, but is to be calculated as a fractional type. Data in a memory location or register can be interpreted as fractional or integer, depending on the needs of a user's program.

All intrinsic functions that generate multiply and divide instructions (DIV, MPY, MAC, MPYR, and MACR) perform fractional arithmetic on implied fractional values. The following equation shows the relationship between a 16-bit integer and a fractional value:

Fractional Value = Integer Value / (2^{15})

Similarly, the equation for converting a 32-bit integer to a fractional value is as follows:

Fractional Value = Long Integer Value / (2^{31})

Table 8.1 on page 163 shows how both 16 and 32-bit values can be interpreted as either fractional or integer values.

Table 8.1 Interpretation of 16- and 32-bit Values

Type	Hex	Integer Value	Fixed-point Value
short int	0x2000	8192	0.25
short int	0xE000	-8192	-0.25
long int	0x20000000	536870912	0.25
long int	0xE0000000	-536870912	-0.25

Macros Used with Intrinsics

These macros are used in intrinsic functions:

- Word16. A macro for signed short.
- Word32. A macro for signed long.

List of Intrinsic Functions: Definitions and Examples

The intrinsic functions supported by the DSP56800 are shown in Table 8.2 on page 164.

Table 8.2 Intrinsic Functions for DSP56800

Category	Function	Category	Function
Absolute/Negate on page 166	__abs on page 166	Multiplication/ MAC on page 179	__mac_r on page 180
	__negate on page 166		__msu_r on page 181
	_L_negate on page 167		__mult on page 181
Addition/ Subtraction on page 167	__add on page 167		__mult_r on page 182
	__sub on page 168		_L_mac on page 183
	_L_add on page 169		_L_msu on page 184
	_L_sub on page 169		_L_mult on page 184
Control on page 170	__stop on page 170		_L_mult_ls on page 185

Table 8.2 Intrinsic Functions for DSP56800 (*continued*)

Conversion on page 170	__fixed2int on page 171	Normalization on page 185	__norm_l on page 186
	__fixed2long on page 171		__norm_s on page 186
	__fixed2short on page 172	Rounding on page 187	__round on page 187
	__int2fixed on page 172	Shifting on page 188	__shl on page 188
	__labs on page 173		__shr on page 189
	__long2fixed on page 174		__shr_r on page 190
	__short2fixed on page 174		_L_shl on page 190
Copy on page 174	__memcpy on page 175		_L_shr on page 191
	__strcpy on page 175		_L_shr_r on page 192
Deposit/ Extract on page 176	__extract_h on page 176		
	__extract_l on page 177		
	_L_deposit_h on page 177		
	_L_deposit_l on page 178		
Division on page 178	__div on page 178		
	__div_ls on page 179		

Absolute/Negate

__abs

Definition

Computes and returns the absolute value of a 16-bit integer. Generates an `ABS` instruction.

Assumption

Prototype

```
int __abs( int );
```

Example

```
int i = -2;
i = __abs( i );
```

__negate

Definition

Negates a 16-bit integer or fractional value returning a 16-bit result. Returns 0x7FFF for an input of 0x8000.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 __negate(Word16 svar1)
```

Example

```
int result, s1 = 0xE000;/*  - 0.25 */
result = __negate(s1);
// Expected value of result: 0x2000 = 0.25
```

_L_negate

Definition

Negates a 32-bit integer or fractional value returning a 32-bit result. Returns 0x7FFFFFFF for an input of 0x80000000.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 _L_negate(Word32 lvar1)
```

Example

```
long result, s1 = 0xE0000000;  /*  - 0.25 */
result = _L_negate(s1);
// Expected value of result: 0x20000000 = 0.25
```

Addition/Subtraction

__add

Definition

Addition of two 16-bit integer or fractional values, returning a 16-bit result.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 __add(Word16 src_dst, Word16 src2)
```

Example

```
short s1 = 0x4000;/* 0.5  */
short s2 = 0x2000;/* 0.25 */
short result;

result = __add(s1,s2);
// Expected value of result: 0x6000 = 0.75
```

__sub

Definition

Subtraction of two 16-bit integer or fractional values, returning a 16-bit result.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 __sub(Word16 src_dst, Word16 src2)
```

Example

```
short s1 = 0x4000;/* 0.5   */
short s2 = 0xE000;/* -0.25 */
short result;

result = __sub(s1,s2);
// Expected value of result: 0x6000 = 0.75
```

_L_add

Definition

Addition of two 32-bit integer or fractional values, returning a 32-bit result.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 _L_add(Word32 src_dst, Word32 src2)
```

Example

```
long la = 0x40000000;/* 0.5  */
long lb = 0x20000000;/* 0.25 */
long result;

result = _L_add(la,lb);
// Expected value of result: 0x60000000 = 0.75
```

_L_sub

Definition

Subtraction of two 32-bit integer or fractional values, returning a 32-bit result.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 _L_sub(Word32 src_dst, Word32 src2)
```

Example

```
long la = 0x40000000;/* 0.5   */
long lb = 0xE0000000;/* -0.25 */
long result;

result = _L_sub(la,lb);
// Expected value of result: 0x60000000 = 0.75
```

Control

__stop on page 170

__stop

Definition

Generates a STOP instruction which places the processor in the low power STOP mode.

Prototype

```
void  __stop(void)
```

Usage

```
__stop();
```

Conversion

The following intrinsics are provided to convert between various integer and fixed point types. The appropriate intrinsic should always be used when referencing an integer constant in fixed point context (i.e., assignment and comparisons).

- __fixed2int on page 171
- __fixed2long on page 171
- __fixed2short on page 172
- __int2fixed on page 172
- __labs on page 173
- __long2fixed on page 174
- __short2fixed on page 174

__fixed2int

Definition

Converts a 16-bit __fixed__ value to a 16-bit integer.

Prototype

```
int  __fixed2int ( __fixed__ );
```

Example

```
int  i;
int  j;
__fixed__   i_fix = 0.645;

i = __fixed2int(  i_fix ); /* Returns 21135 */
j = __fixed2int(  0.645 );

if (i == j)
    printf("PASSED\n");

if (i == __fixed2int(  0.645 ))
    printf("PASSED\n");

if (j == 21135)
    printf("PASSED\n");
```

__fixed2long

Definition

Converts a 32-bit __longfixed__ value to a 32-bit long integer.

Prototype

```
long __fixed2long ( __longfixed__ );
```

Example

```
long l;
__longfixed__  lfix = 0.645;

l = __fixed2long( lfix ); /* Returns 1385126952 */
```

__fixed2short

Definition

Converts a 16-bit __shortfixed__ value to a 16-bit short integer.

Prototype

```
short __fixed2short ( __shortfixed__ );
```

Example

```
short  s;
__shortfixed__  sfix = 0.645;

s = __fixed2short( sfix ); /* Returns 21135 */
```

__int2fixed

Definition

Converts a 16-bit integer value to a 16-bit __fixed__ value.

Prototype

```
__fixed__  __int2fixed ( int );
```

Example

```
int i = 0x2000;
__fixed__ ifix;
__fixed__ jfix;

/* Returns 0.25*/
ifix = __int2fixed( i );
jfix = __int2fixed( 0x2000 );

if (ifix == jfix)
    printf("PASSED\n");

if (ifix == __int2fixed( 0x2000 ))
    printf("PASSED\n");

if (jfix == 0.25)
    printf("PASSED\n");
```

__labs

Definition

Computes and returns the absolute value of a 32-bit long integer. Generates an `ABS` instruction.

Prototype

```
long __labs ( long );
```

Example

```
long l = -2;
l = __labs( l );        /* Returns 2 */
```

__long2fixed

Definition

Converts a 32-bit long integer to a 32-bit __longfixed__ type.

Prototype

```
__longfixed__      __long2fixed ( long );
```

Example

```
long l = 2;
__longfixed__ lfix;
/* Returns 9.31e-10  (2^-30)*/
lfix = __long2fixed( l );
```

__short2fixed

Definition

Converts a 16-bit short integer to a 16-bit __shortfixed__ type.

Prototype

```
__shortfixed__    __short2fixed ( short );
```

Example

```
short  s = 2;
__shortfixed__  sfix;

/* Returns 0.0000610 (2^-14)*/
sfix = __short2fixed( s );
```

Copy

- __memcpy on page 175
- __strcpy on page 175

__memcpy

Definition

Copy a contiguous block of memory of n characters from the item pointed to by `source` to the item pointed to by `dest`. The behavior of `__memcpy( )` is undefined if the areas pointed to by `dest` and `source` overlap.

Prototype

```
void * __memcpy ( void *dest,
            const void *source,
            size_t n );
```

Example

```
const int len =  9;
char a1[len] = "Socrates\0";
char a2[len] = null;

/* Now copy contents of a1 to a2 */
__memcpy( (char *)a2, (char *)a1, len );
```

__strcpy

Definition

Copies the character array pointed to by `source` to the character array pointed to by `dest`. The `source` argument must be a constant string. The function will not be inlined if `source` is defined outside of the function call. The resulting character array at `dest` is null terminated as well.

Prototype

```
char * __strcpy  ( char *dest,
                   const char *source );
```

Example

```
char d[11];

__strcpy( d, "Metrowerks\0" );
/* d array now contains the string "Metrowerks" */
```

Deposit/ Extract

- __extract_h on page 176
- __extract_l on page 177
- _L_deposit_h on page 177
- _L_deposit_l on page 178

__extract_h

Definition

Extracts the 16 MSBs of a 32-bit integer or fractional value. Returns a 16-bit value. Does not perform saturation. When an accumulator is the destination, zeroes out the LSP portion. Corresponds to "truncation" when applied to fractional values.

Prototype

```
Word16 __extract_h(Word32 lsrc)
```

Example

```
long l = 0x87654321;
short result;

result = __extract_h(l);
// Expected value of result: 0x8765
```

__extract_l

Definition

Extracts the 16 LSBs of a 32-bit integer or fractional value. Returns a 16-bit value. Does not perform saturation. When an accumulator is the destination, zeroes out the LSP portion.

Prototype

```
Word16 __extract_l(Word32 lsrc)
```

Example

```
long l = 0x87654321;
short result;

result = __extract_l(l);
// Expected value of result: 0x4321
```

_L_deposit_h

Definition

Deposits the 16-bit integer or fractional value into the upper 16 bits of a 32-bit value, and zeroes out the lower 16 bits of a 32-bit value.

Prototype

```
Word32 _L_deposit_h(Word16 ssrc)
```

Example

```
short s1 = 0x3FFF;
long result;

result = _L_deposit_h(s1);
// Expected value of result: 0x3fff0000
```

_L_deposit_l

Definition

Deposits the 16-bit integer or fractional value into the lower 16 bits of a 32- bit value, and sign extends the upper 16 bits of a 32-bit value.

Prototype

```
Word32 _L_deposit_l(Word16 ssrc)
```

Example

```
short s1 = 0x7FFF;
long result;

result = _L_deposit_l(s1);
// Expected value of result: 0x00007FFF
```

Division

- __div on page 178
- __div_ls on page 179

__div

Definition

Divides two 16-bit short integers as a fractional operation and returns the result as a 16-bit short integer. Generates a `DIV` instruction.

Prototype

```
short __div( short, short );
```

Example

```
short i = 0x2000; /* Assign 0.25 to i */
short j = 0x4000; /* Assign 0.50 to j */
__fixed__ f;

i = __div( i, j );       /* Returns 16384 */
f = __short2fixed( i ); /* Returns 0.50 */
```

__div_ls

Definition

Single quadrant division, that is, both operands are positive two 16-bit fractional values, returning a 16-bit result. If both operands are equal, returns 0x7FFF (occurs naturally).

Note

Does not check for division overflow cases.

Does not check for divide by zero cases.

Prototype

```
Word16 __div_s(Word16 s_denominator, Word16 s_numerator)
```

Example

```
short s1=0x2000;/*  0.25 */
short s2=0x4000;/*  0.5  */
short result;

result = __div_s(s2,s1);
// Expected value of result: 0.25/0.5 = 0.5 = 0x4000
```

Multiplication/ MAC

- __mac_r on page 180
- __msu_r on page 181
- __mult on page 181

__mac_r

Definition

Multiply two 16-bit fractional values and add to 32-bit fractional value. Round into a 16-bit result, saturating if necessary. When an accumulator is the destination, zeroes out the LSP portion.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

OMR's R bit was set to 1 at least 3 cycles before this code, that is, 2's complement rounding, not convergent rounding.

Prototype

```
Word16 __mac_r(Word32 laccum,  Word16 sinp1, Word16 sinp2)
```

Example

```
short s1 = 0xC000;/* - 0.5 */
short s2 = 0x4000;/*   0.5 */
short result;
long Acc = 0x0000FFFF;

result = __mac_r(Acc,s1,s2);
// Expected value of result: 0xE001
```

__msu_r

Definition

Multiply two 16-bit fractional values and subtract this product from a 32-bit fractional value. Round into a 16-bit result, saturating if necessary. When an accumulator is the destination, zeroes out the LSP portion.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

OMR's R bit was set to 1 at least 3 cycles before this code, that is, 2's complement rounding, not convergent rounding.

Prototype

```
Word16 __msu_r(Word32 laccum,  Word16 sinp1, Word16 sinp2)
```

Example

```
short s1 = 0xC000;/* - 0.5 */
short s2 = 0x4000;/*   0.5 */
short result;
long Acc = 0x20000000;

result = __msu_r(Acc,s1,s2);
// Expected value of result: 0x4000
```

__mult

Definition

Multiply two 16-bit fractional values and truncate into a 16-bit fractional result. Saturates only for the case of 0x8000 x 0x8000. When an accumulator is the destination, zeroes out the LSP portion.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 __mult(Word16 sinp1, Word16 sinp2)
```

Example

```
short s1 = 0x2000;/*  0.25 */
short s2 = 0x2000;/*  0.25 */
short result;

result = __mult(s1,s2);
// Expected value of result: 0.625 = 0x0800
```

__mult_r

Definition

Multiply two 16-bit fractional values, round into a 16-bit fractional result. Saturates only for the case of 0x8000 x 0x8000. When an accumulator is the destination, zeroes out the LSP portion.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

OMR's R bit was set to 1 at least 3 cycles before this code, that is, 2's complement rounding, not convergent rounding.

Prototype

```
Word16 __mult_r(Word16 sinp1, Word16 sinp2)
```

Example

```
short s1 = 0x2000;/*  0.25 */
short s2 = 0x2000;/*  0.25 */
short result;

result = __mult_r(s1,s2);
// Expected value of result: 0.0625 = 0x0800
```

_L_mac

Definition

Multiply two 16-bit fractional values and add to 32-bit fractional value, generating a 32-bit result, saturating if necessary.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 _L_mac(Word32 laccum, Word16 sinp1, Word16 sinp2)
```

Example

```
short s1 = 0xC000;/* - 0.5 */
short s2 = 0x4000;/*   0.5 */
long result, Acc = 0x20000000;/*  0.25 */

result = _L_mac(Acc,s1,s2);
// Expected value of result: 0
```

_L_msu

Definition

Multiply two 16-bit fractional values and subtract this product from a 32-bit fractional value, saturating if necessary. Generates a 32-bit result.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 _L_msu(Word32 laccum, Word16 sinp1, Word16 sinp2)
```

Example

```
short s1 = 0xC000;/* - 0.5 */
short s2 = 0xC000;/* - 0.5 */
long result, Acc = 0;

result = _L_msu(Acc,s1,s2);
// Expected value of result: 0.25
```

_L_mult

Definition

Multiply two 16-bit fractional values generating a signed 32-bit fractional result. Saturates only for the case of 0x8000 x 0x8000.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 _L_mult(Word16 sinp1, Word16 sinp2)
```

Example

```
short s1 = 0x2000;/*  0.25 */
short s2 = 0x2000;/*  0.25 */
long result;

result = _L_mult(s1,s2);
// Expected value of result: 0.0625 = 0x08000000
```

_L_mult_ls

Definition

Multiply one 32-bit and one-16-bit fractional value, generating a signed 32-bit fractional result. Saturates only for the case of 0x80000000 x 0x8000.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 _L_mult_ls(Word32 linp1, Word16 sinp2)
```

Example

```
long l1 = 0x20000000;/* 0.25 */
short s2 = 0x2000;/* 0.25 */
long result;

result = _L_mult_ls(l1,s2);
// Expected value of result: 0.0625 = 0x08000000
```

Normalization

• __norm_l on page 186

• __norm_s on page 186

__norm_l

Definition

Computes the number of left shifts required to normalize a 32-bit value, returning a 16-bit result. Returns a shift count of 0 for an input of 0x00000000.

Note

Does not actually normalize the value!

This operation is NOT optimal on the DSP56800 because of the case of returning 0 for an input of 0x00000000.

Prototype

```
Word16 __norm_l(Word32 lsrc)
```

Example

```
long ll = 0x20000000;/* .25 */
short result;

result = __norm_l(ll);
// Expected value of result: 1
```

__norm_s

Definition

Computes the number of left shifts required to normalize a 16-bit value, returning a 16-bit result. Returns a shift count of 0 for an input of 0x0000.

Note

Does not actually normalize the value!

This operation is NOT optimal on the DSP56800 because of the case of returning 0 for an input of 0x0000. See the intrinsic __norm_l on page 186 which is more optimal but generates a different value for the case where the input == 0x0000.

Prototype

```
Word16 __norm_s(Word16 ssrc)
```

Example

```
short s1 = 0x2000;/* .25 */
short result;

result = __norm_s(s1);
// Expected value of result: 1
```

Rounding

__round on page 187

__round

Definition

Rounds a 32-bit fractional value into a 16-bit result. When an accumulator is the destination, zeroes out the LSP portion.

Assumptions

OMR’s R bit was set to 1 at least 3 cycles before this code, that is, 2’s complement rounding, not convergent rounding.

OMR’s SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 __round(Word32 lvar1)
```

Example

```
long l = 0x12348002;/*if low 16 bits = 0xFFFF > 0x8000 then
```

```
     add 1 */
short result;

result = __round(l);
// Expected value of result: 0x1235
```

Shifting

__shl

Definition

Arithmetic shift of 16-bit value by a specified shift amount. If the shift count is positive, a left shift is performed. Otherwise, a right shift is performed. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

Note

This operation is not optimal on the DSP56800 because of the saturation requirements and the bidirectional capability.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 __shl(Word16 sval2shft, Word16 s_shftamount)
```

Example

```
short result;
short s1 = 0x1234;
short s2= 1;

result = __shl(s1,s2);
// Expected value of result: 0x2468
```

__shr

Definition

Arithmetic shift of 16-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

Note

This operation is not optimal on the DSP56800 because of the saturation requirements and the bidirectional capability.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 __shr(Word16 sval2shft, Word16 s_shftamount)
```

Example

```
short result;
short s1 = 0x2468;
short s2= 1;

result = __shr(s1,s2);
// Expected value of result: 0x1234
```

__shr_r

Definition

Arithmetic shift of 16-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. If a right shift is performed, then rounding performed on result. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

Note

This operation is not optimal on the DSP56800 because of the saturation requirements and the bidirectional capability.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 __shr_r(Word16 s_val2shft, Word16 s_shftamount)
```

Example

```
short result;
short s1 = 0x2468;
short s2= 1;

result = __shr(s1,s2);
// Expected value of result: 0x1234
```

_L_shl

Definition

Arithmetic shift of 32-bit value by a specified shift amount. If the shift count is positive, a left shift is performed. Otherwise, a right shift is performed. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

Note

This operation is not optimal on the DSP56800 because of the saturation requirements and the bidirectional capability. See the intrinsic _L_shl on page 190 or result = shlfts(l, s1); on page 191 which are more optimal.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 _L_shl(Word32 lval2shft, Word16 s_shftamount)
```

Example

```
long result, l = 0x12345678;
short s2= 1;

result = _L_shl(l,s2);
// Expected value of result: 0x2468ACF0
result = shlfts(l, s1);
// Expected value of result: 0x91A259E0
```

_L_shr

Definition

Arithmetic shift of 32-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

Note

This operation is not optimal on the DSP56800 because of the saturation requirements and the bidirectional capability.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 _L_shr(Word32 lval2shft, Word16 s_shftamount)
```

Example

```
long result, l = 0x24680000;
short s2= 1;

result = _L_shr(l,s2);
// Expected value of result: 0x12340000
```

_L_shr_r

Definition

Arithmetic shift of 32-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. If a right shift is performed, then rounding performed on result. Saturation may occur during a left shift.

Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 _L_shr_r(Word32 lval2shft, Word16 s_shftamount)
```

Example

```
long l1 = 0x41111111;
short s2 = 1;
long result;

result = _L_shr_r(l1,s2);
// Expected value of result: 0x20888889
```

Pipeline Restrictions

This section gives an overview of how the pipeline restrictions are handled by the DSP56800 compiler.

The following list contains pipeline restrictions that are detected and handled. If any of these cases are detected by the compiler's inline assembler, the compiler generates a warning and inserts a NOP instruction to correct the violation of the pipeline restriction.

1. A NORM instruction cannot be immediately followed by an instruction that accesses X memory using the R0 pointer. The following example shows a warning is generated:

```
NORM    R0,A
MOVE    X:(R0)+,A      ;Cannot reference R0 after NORM
```

2. Any jump, branch, or branch on bit field may not specify the instruction at LA or LA-1 of a hardware DO loop as their target addresses.

```
DO #7,LABEL
BCC LABEL ;Cannot branch to LA
;instruction
LABEL:
```

3. Any jump, branch, or branch on bit field instructions may not be located in the last two locations of a hardware DO loop (that is, at LA or at LA-1).

```
DO #7,LABEL
   BCC ULABEL ;Cannot branch in LA
   ;instruction
LABEL:
```

NOTE A warning will be emitted when pipeline conflicts are detected.

4. If a MOVE instruction changes the value in one of the address registers (R0–R3), then the contents of the register are not available for use until the second following instruction. That is, the instruction immediately following the MOVE instruction does not use the modified register to access X memory or update an address. This also applies to the SP register and M01 register.

```
MOVE    X:(SP-2),R1
MOVE    X:(R1)+,A;    ; R1 is not available
```

In addition, it applies if a 16-bit immediate value is moved to the N register, and the option for **Compiler adjusts for delayed load of N register** in the M56800 Processor target settings panel is enabled.

```
MOVE     #3,n
MOVE     X:(SP+N),Y0    ; N is not available
```

5. If a bit-field instruction changes the value in one of the address registers (R0–R3), then the contents of the register are not available for use until the second following instruction. That is, the instruction immediately following the MOVE instruction does not use the modified register to access X memory or update an address. This applies to the SP and M01 registers.

```
BFCLR    #1,R1
MOVE     X:(R1)+,A;    ; R1 is not available
```

In addition, it applies to the N register when the **Compiler adjusts for delayed load of N register** option in the M56800 Processor target settings panel is enabled.

```
BFCLR    #1,N
MOVE     X:(R0+N),Y0    ;N is not available
```

6. For the case of nested hardware DO loops, it is required that there be at least two instructions after the pop of the LA and LC registers before the instruction at the last address of the outer loop.

```
DO #3,OLABEL ; Beginning of outer loop
PUSH LC
PUSH LA
DO X0,ILABEL ; Beginning of inner loop
; (instructions)
REP Y0 ; Skips ASL if y0 = 0
ASL A
; (instructions)
ILABEL:  ; End of inner loop
      POP LA
      POP LC
      NOP; 3 instructions required after POP
      NOP; 3 instructions required after POP
      NOP; 3 instructions required after POP
OLABEL: ; End of outer loop
```

7. If the CLR instruction changes the value in one of the address registers (R0-R3), then the contents of the register are not available for use until the second following instruction. That is, the instruction immediately following the CLR instruction does not use the modified register to acccess X memory or update an address. This also applies to the SP register and the M01 register.

```
CLR R0
MOVE X:(R0)+,A;Cannot reference R0 after NORM
```

In addition, it applies if the 16-bit immediate value is moved to the N register and the option for **Compiler adjusts for delayed load of N register** in the M56800 Processor target settings panel is enabled.

```
clr N
MOVE X:(SP)+N,Y0 ;N is not available
```

9

Debugging for DSP56800

This chapter, which explains the generic features of the CodeWarrior™ debugger, consists of these sections:

Using Remote Connections

Remote connections are settings that describe how the CodeWarrior IDE should connect to and control program execution on target boards or systems, such as the debugger protocol, connection type, and connection parameters the IDE should use when it connects to the target system. This section shows you how to access remote connections in the CodeWarrior IDE, and describes the various debugger protocols and connection types the IDE supports.

NOTE We have included several types of remote connections in the default CodeWarrior installation. You can modify these default remote connections to suit your particular needs.

TIP When you import a Makefile into the CodeWarrior IDE to create a CodeWarrior project, the IDE asks you to specify the type of debugger interface (remote connection) you want to use. To debug the generated CodeWarrior project, you must properly configure the remote connection you selected when you created the project.

Accessing Remote Connections

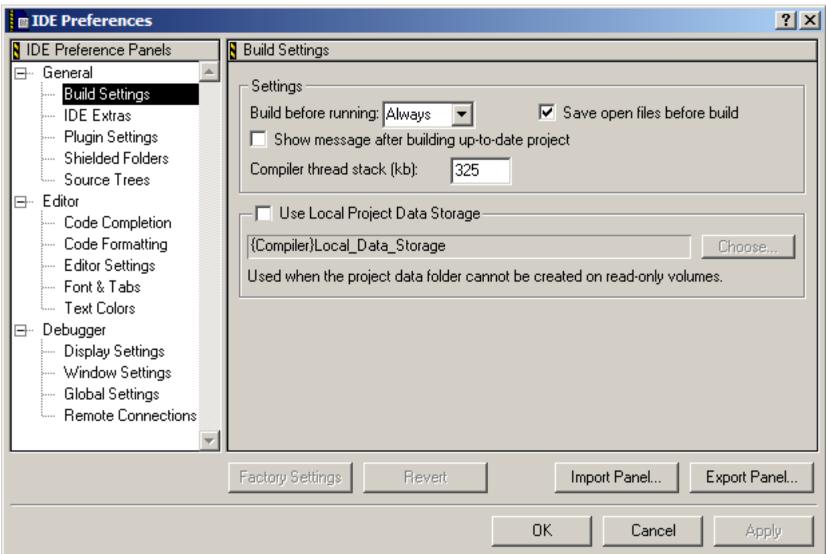
You access remote connections in the CodeWarrior **IDE Preferences** window. Remote connections listed in the preferences window are available for use in all CodeWarrior projects and build targets.

To access remote connections:

1. From the CodeWarrior menu bar, select **Edit** > **Preferences**.

   The **IDE Preferences** window (Figure 9.1 on page 198) appears.

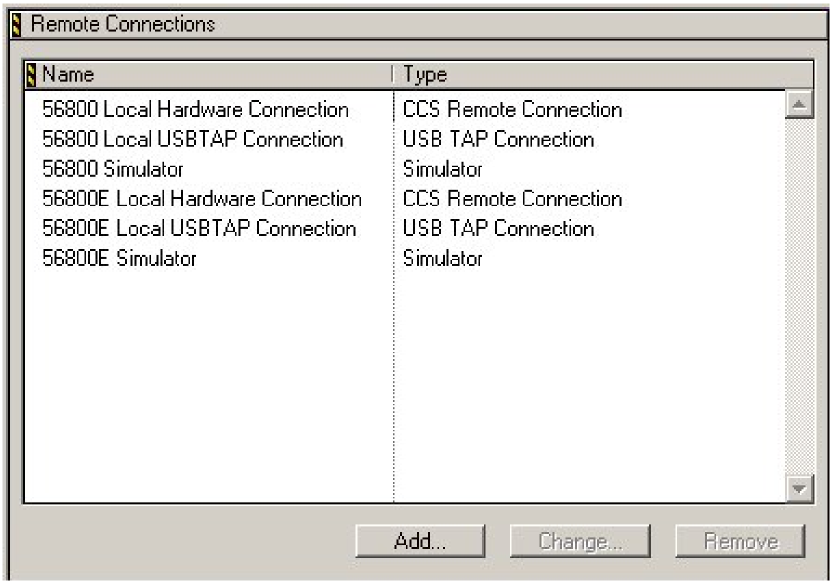
Figure 9.1 IDE Preferences Window



2. In the **IDE Preference Panels** list, select **Remote Connections**.

   The **Remote Connections** preference panel (Figure 9.2 on page 199) appears.

Figure 9.2 Remote Connections Preference Panel



NOTE The specific remote connections that appear in the Remote Connections list differ between CodeWarrior products and hosts.

The **Remote Connections** preference panel lists all of the remote connections of which the CodeWarrior IDE is aware. You use this preference panel to add your own remote connections, remove remote connections, and configure existing remote connections to suit your needs.

To add a new remote connection, click **Add**.

To configure an existing remote connection, select it and click **Change**.

To remove an existing remote connection, select it and click **Remove**.

TIP To specify a remote connection for a particular build target in a CodeWarrior project, you select the remote connection from the **Connection** list box in the **Remote Debugging** target settings panel. For an overview of the **Remote Debugging** settings panel, see the *CodeWarrior IDE User's Guide*.

Understanding Remote Connections

Every remote connection specifies a debugger protocol and a connection type.

A *debugger protocol* is the protocol the IDE uses to debug the target system. This setting generally relates specifically to the particular device you use to physically connect to the target system.

A *connection type* is the type of connection (such as CCS, USBTAP, or Simulator) the CodeWarrior IDE uses to communicate with and control the target system.

Table 9.1 on page 200 describes each of the supported debugger protocols.

Table 9.1 Debugger Protocols

Debugger Protocol	Description
CCS 56800 Protocol Plugin	Select to use a CCS hardware target system.
56800 Simulator	Select to use the Simulator on the host computer.

Each of these protocols supports one or more types of connections (CCS, USBTAP, and Simulator). “Editing Remote Connections” on page 200 describes each supported connection type and how to configure them.

Editing Remote Connections

Based on the specified debugger protocol and connection type, the IDE makes different settings available to you. For example, if you specify a Serial connection type, the IDE presents settings for baud rate, stop bits, flow control, and so on. Table 9.2 on page 200 describes the supported connection types for each debugger protocol.

Table 9.2 Supported Connection Types

Debugger Protocol	Supported Connection Types
CCS 56800 Protocol Plugin	CCS Remote Connection on page 201, USBTAP on page 203
56800 Simulator	on page 205Simulator on page 205

To configure a remote connection to correspond to your particular setup, you must edit the connection settings. You access the settings with the **Edit Connection** dialog box. You can view this dialog box in one of these ways:

- In the **Remote Connections** IDE preference panel, select a connection from the list, and click **Edit**. The **Edit Connection** dialog box appears.

- In the **Remote Connections** IDE preference panel, click **Add** to create a new remote connection. The **New Connection** dialog box appears.
- In the **Remote Debugging** target settings panel, select a connection from the **Connection** list box, then click the **Edit Connection** button. The **Edit Connection** dialog box appears.

This section describes the settings for each connection type:

- CCS Remote Connection on page 201
- USBTAP on page 203
- Simulator on page 205

CCS Remote Connection

Use this connection type to configure how the IDE uses the Command Converter Server (CCS) protocol to connect with the target system. This connection type is available only when the **CCS 56800 Protocol Plugin** debugger protocol is selected.

Figure 9.3 on page 202 shows the settings that are available to you when you select **CCS Remote Connection** from the **Connection Type** list box in the **Edit Connection** dialog box.

Figure 9.3 CCS Remote Connection Settings

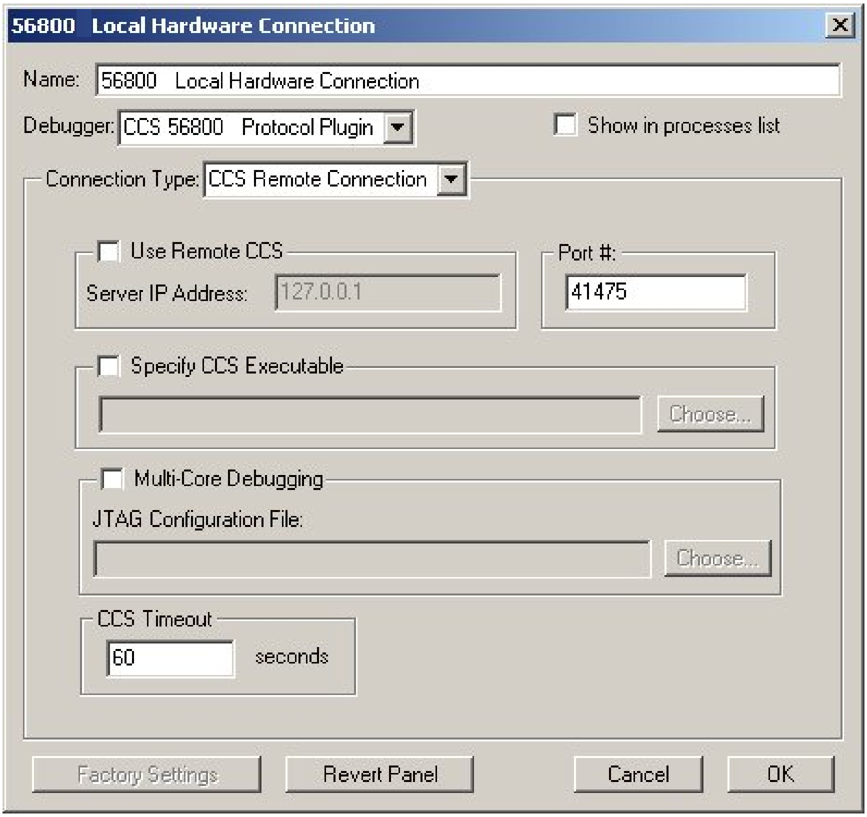


Table 9.3 on page 202 describes the options in this dialog box.

Table 9.3 CCS Remote Connection Options

Option	Description
Name	Enter the name you want to use to refer to this remote connection within the CodeWarrior IDE.
Debugger	Select **CCS 56800 Protocol Plugin**.
Connection Type	Select **CCS Remote Connection**.
Use Remote CCS	Check to debug code on a target system when the system already has CCS running and connected.

Table 9.3 CCS Remote Connection Options (*continued*)

Option	Description
Server IP Address	Enter the Internet Protocol (IP) address assigned to the target system.
Port #	Enter the port number on the target system to which the IDE should connect for CCS operations. The default port number for CCS hardware connections is 41475. Enter 41476 for the CCS Simulator.
Specify CCS Executable	Check to use another CCS executable file rather than the default CCS executable file: `CWInstall\ccs\bin\ccs.exe`
Multi-Core Debugging	Check to debug code on a target system with multiple cores where you need to specify the JTAG chain for debugging. Click **Choose** to specify the JTAG initialization file. A JTAG initialization file contains the names and order of the boards / cores you want to debug.
CCS Timeout	Enter the duration (in seconds) after which the CCS should attempt to reconnect to the target system if a connection attempt fails.

USBTAP

Use this connection type to configure how the IDE uses CodeWarrior USB TAP device to connect with the target system. This connection type is available only when the **CCS 56800 Protocol Plugin** debugger protocol is selected.

Figure 9.4 on page 204 shows the settings that are available to you when you select **USBTAP** from the **Connection Type** list box in the **Edit Connection** dialog box.

Figure 9.4 USBTAP Connection Settings

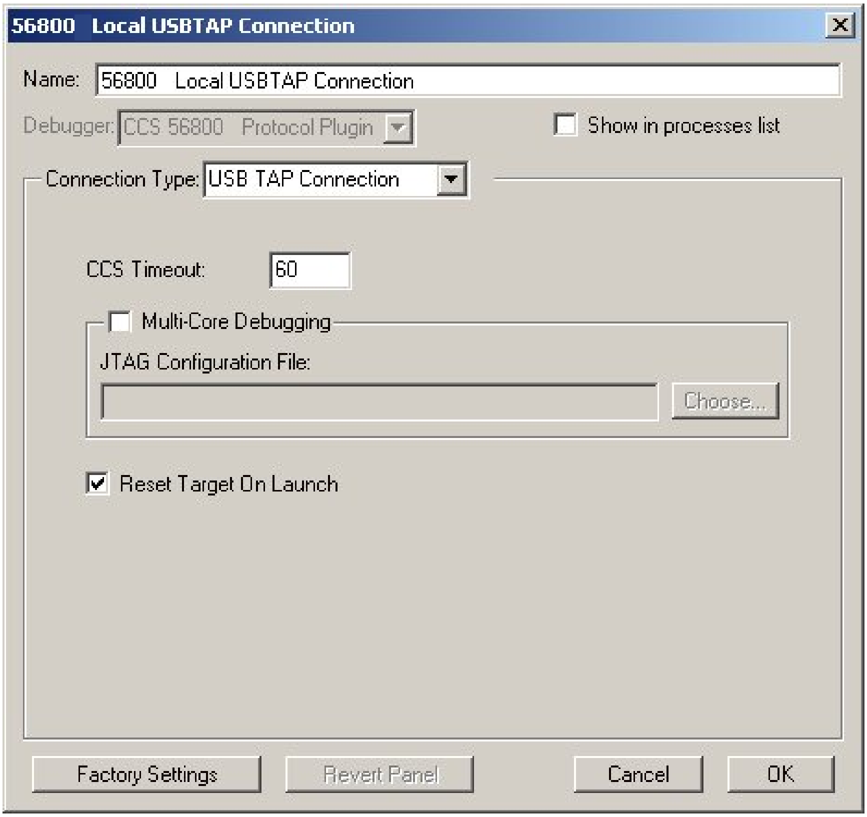


Table 9.4 on page 204 describes the options in this dialog box.

Table 9.4 UBTAP Options

Option	Description
Name	Enter the name you want to use to refer to this remote connection within the CodeWarrior IDE.
Debugger	Select **CCS 56800 Protocol Plugin**.
Connection Type	Select **USBTAP Connection**.
CCS Timeout	Enter the maximum number of seconds the debugger should wait for a response from CCS. By default, the debugger waits up to 10 seconds for responses.

Table 9.4 UBTAP Options (*continued*)

Option	Description
Multi-Core Debugging	Check to debug code on a target system with multiple cores where you need to specify the JTAG chain for debugging. Click **Choose** to specify the JTAG initialization file. A JTAG initialization file contains the names and order of the boards / cores you want to debug.
Reset Target on Launch	Check to have the debugger send a reset signal to the target system when you start debugging. Clear to prevent the debugger from resetting the target device when you start debugging.

Simulator

Use this connection type to configure the behavior of the simulator. This connection type is available only when the **56800 Simulator Protocol Plugin** debugger protocol is selected.

Figure 9.5 on page 206 shows the setting that are available to you when you select **Simulator** from the **Connection Type** list box in the **Edit Connection** dialog box.

Figure 9.5 Simulator Connection Settings

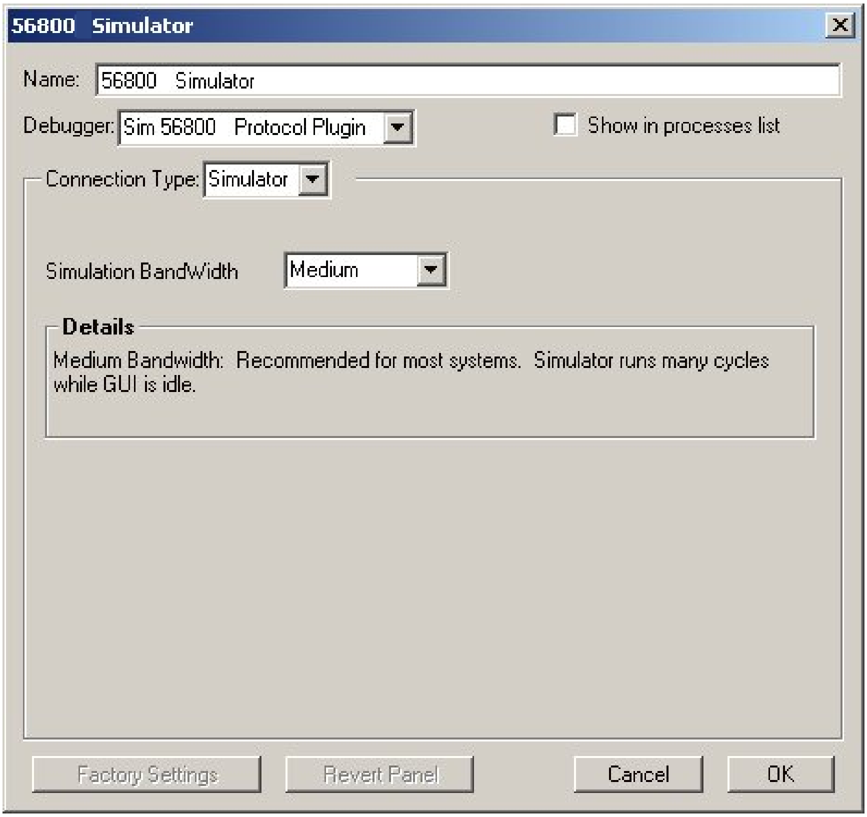


Table 9.5 on page 206 describes the options in this dialog box.

Table 9.5 Simulator Options

Option	Description
Name	Enter the name you want to use to refer to this remote connection within the CodeWarrior IDE.
Debugger	Select **SIM 56800 Protocol Plugin**.
Connection Type	Select **Simulator**.
Simulation Bandwidth	Select the simulator bandwidth (**low**, **medium**, or **high**).

Target Settings for Debugging

This section explains how to control the debugger by modifying the appropriate settings panels.

To properly debug DSP56800 software, you must set certain preferences in the **Target Settings** window. The **M56800 Target** panel is specific to DSP56800 development. The remaining settings panels are generic to all build targets.

Other settings panels can affect debugging. Table 9.6 on page 207 lists these panels.

Table 9.6 Setting Panels that Affect Debugging

This panel…	Affects…	Refer to…
M56800 Linker	symbolics, linker warnings	"M56800 Linker"
M56800 Processor	optimizations	"Optimizing Code"
Debugger Settings	Debugging options	
Remote Debugging	Debugging communication protocol	"Remote Debugging"
Remote Debug Options	Debugging options	"Remote Debug Options"

The **M56800 Target** panel is unique to DSP56800 debugging. The available options in this panel depend on the DSP56800 hardware you are using and are described in detail in the section on "Remote Debug Options".

Command Converter Server

The command converter server (CCS) handles communication between the CodeWarrior debugger and the target board. An icon in the status bar indicates the CCS is running. The CCS is automatically launched by your project when you start a CCS debug session if you are debugging a target board using a local machine. However, when debugging a target board connected to a remote machine, see "Setting Up a Remote Connection".

NOTE Projects are set to debug locally by default. The protocol the debugger uses to communicate with the target board, for example, PCI, is determined by how you installed the CodeWarrior software. To modify the protocol, make changes in the **Freescale Command Converter Server** window (Figure 9.8 on page 209).

Essential Target Settings for Command Converter Server

Before you can download programs to a target board for debugging, you must specify the target settings for the command converter server:

- Local Settings

  If you specify that the CodeWarrior IDE start the command converter server locally, the command converter server uses the connection port (for example, LPT1) that you specified when you installed CodeWarrior Development Studio for Freescale 56800.

- Remote Settings

  If you specify that the CodeWarrior IDE start the command converter server on a remote machine, specify the IP address of the remote machine on your network (as described in "Setting Up a Remote Connection".)

- Default Settings

  By default, the command converter server listens on port 41475. You can specify a different port number for the debugger to connect to if needed (as described in "Setting Up a Remote Connection".) This is necessary if the CCS is configured to a port other than 41475.

After you have specified the correct settings for the command converter server (or verified that the default settings are correct), you can download programs to a target board for debugging.

The CodeWarrior IDE starts the command converter server at the appropriate time if you are debugging on a local target.

Before debugging on a board connected to a remote machine, ensure the following:

- The command converter server is running on the remote host machine.
- Nobody is debugging the board connected to the remote host machine.

Changing the Command Converter Server Protocol to Parallel Port

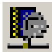
If you specified the wrong parallel port for the command converter server when you installed CodeWarrior Development Studio for Freescale 56800, you can change the port.

Change the parallel port:

1. Click the command converter server icon.

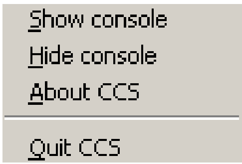
   While the command converter server is running, locate the command converter server icon on the status bar. Right-click on the command converter server icon (Figure 9.6 on page 209):

Figure 9.6 Command Converter Server Icon



A menu appears (Figure 9.7 on page 209):

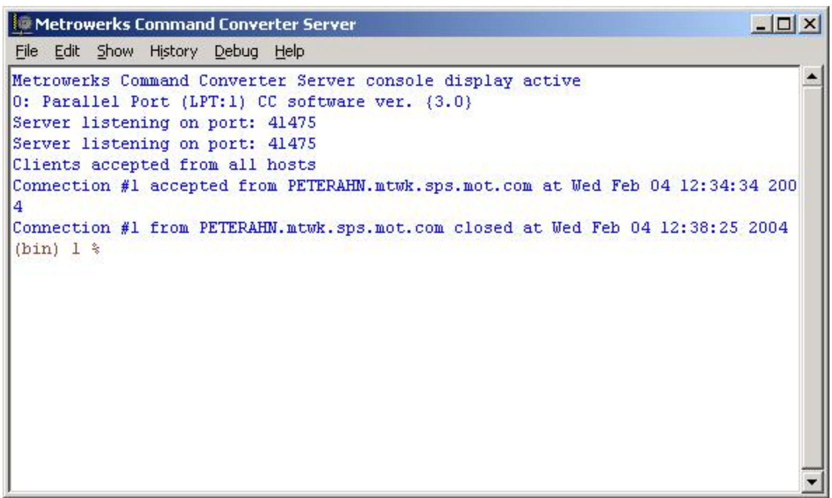
Figure 9.7 Command Converter Server Menu



2. Select Show console from the menu.

   The **Freescale Command Converter Server** window appears (Figure 9.8 on page 209).

Figure 9.8 Freescale Command Converter Server Window



3. On the console command line, type the following command:

   `delete all`

4. Press Enter.
5. Type the following command, substituting the number of the parallel port to use (for example, 1 for LPT1):

   `config cc parallel:1`

6. Press Enter.
7. Type the following command to save the configuration:

   `config save`

8. Press Enter.

Changing the Command Converter Server Protocol to HTI

To change the command converter server to an HTI Connection:

1. While the command converter server is running, right-click on the command converter server icon shown in Figure 9.6 on page 209 or double click on it.
2. From the menu shown in Figure 9.7 on page 209, select Show Console.
3. At the console command line in the Freescale Command Converter Server window shown in Figure 9.8 on page 209, type the following command:

   `delete all`

4. Press Enter.
5. Type the following command:

   `config cc: address`

   (substituting for **`address`** the name of the IP address of your CodeWarrior HTI)

NOTE If the software rejects this command, your CodeWarrior HTI may be an earlier version. Try instead the command: `config cc nhti:address`, or the command: `config cc Panther:address`, substituting for **`address`** the IP address of the HTI.

6. Press Enter.
7. Type the following command to save the configuration:

   `config save`

8. Press Enter.

Changing the Command Converter Server Protocol to PCI

To change the command converter server to a PCI Connection:

1. While the command converter server is running, right-click on the command converter server icon shown in Figure 9.6 on page 209 or double click on it.
2. From the menu shown in Figure 9.7 on page 209, select Show Console.
3. At the console command line in the Freescale Command Converter Server window shown in Figure 9.8 on page 209, type the following command:

   `delete all`
4. Press Enter.
5. Type the following command:

   `config cc pci`
6. Press Enter.
7. Type the following command to save the configuration:

   `config save`
8. Press Enter.

Setting Up a Remote Connection

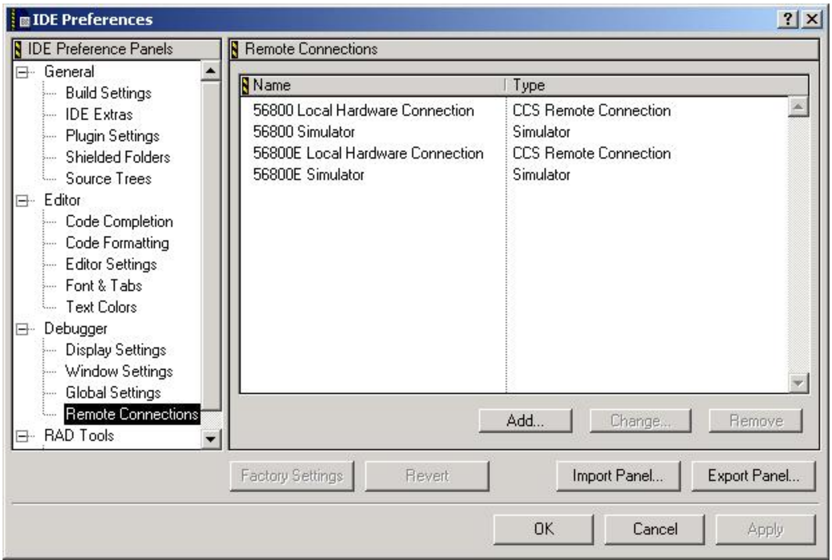
A remote connection is a type of connection to use for debugging along with any preferences that connection may need. To change the preferences for a remote connection or to create a new remote connection:

1. On the main menu, select Edit > Preferences.

   The IDE Preferences Window appears.
2. Click Remote Connections in the left column.

   The **Remote Connections** panel shown in Figure 9.9 on page 212 appears.

Figure 9.9 Remote Connections Panel



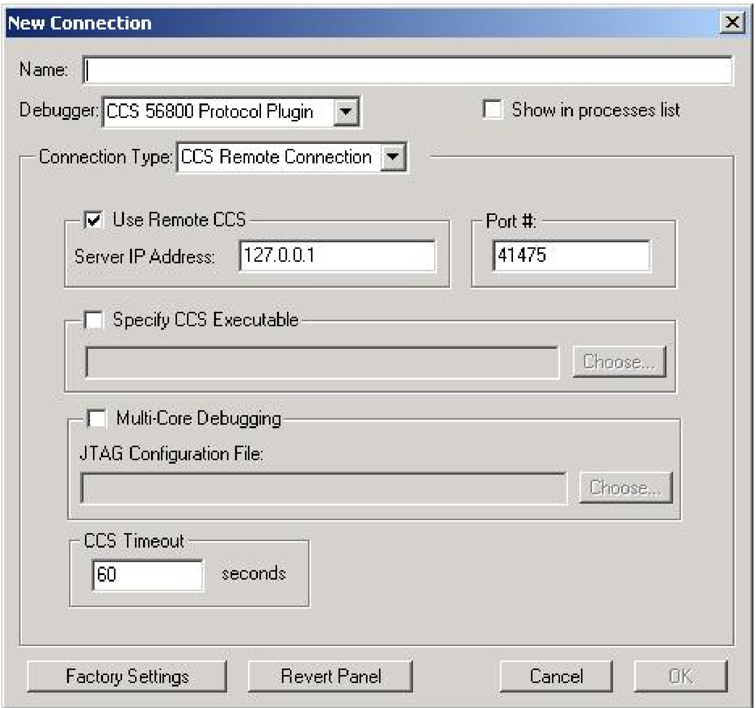
To Add a New Remote Connection

To add a new remote connection:

1. Click the Add button.

   The **New Connection** window appears as shown in Figure 9.10 on page 213.

Figure 9.10 New Connection Window



2. In the Name edit box, type in the connection name.
3. Check Use Remote CCS checkbox.

   Select this checkbox to specify that the CodeWarrior IDE is connected to a remote command converter server. Otherwise, the IDE starts the command converter server locally
4. Enter the Server IP address or host machine name.

   Use this text box to specify the IP address where the command converter server resides when running the command converter server from a location on the network.
5. Enter the Port # to which the command converter server listens or use the default port, which is 41475.
6. Click the OK button.

To Change an Existing Remote Connection

To change an existing remote connection:

Double click on the connection name that you want to change, or click once on the connection name and click the **Change** button (shown in Figure 9.9 on page 212 in grey).

To Remove an Existing Remote Connection

To remove an existing remote connection:

Click once on the connection name and click the **Remove** button (shown in Figure 9.9 on page 212 in grey).

Debugging a Remote Target Board

For debugging a target board connected to a remote machine with Code Warrior IDE installed, perform the following steps:

1. Connect the target board to the remote machine.
2. Launch the command converter server (CCS) on the remote machine with the local settings configuration using instructions described in the section "Essential Target Settings for Command Converter Server".
3. In the Target Settings>Remote Debugging panel for your project, make sure the proper remote connection is selected.
4. Launch the debugger.

Launching and Operating the Debugger

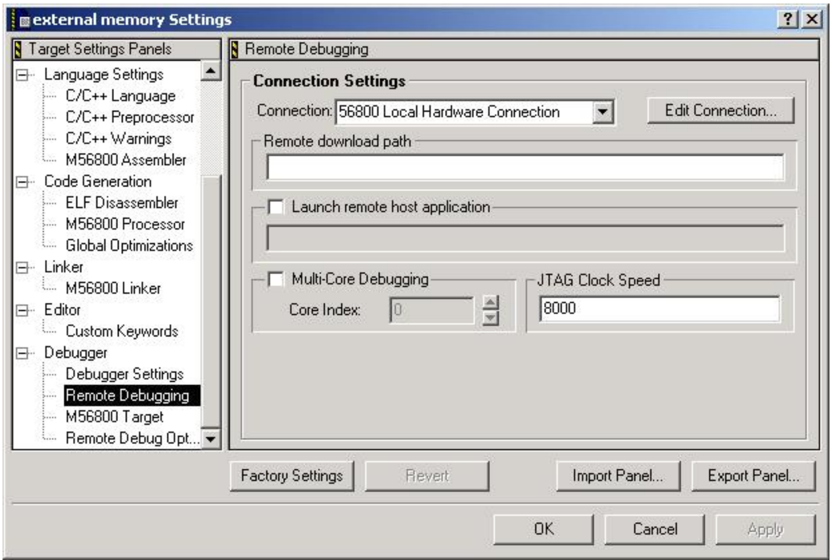
NOTE CodeWarrior IDE automatically enables the debugger and sets debugger-related settings within the project.

1. Set debugger preferences.

   Select **Edit >external memory Settings** from the menu bar of the Freescale CodeWarrior window.

   The IDE displays the **Remote Debugging** window.

Figure 9.11 Remote Debugging Panel



2. Select the Connection.

   For example, select **56800 Local Hardware Connection (CCS).**

3. Click OK button.

4. Debug the project.

   Use either of the following options:

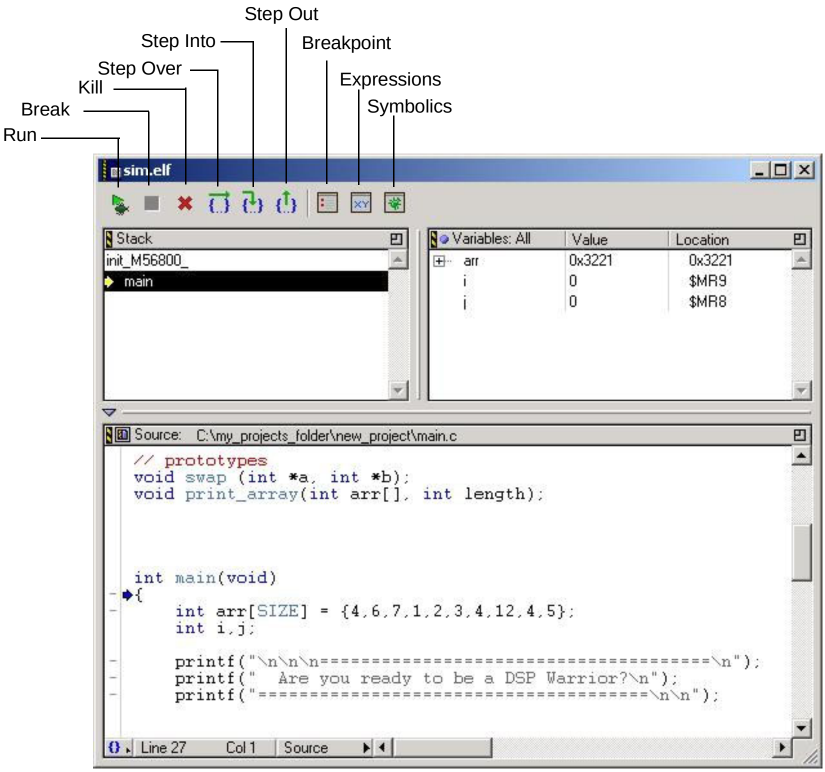
   - From the Freescale CodeWarrior window, select **Project > Debug**.
   - Click the **Debug** button in the project window.

This command resets the board (if **Always reset on download** is checked in the Debugger's **M56800 Target** panel shown in Figure 5.13 on page 95) and the download process begins.

When the download to the board is complete, the IDE displays the **Program** window (**sim.elf** in sample) shown in Figure 9.12 on page 216.

NOTE Source code is shown only for files that are in the project folder or that have been added to the project in the project manager, and for which the IDE has created debug information. You must navigate the file system in order to locate sources that are outside the project folder and not in the project manager, such as library source files.

Figure 9.12 Program Window



5. Navigate through your code.

   The **Program** window has three panes:

   - Stack pane

     The **Stack** pane shows the function calling stack.

   - Variables pane

     The **Variables** pane displays local variables.

   - Source pane

     The **Source** pane displays source or assembly code.

   The toolbar at the top of the window has buttons that allow you access to the execution commands in the **Debug** menu.

Setting Breakpoints

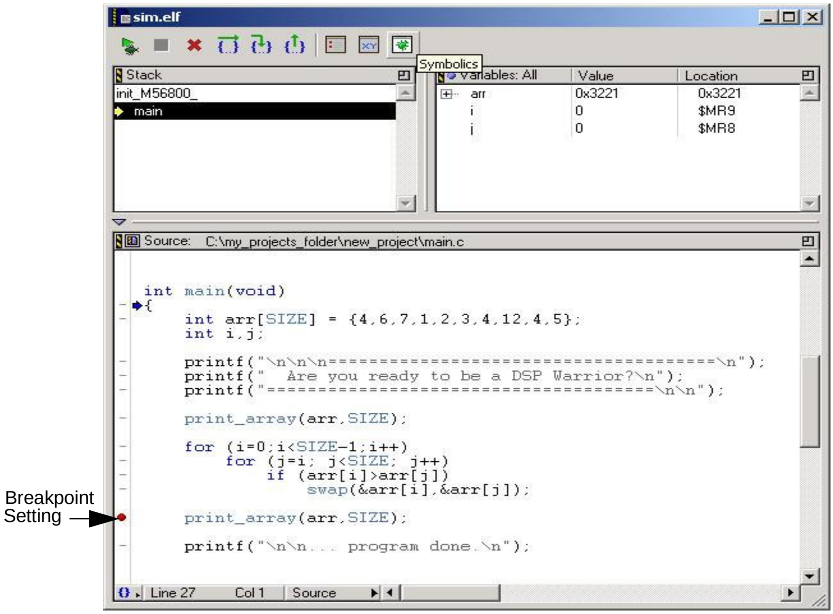
1. Locate the code line.

   Scroll through the code in the **Source** pane of the **Program** window until you come across the `main()` function.

2. Select the code line.

   Click the gray dash in the far left-hand column of the window, next to the first line of code in the `main()` function. A red dot appears (Figure 9.13 on page 217), confirming you have set your breakpoint.

Figure 9.13 Breakpoint in the Program Window



NOTE To remove the breakpoint, click the red dot. The red dot disappears.

Setting Watchpoints

For details on how to set and use watchpoints, see the "OnCE Debugger Features"..

NOTE For the DSP56800 only one watchpoint is available. This watchpoint is only available on hardware targets.

Viewing and Editing Register Values

Registers are platform-specific. Different chip architectures have different registers.

1. **Access the Registers** window.

   From the menu bar of the Freescale CodeWarrior window, select **View** > **Registers**.

   Expand the **General Purpose Registers** tree control to view the registers as in Figure 9.14 on page 218, or double-click on **General Purpose Registers** to view the registers as in Figure 9.15 on page 219.

Figure 9.14 General Purpose Registers for DSP56800

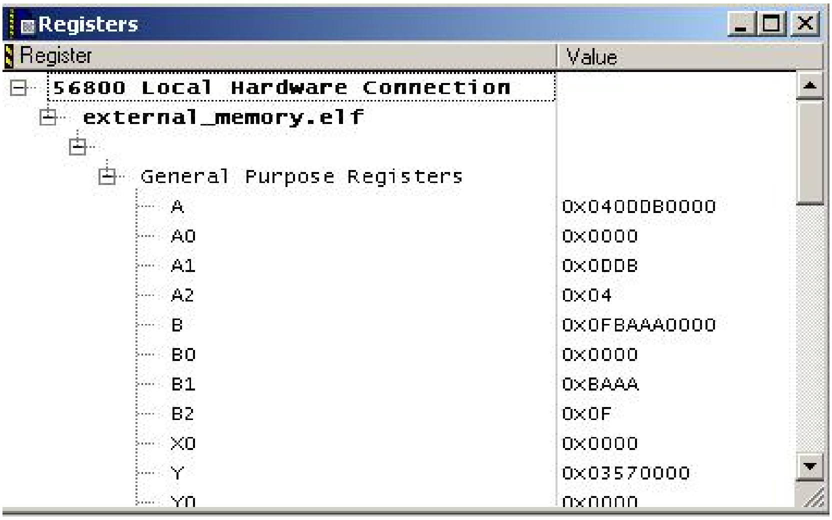
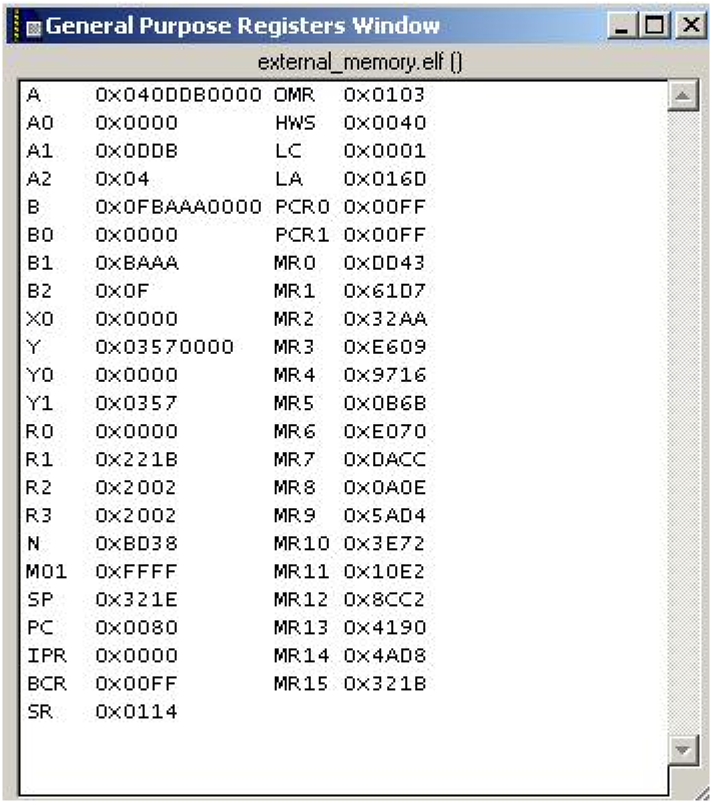


Figure 9.15 General Purpose Registers Window



2. Edit register values.

   To edit values in the register window, double-click a register value. Change the value as you wish.

3. Exit the window.

   The modified register values are saved.

NOTE To view peripheral registers, select the appropriate processor form the processor list box in the M56800 Target Settings Panel.

Viewing X: Memory

You can view X memory space values as hexadecimal values with ASCII equivalents. You can edit these values at debug time.

NOTE On targets that have Flash ROM, you cannot edit those values in the memory window that reside in Flash memory.

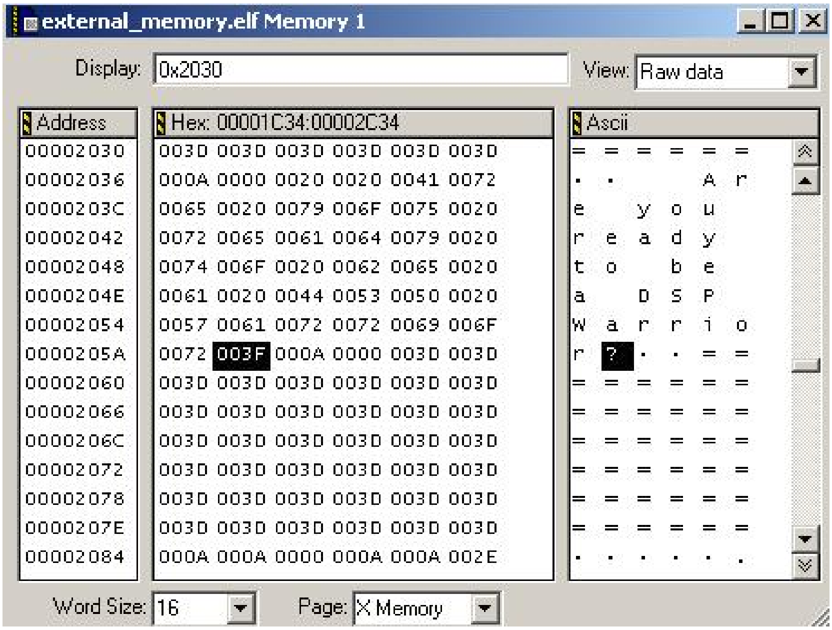
1. Locate a particular address in program memory.

   From the menu bar of the Freescale CodeWarrior window, select **Data > View Memory**.

NOTE The **Source** pane in the **Program** window needs to be the active one in order for the **Data > View Memory** to be activated.

The **Memory** window appears (Figure 9.16 on page 220).

Figure 9.16 View X:Memory Window



2. Select type of memory.

   Locate the **Page** list box at the bottom of the **View Memory** window. Select **X Memory** for X Memory.

3. Enter memory address.

   Type the memory address in the **Display** field located at the top of the **Memory** window.

   To enter a hexadecimal address, use standard C hex notation, for example, 0x0. You also can enter the symbolic name whose value you want to view by typing its name in the **Display** field of the **Memory** window.

NOTE The other view options (Disassembly, Source and Mixed) do not apply when viewing X memory.

Viewing P: Memory

You can view P memory space and edit the opcode hexadecimal values at debug time.

NOTE On targets that have Flash ROM, you cannot edit those values in the memory window that reside in Flash memory.

1. Locate a particular address in program memory.

   To view program memory, from the menu bar of the Freescale CodeWarrior window, select **Data** > **View Memory**.

   The **Memory** window appears (Figure 9.16 on page 220).

2. Select type of memory.

   Locate the **Page** list box at the bottom of the **View Memory** window. Select **P Memory** for P Memory.

3. Enter memory address.

   Type the memory address in the **Display** field located at the top of the **Memory** window.

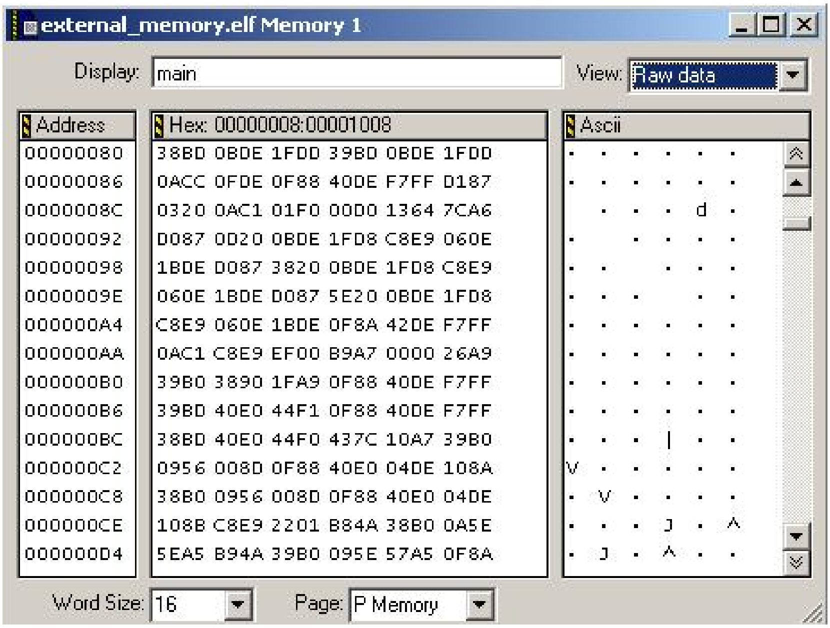
   To enter a hexadecimal address, use standard C hex notation, for example: `0x82`.

4. Select how you want to view P memory.

   Using the **View** list box, you have the option to view P Memory in four different ways.

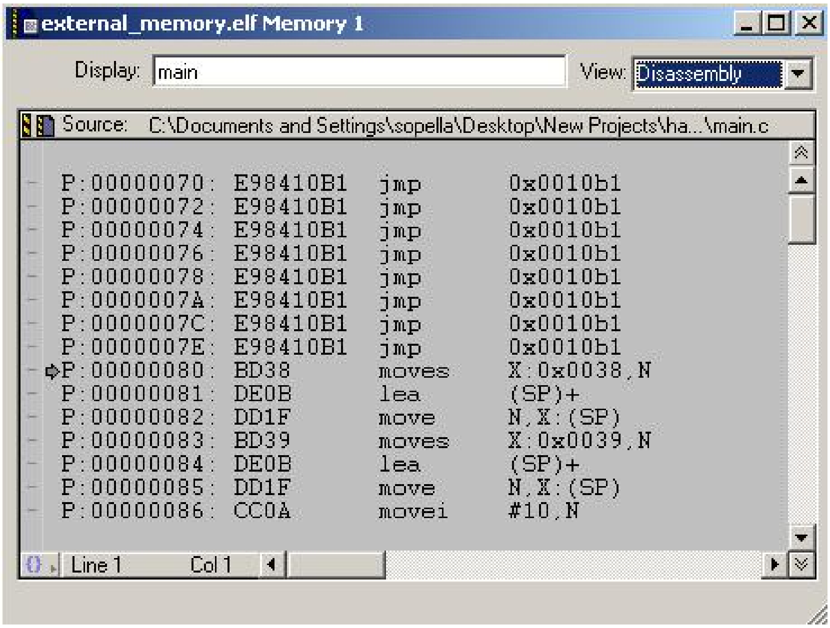
   - **Raw Data** (Figure 9.17 on page 222).

Figure 9.17 View P:Memory (Raw Data) Window



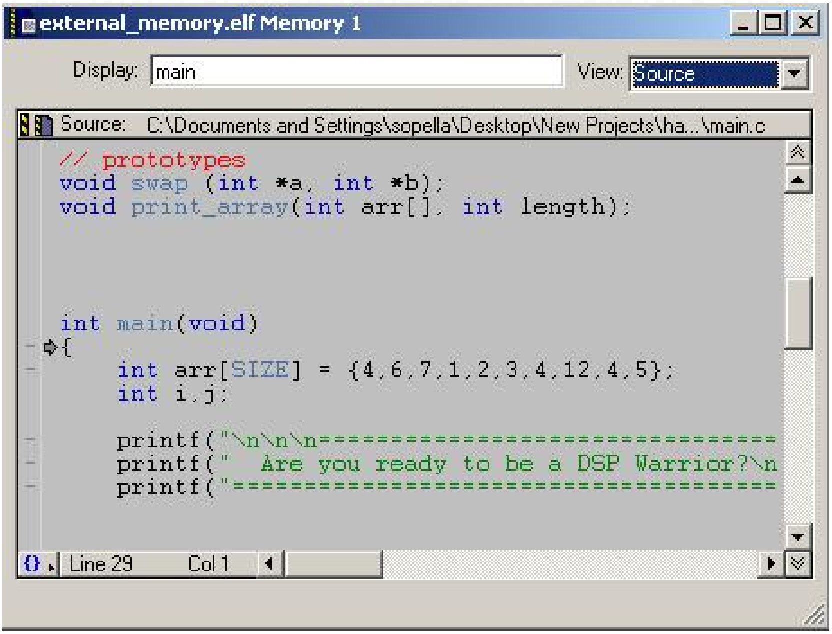
- **Disassembly** (Figure 9.18 on page 223).

Figure 9.18 View P:Memory (Disassembly) Window



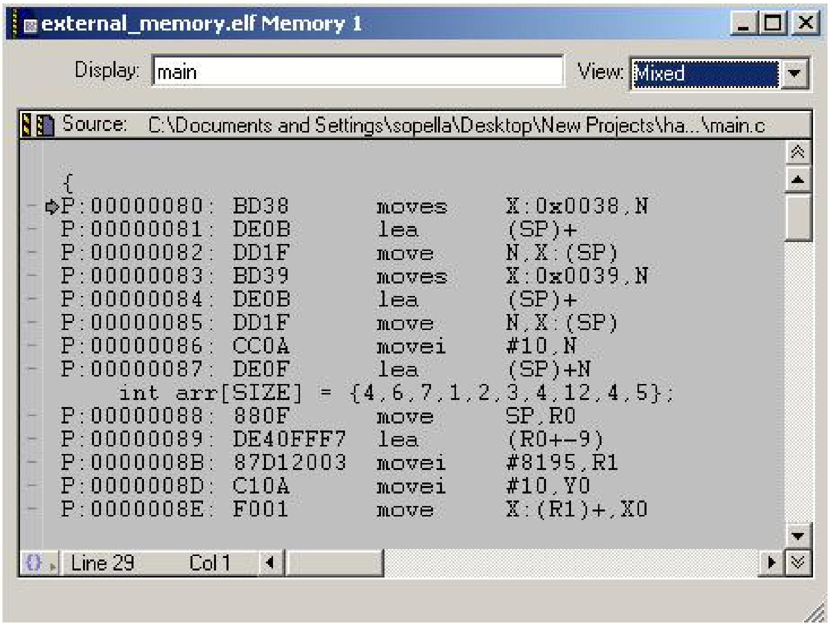
- **Source** (Figure 9.19 on page 224).

Figure 9.19 View P:Memory (Source) Window



- **Mixed** (Figure 9.20 on page 225).

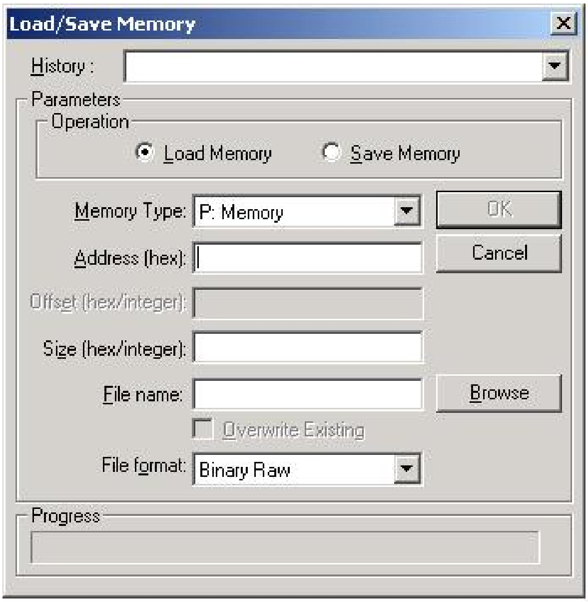
Figure 9.20 View P:Memory (Mixed) Window



Load/Save Memory

From the menu bar of the Freescale CodeWarrior window, select **Debug > 56800 > Load/Save Memory** to display the **Load/Save Memory** dialog box (Figure 9.21 on page 226).

Figure 9.21 Load/Save Memory Dialog Box



Use this dialog box to load and save memory at a specified location and size with a user-specified file. You can associate a key binding with this dialog box for quick access. Press the **Tab** key to cycle through the dialog box displays, which lets you quickly make changes without using the mouse.

History Combo Box

The **History** combo box displays a list of recent loads and saves. If this is the first time you load or save, the **History** combo box is empty. If you load/save more than once, the combo box fills with the memory address of the start of the load or save and the size of the fill, to a maximum of ten sessions.

If you enter information for an item that already exists in the history list, that item moves up to the top of the list. If you perform another operation, that item appears first.

NOTE By default, the **History** combo box displays the most recent settings on subsequent viewings.

Radio Buttons

The **Load/Save Memory** dialog box has two radio buttons:

- Load Memory
- Save Memory

The default is **Load Memory**.

Memory Type Combo Box

The memory types that appear in the **Memory Type** Combo box are:

- P: Memory (Program Memory)
- X: Memory (Data Memory)

Address Text Field

Specify the address where you want to write the memory. If you want your entry to be interpreted as hex, prefix it with `0x`; otherwise, it is interpreted as decimal.

Size Text Field

Specify the number of words to write to the target. If you want your entry to be interpreted as hex, prefix it with `0x`; otherwise, it is interpreted as decimal.

Dialog Box Controls

Cancel, Esc, and OK

In Load and Save operations, all controls are disabled except **Cancel** for the duration of the load or save. The status field is updated with the current progress of the operation. Clicking **Cancel** halts the operation, and re-enables the controls on the dialog box. Clicking **Cancel** again closes the dialog box. Pressing the **Esc** key is same as clicking the **Cancel** button.

With the **Load Memory** radio button selected, clicking **OK** loads the memory from the specified file and writes it to memory until the end of the file or the size specified is reached. If the file does not exist, an error message appears.

With the **Save Memory** radio button selected, clicking **OK** reads the memory from the target piece by piece and writes it to the specified file. The status field is updated with the current progress of the operation.

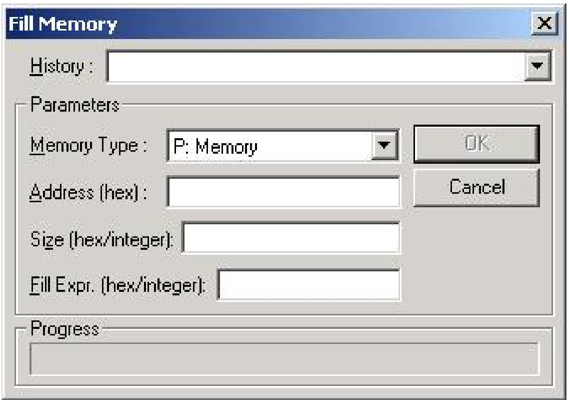
Browse Button

Clicking the **Browse** button displays OPENFILENAME or SAVEFILENAME, depending on whether you selected the **Load Memory** or **Save Memory** radio button.

Fill Memory

From the menu bar of the Freescale CodeWarrior window, select **Debug > 56800> Fill memory** to display the **Fill Memory** dialog box (Figure 9.22 on page 228).

Figure 9.22 Fill Memory Dialog Box



Use this dialog box to fill memory at a specified location and size with user- specified raw memory data. You can associate a key binding with this dialog box for quick access. Press the **Tab** key to cycle through the dialog box display, which lets you quickly make changes without using the mouse.

History Combo Box

The **History** combo box displays a list of recent fill operations. If this is the first time you perform a fill operation, the **History** combo box is empty. If you do more than one fill, then the combo box populates with the memory address of that fill, to a maximum of ten sessions.

If you enter information for an item that already exists in the history list, that item moves up to the top of the list. If you do another fill, then this item is the first one that appears.

NOTE By default, the **History** combo box displays the most recent settings on subsequent viewings.

Memory Type Combo Box

The memory types that can appear in the **Memory Type** Combo box are:

- P:Memory (Program Memory)
- X:Memory (Data Memory)

Address Text Field

Specify the address where you want to write the memory. If you want it to be interpreted as hex, prefix it with `0x`; otherwise, it is interpreted as decimal.

Size Text Field

Specify the number of words to write to the target. If you want it to be interpreted as hex, prefix your entry with `0x`; otherwise, it is interpreted as decimal.

Fill Expression Text Field

Fill writes a set of characters to a location specified by the address field on the target, repeatedly copying the characters until the user-supplied fill size has been reached. **Size** is the total words written, not the number of times to write the string.

Interpretation of the Fill Expression

The fill string is interpreted differently depending on how it is entered in the Fill String field. Any words prefixed with `0x` is interpreted as hex bytes. Thus, `0xBE 0xEF` would actually write `0xBEEF` on the target. Optionally, the string could have been set to `0xBEEF` and this would do the same thing. Integers are interpreted so that the equivalent signed integer is written to the target.

ASCII Strings

ASCII strings can be quoted to have literal interpretation of spaces inside the quotes. Otherwise, spaces in the string are ignored. Note that if the ASCII strings are not quoted and they are numbers, it is possible to create illegal numbers. If the number is illegal, an error message is displayed.

Dialog Box Controls

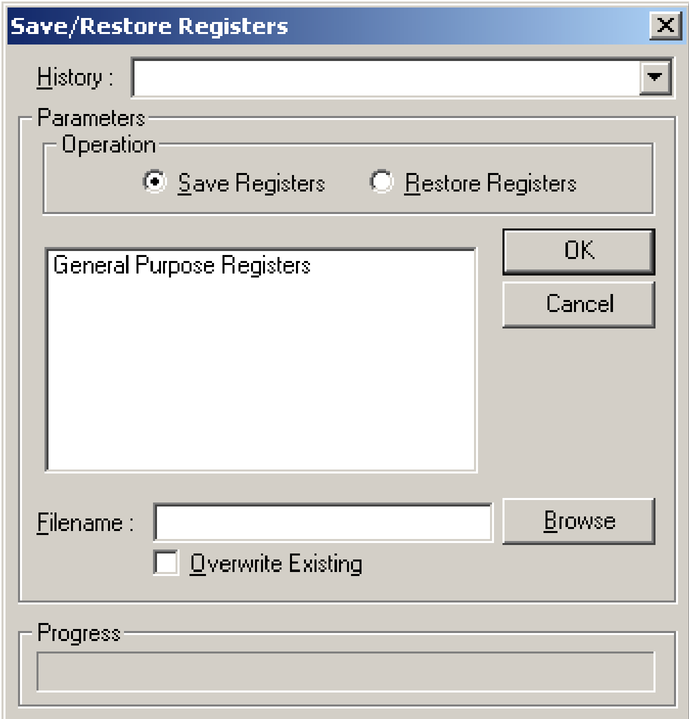
OK, Cancel, and Esc

Clicking **OK** writes the memory piece by piece until the target memory is filled in. The **Status** field is updated with the current progress of the operation. When this is in progress, the entire dialog box grays out except the **Cancel** button, so the user cannot change any information. Clicking the **Cancel** button halts the fill operation, and re-enables the controls on the dialog box. Clicking the **Cancel** button again closes the dialog box. Pressing the **Esc** key is same as pressing the **Cancel** button.

Save/Restore Registers

From the menu bar of the Freescale CodeWarrior window, select **Debug > 56800 > Save/Restore Registers** to display the **Save/Restore Registers** dialog box (Figure 9.23 on page 231).

Figure 9.23 Save/Restore Registers Dialog Box



Use this dialog box to save and restore register groups to and from a user-specified file.

History Combo Box

The **History** combo box displays a list of recent saves and restores. If this is the first time you have saved or restored, the **History** combo box is empty. If you saved or restored before, the combo box remembers your last ten sessions. The most recent session will appear at the top of the list.

Radio Buttons

The **Save/Restore Registers** dialog box has two radio buttons:

- Save Registers
- Restore Registers

The default is **Save Registers**.

Register Group List

This list is only available when you have selected **Save Registers**. If you have selected **Restore Registers**, the items in the list are greyed out. Select the register group that you wish to save.

Dialog Box Controls

Cancel, Esc, and OK

In Save and Restore operations, all controls are disabled except **Cancel** for the duration of the load or save. The status field is updated with the current progress of the operation. Clicking **Cancel** halts the operation, and re-enables the controls on the dialog box. Clicking **Cancel** again closes the dialog box. Pressing the **Esc** key is same as clicking the **Cancel** button.

With the **Restore Registers** radio button selected, clicking **OK** restores the registers from the specified file and writes it to the registers until the end of the file or the size specified is reached. If the file does not exist, an error message appears.

With the **Save Register** radio button selected, clicking **OK** reads the registers from the target piece by piece and writes it to the specified file. The status field is updated with the current progress of the operation.

Browse Button

Clicking the **Browse** button displays OPENFILENAME or SAVEFILENAME, depending on whether you selected the **Restore Registers** or **Save Registers** radio button.

OnCE Debugger Features

The following OnCE Debugger features are discussed in this section:

- Watchpoints and Breakpoints on page 232
- Trace Buffer on page 239

Watchpoints and Breakpoints

The CodeWarrior DSP56800 debugger allows you to monitor the status of a watchpoint. Since the OnCE™ port only supports either a hardware breakpoint or a watchpoint, you cannot have both active at the same time.

Watchpoints are useful for monitoring memory and processes where software breakpoints cannot be set, such as in Flash ROM, or a data or address bus. If the watchpoint status is used as a trace counter, it can also be helpful to debug sections of code that do not have a normal flow or are hung up in infinite loops.

Watchpoints are available regardless of whether you have checked "Use Hardware Breakpoints." The watchpoint status window does not report the status of hardware breakpoints. OnCE™ hardware only supports one hardware breakpoint or watchpoint at a time. If a watchpoint is in place, you cannot use a breakpoint and vice versa.

The CodeWarrior watchpoint debugger can monitor:

- Program memory addresses
- Data memory addresses
- The value on the Core Global Data Bus
- The value on the Program Address Bus
- Specified number of occurrences

NOTE If you are debugging Flash ROM, enable the Use Hardware breakpoints option in the M56800 Target Settings panel. However, you can use the Watchpoint status window debugging RAM as well.

Opening the Watchpoint Status Window

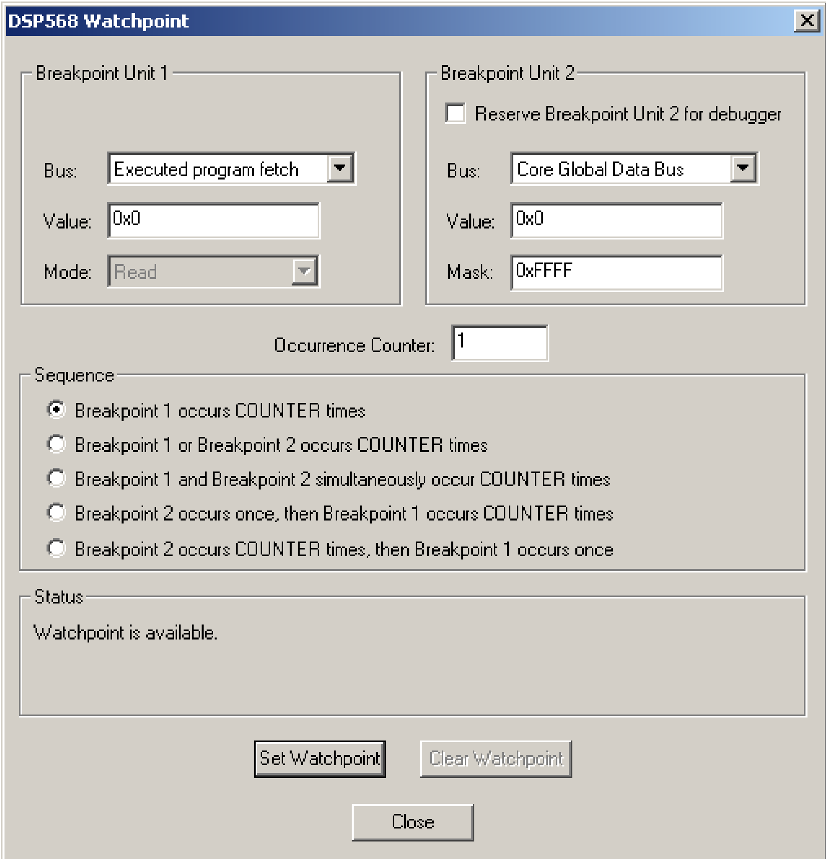
To select a new watchpoint status:

1. Start a debugging session.
2. From the menu bar of the Freescale CodeWarrior window, select **DSP56800 > Watchpoint status**.

   The **Watchpoint Status** window appears (Figure 9.24 on page 234).

NOTE The **Watchpoint Status** menu item is disabled when you use the Simulator or during a system-level connect.

Figure 9.24 Watchpoint Status Window



NOTE When you clear a custom watchpoint, the settings you last used are now selected instead of the previous default values. These settings do not carry over from previous debugging sessions.

Breakpoint Unit 1

Breakpoint unit 1 (BPU1) of the watchpoint status window allows you to monitor address values and access type for any X or P memory location.

Options for setting BPU1 are in the Breakpoint Unit 1 group box shown in Figure 9.25 on page 235 and listed in Table 9.7 on page 235.

Figure 9.25 Breakpoint Unit 1 Options

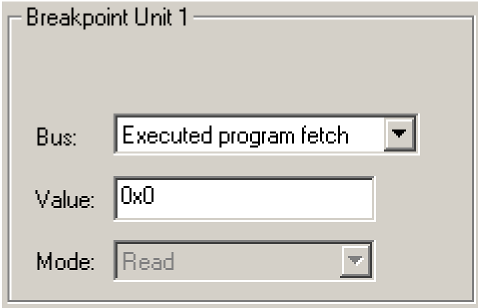


Table 9.7 Options for Breakpoint Unit 1

Setting	Value	Comment
Bus	Execute program fetch	When a P memory instruction is executed. **Mode** defaults to **Read**. Useful when only interest is opcode instructions.
	Any P memory access	Any time a P memory address is accessed, depending on the value of **Mode**. Useful when writing or reading data from P memory.
	X Address Bus 1	Access for all X address values through XAB1 (internal or external memory) depending on the **Mode** you select.
Value	C hexadecimal or decimal notation	Range: `0x0` to `0xFFFF`
Mode	Read	
	Write	
	Read and Write	

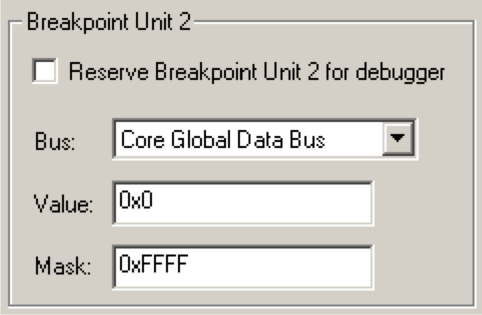
NOTE If Breakpoint Unit 2 is disabled (in use by the debugger), then the occurrence counter is set to 1 as the default.

Breakpoint Unit 2

Breakpoint unit 2 (BPU2) of the watchpoint status window allows you to monitor values (and their masks) in either the Core Global Data Bus (CGDB) or Program Address Bus (PAB). When you use BPU2 in conjunction with BPU1 and the occurrence counter, you can monitor the status of a watchpoint to a resolution as fine as 1 bit at single memory location.

Options for setting BPU2 are in the Breakpoint Unit 2 group box are in Figure 9.26 on page 236 and listed in Table 9.8 on page 236.

Figure 9.26 Breakpoint Unit 2 Options



NOTE If you are using Breakpoint Unit 2, ensure that one of the radio buttons is set to use Breakpoint 2 in the **Sequence** group box.

Table 9.8 Options for Breakpoint Unit 2

Setting	Value	Comment
Reserve Breakpoint Unit 2 for Debugger	Enabled	Breakpoint unit 2 cannot be user defined and the occurrence counter defaults to 1 for BPU1.
	Disabled	Breakpoint unit 2 is user-defined and occurrence counter is available for both BPU1 and BPU2. Single stepping, stepping over, and stepping out of functions cannot be done when hardware breakpoints are enabled.
Bus	**Core Global Data Bus** (CGDB)	Data transfer between the data ALU and X data memory for one memory access.
	Program Address Bus (PAB)	19-bit program memory address bus.

Table 9.8 Options for Breakpoint Unit 2 (*continued*)

Setting	Value	Comment
Value	The hexadecimal value read from the specified **Bus**.	To read full value, set **Mask** to `0xFFFF`.
Mask	Mask value in C hex notation from `0x0` to `0xFFFF`.	Specify a value of `0xFFFF` for full value specified by **Value**. Specify other hex value to exclude bits. For example, if you wanted to stop at any value where bit 15 is set, you would specify `0x8000` in both the **Mask** and **Value** fields

Occurrence Counter and Sequence Options

This section explains how the debugger uses the Occurrence Counter (hardware breakpoint counter) and Sequence Options when halting the debugger.

Occurrence Counter

The **Occurrence Counter** uses the OnCE breakpoint counter (OCNTR) for stopping on the *n*th iteration of a program loop or when the *n*th occurrence of a data memory access occurs. When you specify a value from 1 to 256 in the **Occurrence Counter** text box, it sets ONCTR to that value minus 1. Refer to *OnCE Breakpoint Counter (OCNTR)* in the *DSP56800 Family Manual* for more information.

NOTE Once the **Occurrence Counter** is decremented and a breakpoint is reached, the counter is not reset. Hence, the **Occurrence Counter** remains at one and stops at every specified breakpoint.

Sequence Options

To define the criteria for how often the debugger stops on a watchpoint, use the **Sequence** group box (Figure 9.27 on page 238). The value you set in the **Occurrence Counter** text box determines the value of COUNTER.

Figure 9.27 Sequence Counter Options in the Watchpoint Status Window

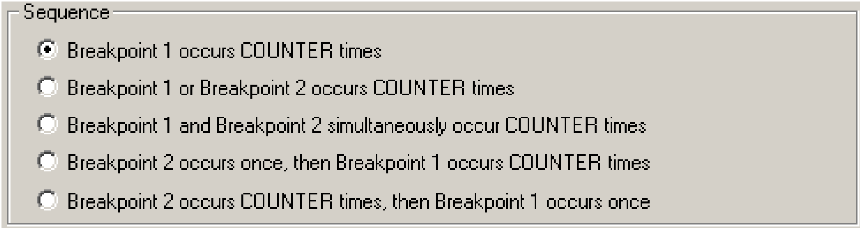


Table 9.9 on page 238 explains the options available in the **Sequence** group box

Table 9.9 Options for the Occurrence Counter

Option	Comment
Breakpoint 1 occurs COUNTER times	If **Reserve Breakpoint Unit 2 for Debugger** is enabled, this is the default option and COUNTER is 1.
Breakpoint 1 or Breakpoint 2 occurs COUNTER times	BPU1 and BPU2 work independently. If you are only interested in using BPU2, set BPU1 to a value you know will not be reached during program execution.
Breakpoint 1 and Breakpoint 2 simultaneously occur COUNTER times	BPU1 and BPU2 work together. This is useful for monitoring bit status with a defined mask.
Breakpoint 2 occurs once, then Breakpoint 1 occurs COUNTER times	Useful for monitoring the status of recursive or nested algorithms.
Breakpoint 2 occurs COUNTER times, then Breakpoint 1 occurs once	Useful for monitoring the status of recursive or nested algorithms

Setting and Clearing Watchpoint Status

You can set and clear a watchpoint only through the **Watchpoint Status** window. Use the following commands:

- Set Watchpoint

  Enables a watchpoint for the values specified by BPU1 and BPU2. Hardware breakpoints are not available when a watchpoint is set.

- Clear Watchpoint

  Disables the current watchpoint and returns all values in the **Watchpoint Status** window to their default values.

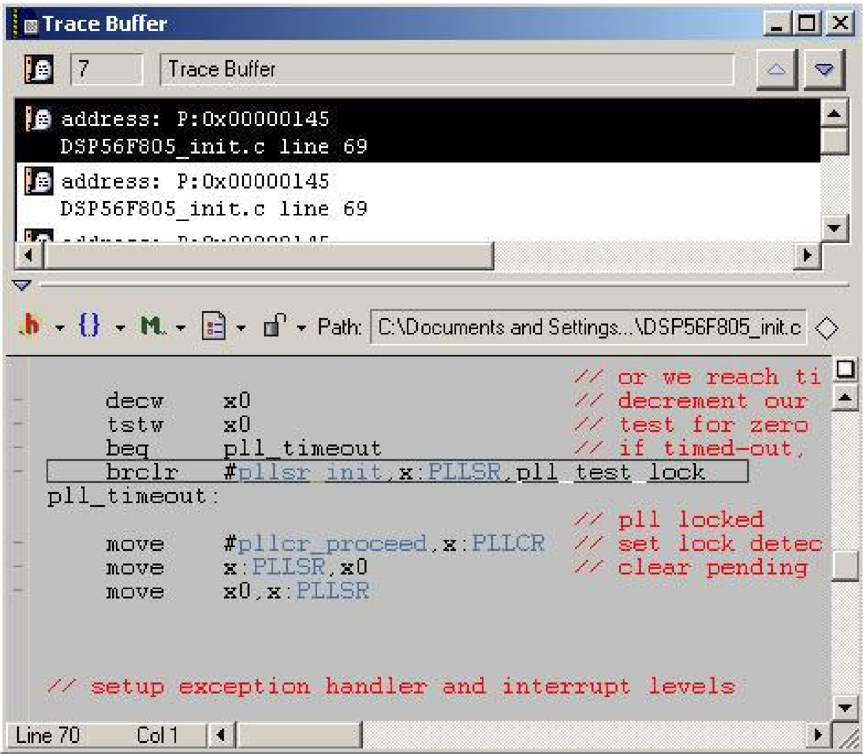
Trace Buffer

From the menu bar of the Freescale CodeWarrior window, select **DSP56800 > Dump Trace Buffer** to see the most recent changes in the program flow and a reconstructed program trace (Figure 9.28 on page 239).

Use this feature to query the Trace Buffer, located in the On-Chip Emulation module of a hardware target. This buffer stores the eight most recent changes in the program flow. The debugger retrieves these addresses and attempts to reconstruct a trace of the program flow. This occurs both when the window is opened and whenever debugging stops while the window is open.

The **Trace Buffer** menu item is enabled when the IDE is debugging a hardware target and debugging has stopped.

Figure 9.28 Trace Buffer Window

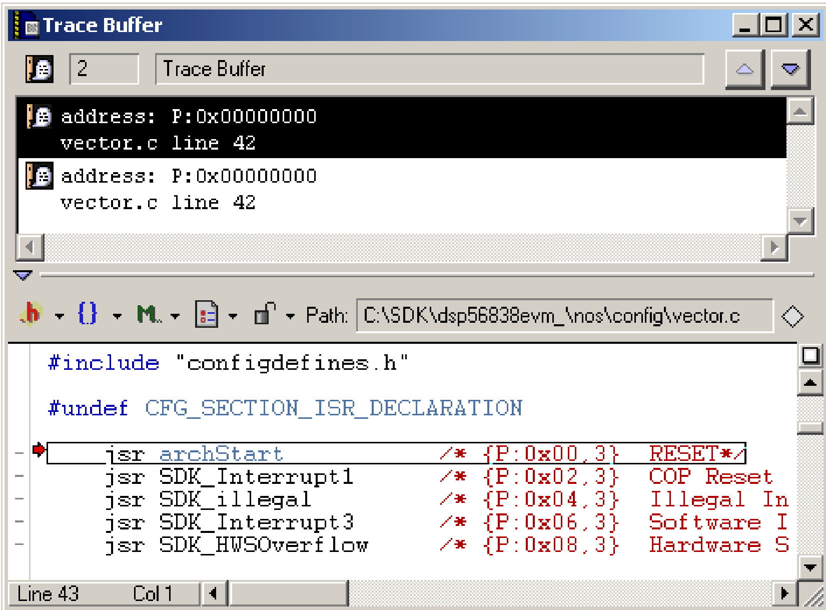


The trace buffer lets you view the target addresses of change-of-flow instructions that the program executes.

To view the contents of the trace buffer (Figure 9.29 on page 240):

1. From the IDE menu bar, select DSP56800 > Dump Trace Buffer.

Figure 9.29 Contents of Trace Buffer



Using the 56800 Simulator

The CodeWarrior Development Studio for Freescale 56800 includes the Freescale 56800 Simulator. This software lets you run and debug code on a simulated 56800 architecture without installing any additional hardware.

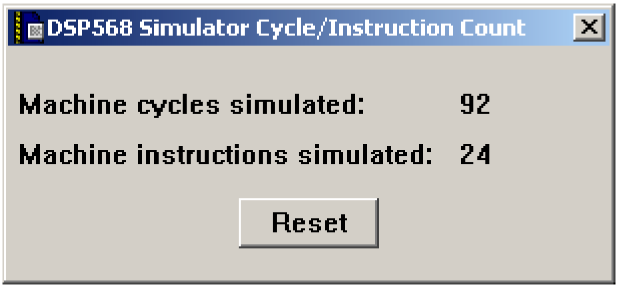
The simulator simulates the 56800 processor, not the peripherals. In order to use the simulator, you must select a connection that uses the simulator as your debugging protocol from the **Remote Debugging** panel.

NOTE The simulator also enables the 56800 menu for retrieving the machine cycle count and machine instruction count when debugging.

Cycle/Instruction Count

From the menu bar of the Freescale CodeWarrior window, select **56800 > Display Cycle/Instruction count**. The following window appears (Figure 9.30 on page 241):

Figure 9.30 Simulator Cycle/Instruction Count

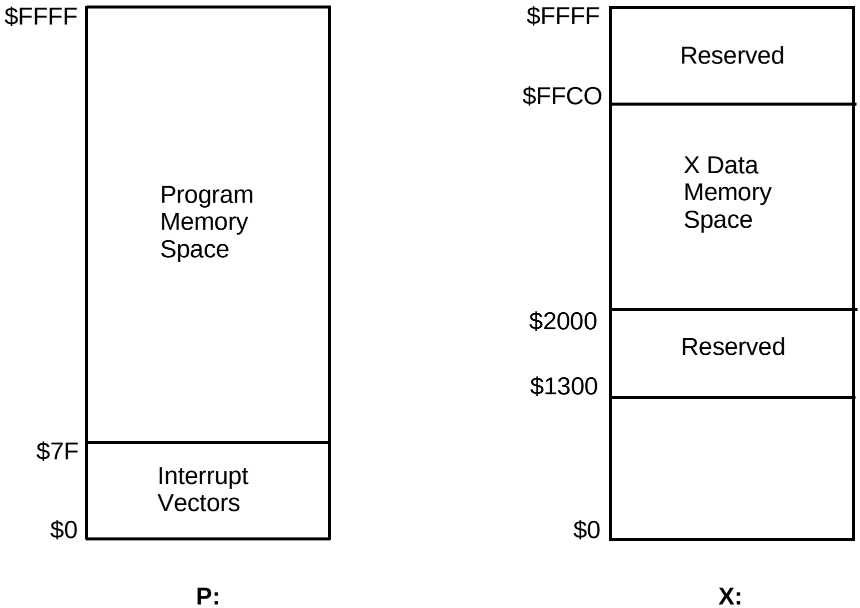


NOTE Cycle counting is not accurate while single stepping through source code in the debugger. It is only accurate while running. Thus, the cycle counter is more of a profiling tool than an interactive tool.

Press the **Reset** button to zero out the current machine-cycle and machine-instruction readings.

Memory Map

Figure 9.31 Simulator Memory Map

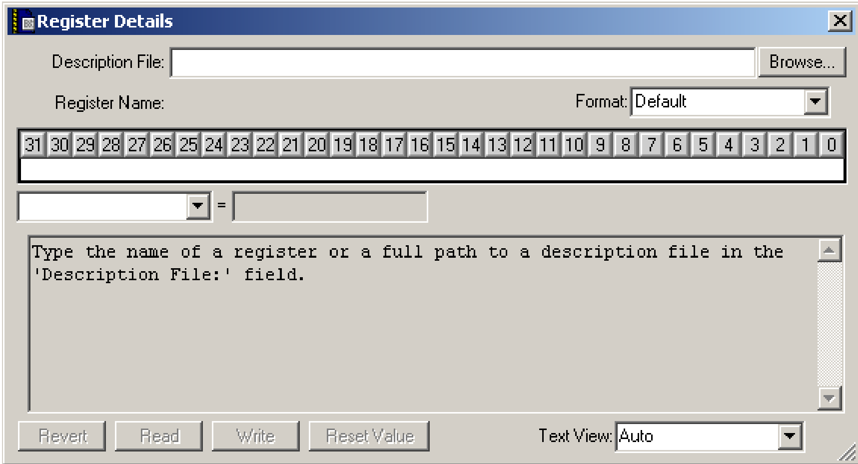


NOTE Figure 9.31 on page 242 is the memory map configuration for the simulator. Therefore, the simulator does not simulate each DSP568xx device's specific memory map, but assumes the memory map of the DSP56824.

Register Details Window

From the menu bar of the Freescale CodeWarrior window, select **View > Register Details** or in the Registers window (Figure 9.14 on page 218) double-click on the register. The **Register Details** window appears (Figure 9.32 on page 243).

Figure 9.32 Register Details Window



In the **Register Details** window, type the name of the register (e.g., OMR, SR, IPR, etc.) in the **Description File** field. The applicable register and its values appears.

By default, the CodeWarrior IDE looks in the following path when searching for register description files.

```
\CodeWarrior\bin\Plugins\support\Registers\M56800\GPR
```

Register description files must end with the `.xml` extension. Alternatively, you can use the **Browse** button to locate the register description files.

Using the **Format** list box in the **Register Details** window, you can change the format in which the CodeWarrior IDE displays the registers.

Using the **Text View** list box in the **Register Details** window, you can change the text information the CodeWarrior IDE displays.

Loading a .elf File without a Project

You can load and debug a `.elf` file without an associated project. To load a `.elf` file for debugging without an associated project:

1. Launch the CodeWarrior IDE.
2. Choose **File > Open** and specify the file to load in the standard dialog box that appears.

   Alternatively, you can drag and drop a `.elf` file onto the IDE.
3. You may have to add additional access paths in the Access Path preference panel in order to see all of the source code.

4. Choose **Project > Debug** to begin debugging the application.

NOTE When you debug a `.elf` file without a project, the IDE sets the **Build before running** setting on the Build Settings panel of the IDE Preference panels to Never. Consequently, if you open another project to debug after debugging a `.elf` file, you must change the **Build before running** setting before you can build the project.

The project that the CodeWarrior tools uses to create a new project for the given `.elf` file is `56800_Default_Project.xml` and is located in the path:

`CodeWarrior\bin\plugins\support directory`

You can create your own version of this file to use as a default setting when opening a `.elf` file:

1. Create a new project with the default setting you want.
2. Export the project to xml format.
3. Rename the xml format of the project to 56800_Default_Project.xml and place it in the support directory.

NOTE Back up or rename the original version of the default xml project before overwriting it with your own customized version.

Using the Command Window

In addition to using the regular CodeWarrior IDE debugger windows, you also can debug using Tcl scripts or the Command Window.

For more information on Tcl scripts and the Command Window, please see the *CodeWarrior Development Studio IDE 5.6 Windows® Automation Guide*.

System-Level Connect

The CodeWarrior DSP56800 debugger lets you connect to a loaded target board and view system registers and memory. A system-level connect does not let you view symbolic information during a connection.

NOTE The following procedure explains how to connect in the context of developing and debugging code on a target board. However, you can select the **Debug > Connect** command anytime you have a project window open, even if you have not yet downloaded a file to your target board.

To perform a system-level connect:

1. Select the Project window for the program you downloaded.
2. From the menu bar, select Debug > Connect.

   The debugger connects to the board. You can now examine registers and the contents of memory on the board.

Debugging on a Complex Scan Chain

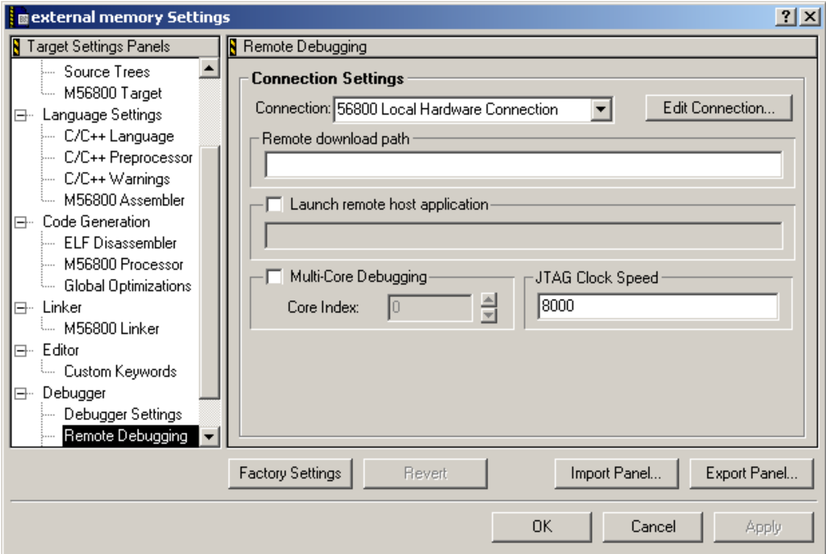
This section describes the procedure for debugging a chip connected on a complex JTAG chain.

Setting Up

The general steps for debugging a DSP56800 chip connected on a complex scan chain are:

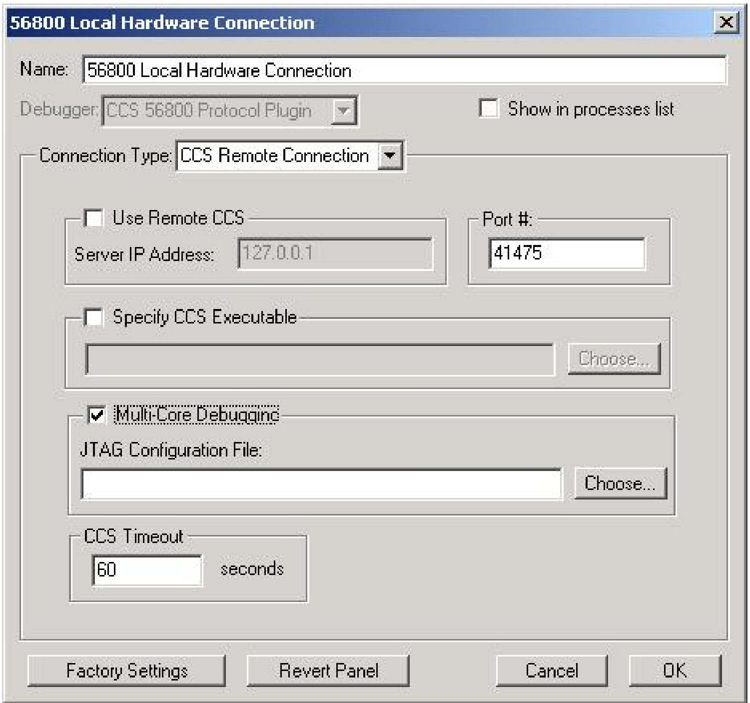
1. Set up and connect your JTAG chain of target boards.
2. Write a JTAG initialization file that describes the items on the JTAG chain.
3. Open a project to debug.
4. In the project you are debugging, open the **Remote Debugging** preference panel (Figure 9.33 on page 246).

Figure 9.33 Remote Debugging Preference Panel



5. Click the **Edit Connection** button and enable the **Multi-Core Debugging** checkbox ([Figure 9.34 on page 247](#)).

Figure 9.34 56800 Local Hardware Connection with Multi-Core Debugging Enabled



6. Specify the name and path of the JTAG initialization file in the **JTAG Configuration File** text field.
7. Click **OK** to close the connection panel.
8. In the **Remote Debugging** panel, specify the index of the core to debug by enabling the **Multi-Core Debugging** checkbox and changing the **Core Index** selection.
9. Select **Project** > **Run**.

   The IDE downloads the program to the specified core. You can begin debugging.

JTAG Initialization File

Although you may debug only one single chip at a time, you must create a JTAG initialization file that specifies the type and order of all the chips in the chain.

To specify DSP56800 chips, you must specify DSP56800 as the name of a the chip you are debugging. For example, Listing 9.1 on page 248 shows a JTAG initialization file for three 56800 chips, an SC140 and an MCore210 in a JTAG chain.

NOTE Device 0 is the device closest to the TDO signal on the Command Converter Server.

Listing 9.1 Example JTAG Initialization File for DSP56800, SC140 and MCore210 Boards

```
# JTAG Initialization File

# Has an index value of 0 in the JTAG chain
DSP56800
# Has an index value of 1 in the JTAG chain
DSP56800
# Has an index value of 2 in the JTAG chain
DSP56800
# Has an index value of 3 in the JTAG chain
SC140
# Has an index value of 4 in the JTAG chain
MCore210
```

NOTE See the sample initialization file in the `DSP56800x_EABI_Tools/JTAG` folder.

In addition, you can specify other chips to debug on the JTAG chain. To do so, you use the following syntax to specify the chip as a generic device:

```
Generic instruct_reg_length data_reg_bypass_length
JTAG_bypass_instruction
```

on page 249Table 9.10 on page 248 shows the definitions of the variables that you must specify for a generic device.

Table 9.10 Syntax Variables to Specify a Generic Device on a JTAG Chain

Variable	Description
`instruct_reg_length`	Length in bits of the JTAG instruction register.
`data_reg_bypass_length`	Length in bits of the JTAG bypass register.
`JTAG_bypass_instruct`	Hexadecimal value that specifies the JTAG bypass instruction.

Listing 9.2 on page 249 shows a JTAG initialization file that includes a DSP56800 chip and a generic device in a JTAG chain.

Listing 9.2 Example JTAG Initialization File with a Generic Device

```
# JTAG Initialization File

# Has an index value of 0 in the JTAG chain
DSP56800
# Has an index value of 1 in the JTAG chain
Generic 4 1 Oxf
```

Debugging in the Flash Memory

The debugger is capable of programming flash memory. The programming occurs at launch, during download. The flash programming option is turned on and the parameters are set in the initialization file. This file is specified in the **Debugger>M56800 Target** preference panel. A list of flash memory commands is given in the next section.

The stationery provides an example of how to specify a default initialization file, how to write a linker command file for flash memory, and how to copy initialized data from ROM to RAM using provided library functions.

Flash Memory Commands

The following is a list of flash memory commands that can be included in your initialization file.

set_hfmclkd <value>

This command writes the `value` which represents the clock divider for the flash memory to the hfmclkd register.

The value for the `set_hfmclkd` command depends on the frequency of the clock. If you are using a supported EVM, this value should not be changed from the value provided in the default initialization file. However, if you are using an unsupported board and the clock frequency is different from that of the supported EVM, a new value must be calculated as described in the user's manual of the particular processor that you are using.

NOTE The `set_hfmclkd`, `set_hfm_base`, and at least one `add_hfm_unit` command must exist to enable flash programming. All other flash memory commands are optional.

set_hfm_base <address>

This command sets the address of `hfm_base`, which is where the flash control registers are mapped in X: memory.

NOTE The `set_hfm_base` and `add_hfm_unit` commands should not be changed for a particular processor. Their values will always be the same.

set_hfm_config_base <address>

This command sets the address of `hfm_config_base`, which is where the flash security values are written in program flash memory. If this command is present, the debugger used the address to mimic part of the hardware reset behavior by copying the protection values from the configuration field to the appropriate flash control registers.

add_hfm_unit <startAddr> <endAddr> <bank> <numSectors> <pageSize> <progMem> <boot> <interleaved>

This command adds a flash unit to the list and sets its parameters.

NOTE The `set_hfm_base` and `add_hfm_unit` commands should not be changed for a particular processor. Their values will always be the same.

set_hfm_erase_mode units | pages | all

This command sets the erase mode as `units`, `pages` or `all`. If you set this to `units`, the units that are programmed are mass erased. If set this to `pages`, the pages that are programmed are erased. If you set this to `all`, all units are mass erased including those that have not been programmed. If you omit this command, the erase mode defaults to the unit mode.

set_hfm_verify_erase 1 | 0

If you set this to `1`, the debugger verifies that the flash memory has been erased, and alerts you if the erase failed. If this command is omitted, the flash erase is not verified.

set_hfm_verify_program 1 | 0

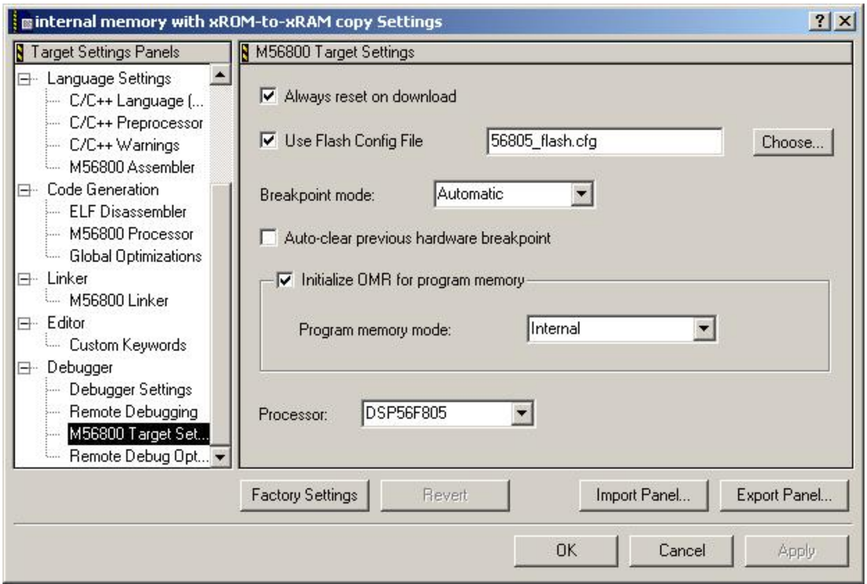
If you set this to `1`, the debugger verifies that the flash has been programmed correctly, and alerts you if the programming failed. If you omit this command, flash programming is not verified.

Setting up the Debugger for Flash Programming

In order for the debugger to download into Flash, the **Use Flash Config File** option is required in the M56800 Target panel and must be enabled.

Figure 9.35 on page 252 shows the **M56800 Target** panel when you use minimum requirements for Flash programming.

Figure 9.35 M56800 Target Panel for Programming Flash

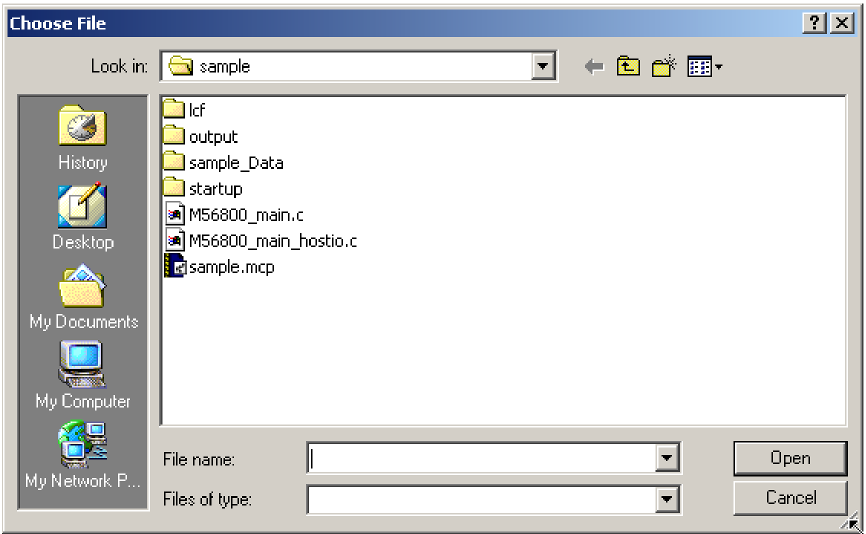


Use Flash Config File

When the **Use Flash Config File** option is enabled, you can specify the use of a flash configuration file ([Listing 5.3 on page 98]) in the text box. If the full path and file name are not specified, the default location is the same as the project file.

You can click the **Choose** button to specify the file. The **Choose File** dialog box appears ([Figure 9.36 on page 253]).

Figure 9.36 Choose File Dialog Box



For more information on the Flash Configuration File Line Format, see "M56800 Target (Debugging)".

Notes for Debugging on Hardware

Below are some tips and somethings to be aware of when debugging on a hardware target:

- Ensure your Flash data size fits into Flash memory.

  The linker command file specifies where data is written to. There is no bounds checking for Flash programming.

- The standard library I/O function such as `printf` uses large amount of memory and may not fit into flash targets.

- Use the Flash stationery when creating a new project intended for ROM.

  The default stationery contains the Flash configuration file and debugger settings required to use the Flash programmer.

- There is only one hardware breakpoint available, which is shared by IDE breakpoints (when the Breakpoint Mode is set to hardware in the **M56800 Target** panel), watchpoints, and OnCE triggers. Only one of these may be set at a time.

Flash Programming the Reset and Interrupt Vectors

The first four P: (program) memory locations in Flash ROM are actually "mirrored" from the first four memory locations of Boot Flash. Therefore, when Flash programming the reset vectors, write the reset vectors to the beginning of Boot Flash. The interrupt vectors are located in Program Flash. Write the interrupt vectors normally, starting at P:0x0004. The Flash targets in the stationery demonstrate how the source, linker command file, and flash configuration file look.

NOTE It is important that you use the flash configuration file provided in the stationery. Using a flash configuration file with extra sections can lead to multiple erases of the same flash unit resulting in Flash programming errors.

10

Data Visualization

Data visualization lets you graph variables, registers, and regions of memory as they change over time.

The Data Visualization tools can plot memory data, register data, and global variable data.

- Starting Data Visualization on page 255
- Data Target Dialog Boxes on page 256
- Graph Window Properties on page 259

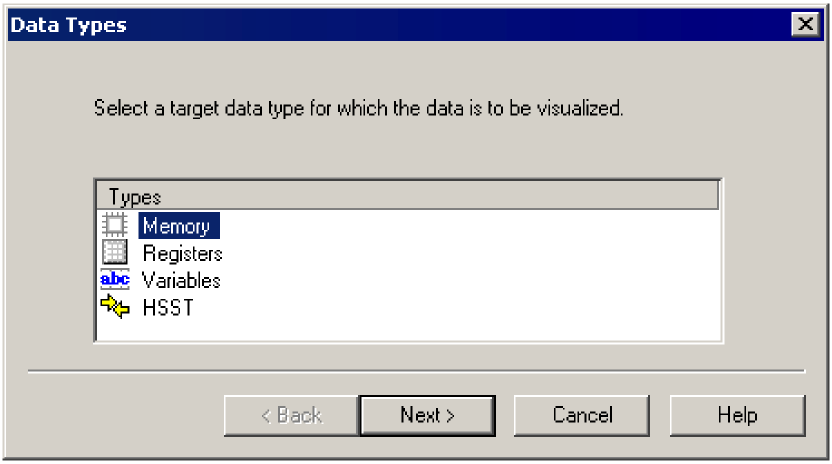
Starting Data Visualization

To start the Data Visualization tool:

1. Start a debug session
2. Select Data Visualization > Configurator.

   The Data Types window (Figure 10.1 on page 255) appears. Select a data target type and click the Next button.

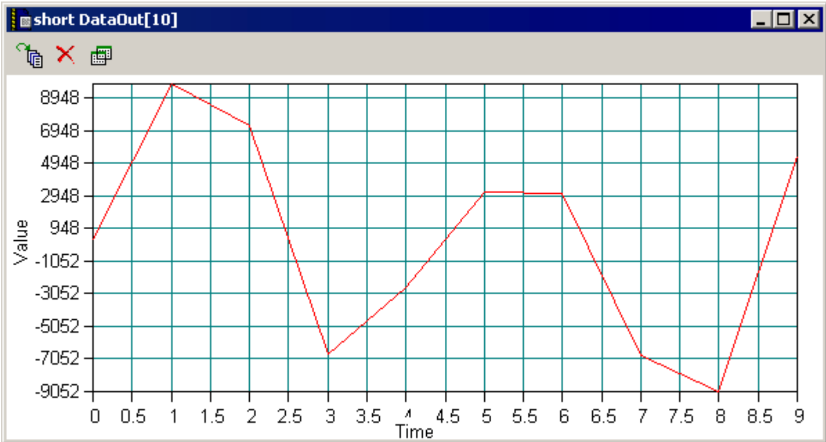
Figure 10.1 Data Types Window



3. Configure the data target dialog box and filter dialog box.

4. Run your program to display the data (Figure 10.2 on page 256).

Figure 10.2 Graph Window



Data Target Dialog Boxes

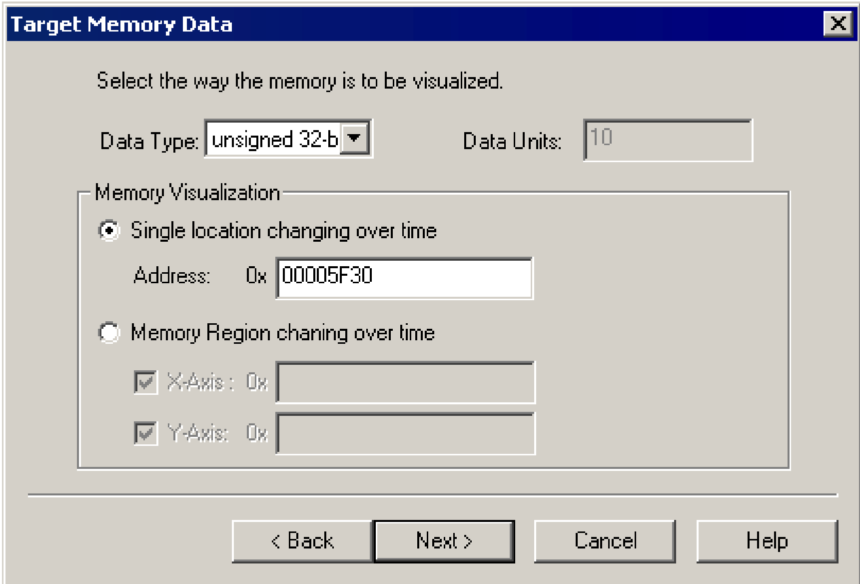
There are four possible data targets. Each target has its own configuration dialog.

- Memory on page 256
- Registers on page 258
- Variables on page 258

Memory

The Target Memory dialog box lets you graph memory contents in real-time.

Figure 10.3 Target Memory Dialog Box



Data Type

The Data Type list box lets you select the type of data to be plotted.

Data Unit

The Data Units text field lets you enter a value for number of data units to be plotted. This option is only available when you select Memory Region Changing Over Time.

Single Location Changing Over Time

The Single Location Changing Over Time option lets you graph the value of a single memory address. Enter this memory address in the Address text field.

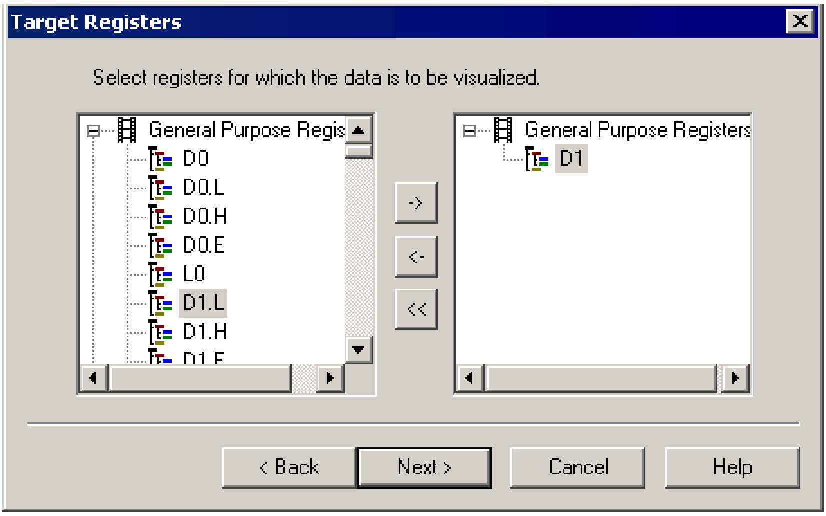
Memory Region Changing Over Time

The Memory Region Changing Over Time options lets you graph the values of a memory region. Enter the memory addresses for the region in the X-Axis and Y-Axis text fields.

Registers

The Target Registers dialog box lets you graph the value of registers in real-time.

Figure 10.4 Target Registers Dialog Box

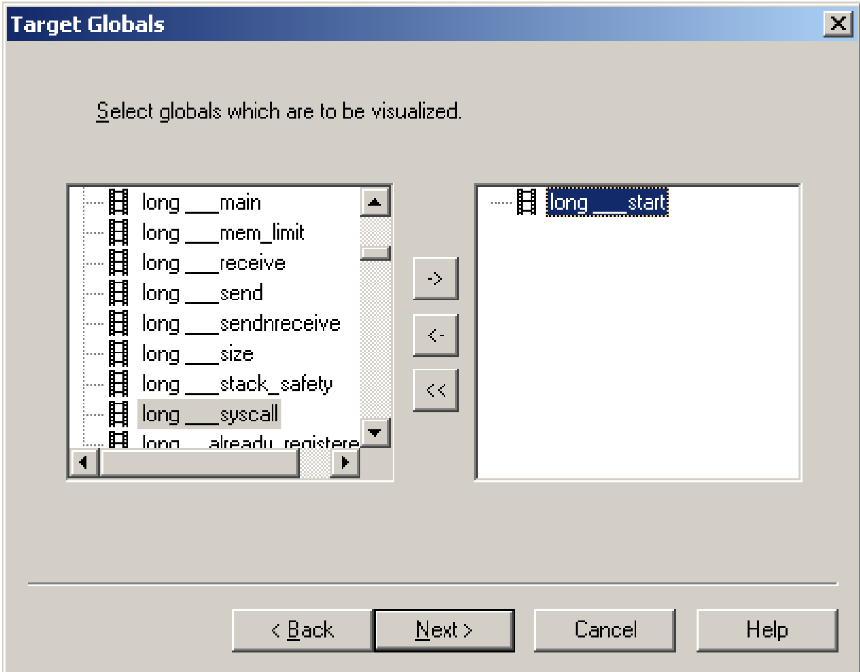


Select registers from the left column, and click the -> button to add them to the list of registers to be plotted.

Variables

The Target Globals dialog box lets you graph the value of global variables in real-time. (See Figure 10.5 on page 259.)

Figure 10.5 Target Globals Dialog Box

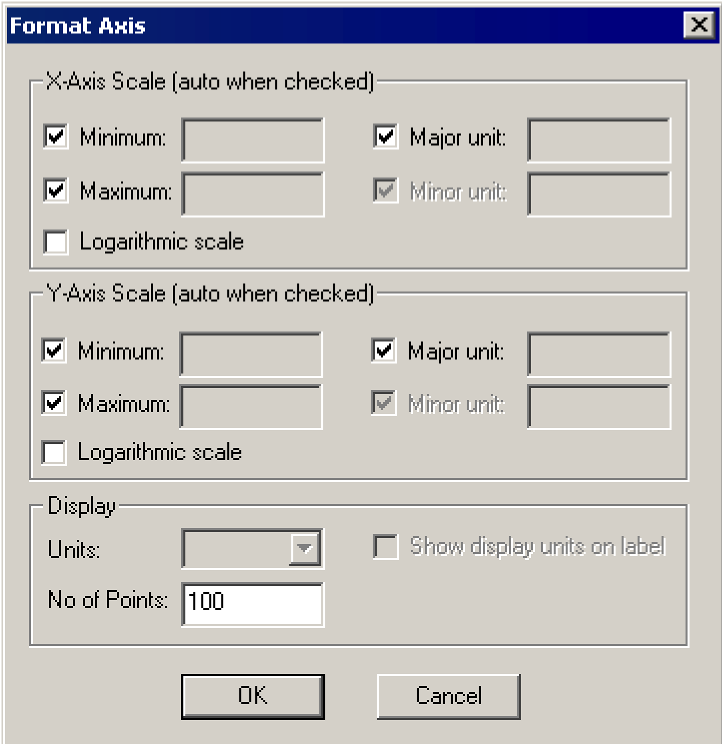


Select global variables from the left column, and click the -> button to add them to the list of variables to be plotted.

Graph Window Properties

To change the look of the graph window, click the  graph properties button to open the Format Axis dialog box.

Figure 10.6 Format Axis Dialog Box



Scaling

The default scaling settings of the data visualization tools automatically scale the graph window to fit the existing data points.

To override the automatic scaling, uncheck a scaling checkbox to enable the text field and enter your own value.

To scale either axis logarithmically, enable the Logarithmic Scale option of the corresponding axis.

Display

The Display settings let you change the maximum number of data points that are plotted on the graph.

11

Profiler

The profiler is a run-time feature that collects information about your program. It records the minimum, maximum, and total number of clock cycles spent in each function. The profiler allows you to evaluate your code and determine which functions require optimization.

When profiling is enabled, the compiler adds code to call the entry functions in the profiler library. These profiler library functions do all of the data collection. The profiler library, with the help of the debugger create a binary output file, which is opened and displayed by the CodeWarrior IDE.

NOTE For more information on the profiler library and its usage, see the *CodeWarrior Development Studio IDE 5.5 User's Guide Profiler Supplement*.

To enable your project for profiling:

1. Add the following path to your list of user paths in the Access Paths settings panel:

   ```
   {Compiler}M56800x Support\profiler
   ```

2. Add the following line to the file that contains the function main():

   ```
   #include "Profiler.h"
   ```

3. Add the profiler library file to your project. Select the library that matches your target from this path:

   ```
   {CodeWarrior path}M56800x Support\profiler\lib
   ```

4. Add the following function calls to main():

   ```
   ProfilerInit()
   ProfilerClear()
   ProfilerSetStatus()
   ProfilerDump()
   ProfilerTerm()
   ```

   For more details of these functions, see the *CodeWarrior Development Studio IDE 5.5 User's Guide Profiler Supplement*.

5. It may be necessary to increase the heap size to accommodate the profiler data collection. This can be set in the linker command file by changing the value of __heap_size.

6. Enable profiling by setting the **Generate code for profiling** option in the **M56800 Processor** settings panel or by using the profile on | off pragma to select individual functions to profile.

NOTE For a profiler example, see the profiler example in this path: `{CodeWarrior path}(CodeWarrior_Examples)\SimpleProfiler`

12

ELF Linker

The CodeWarrior™ Executable and Linking Format (ELF) Linker makes a program file out of the object files of your project. The linker also allows you to manipulate code in different ways. You can define variables during linking, control the link order to the granularity of a single function, change the alignment, and even compress code and data segments so that they occupy less space in the output file.

All of these functions are accessed through commands in the linker command file (LCF). The linker command file has its own language complete with keywords, directives, and expressions, that are used to create the specifications for your output code. The syntax and structure of the linker command file is similar to that of a programming language.

This chapter contains the following sections:

Structure of Linker Command Files

Linker command files contain three main segments:

- Memory Segment
- Closure Blocks
- Sections Segment

A command file must contain a memory segment and a sections segment. Closure segments are optional.

Memory Segment

In the memory segment, available memory is divided into segments. Listing 12.1 on page 263 shows a sample memory-segment format.

Listing 12.1 Sample MEMORY Segment

```
MEMORY {
```

```
    segment_1 (RWX): ORIGIN = 0x1000, LENGTH = 0x1000
    segment_2 (RWX): ORIGIN = AFTER(segment_1), LENGTH = 0
    data      (RW) : ORIGIN = 0x2000, LENGTH = 0x0000
    #segment_name (RW) : ORIGIN = memory address, LENGTH = segment
    #length
    #and so on...
}
```

The (RWX) portion consists of ELF access permission flags, **r**ead, **w**rite, and e**x**ecute where:

- ORIGIN represents the start address of the memory segment.
- LENGTH represents the maximum size allowed for the memory segment.

Memory segments with RWX attributes are placed into P memory while RW attributes are placed into X memory.

Memory segments with R attributes denote X ROM memory, and memory segments with RX attributes denote P ROM memory.

You can put a segment immediately after the previous one using the AFTER command.

If you cannot predict how much space a segment will occupy, you can use the command LENGTH = 0 (unlimited length) and let the linker figure out the size of the segment.

Closure Blocks

The linker is very good at deadstripping unused code and data. Sometimes, however, symbols need to be kept in the output file even if they are never directly referenced. Interrupt handlers, for example, are usually linked at special addresses, without any explicit jumps to transfer control to these places.

Closure blocks provide a way to make symbols immune from deadstripping. The closure is transitive, meaning that symbols referenced by the symbol being closed are also forced into closure, as are any symbols referenced by those symbols, and so on.

NOTE The closure blocks need to be in place before the SECTIONS definition in the linker command file.

The two types of closure blocks available are:

- Symbol-level

  Use FORCE_ACTIVE to include a symbol into the link that would not be otherwise included. An example is in Listing 12.2 on page 265.

Listing 12.2 Sample Symbol-level Closure Block

```
FORCE_ACTIVE {break_handler, interrupt_handler, my_function}
```

- Section-level

  Use `KEEP_SECTION` when you want to keep a section (usually a user-defined section) in the link. Listing 12.3 on page 265 is an example.

Listing 12.3 Sample Section-level Closure Block

```
KEEP_SECTION {.interrupt1, .interrupt2}
```

A variant is `REF_INCLUDE`. It keeps a section in the link, but only if the file where it is coming from is referenced. This is very useful to include version numbers. Listing 12.4 on page 265 is an example.

Listing 12.4 Sample Section-level Closure Block With File Dependency

```
REF_INCLUDE {.version}
```

Sections Segment

In the Sections segment, you define the contents of memory segments and any global symbols to be used in the output file.

The format of a typical sections block is in Listing 12.5 on page 265.

Listing 12.5 Sample SECTIONS Segment

```
SECTIONS {
   .section_name : #the section name is for your reference
   {               #the section name must begin with a '.'
      filename.c  (.text) #put the .text section from filename.c
      filename2.c (.text) #then the .text section from filename2.c
      filename.c  (.data)
      filename2.c (.data)
      filename.c  (.bss)
      filename2.c (.bss)
      . = ALIGN (0x10);   #align next section on 16-byte boundary.
   } > segment_1     #this means "map these contents to segment_1"

   .next_section_name:
   {
      more content descriptions
   } > segment_x      # end of .next_section_name definition
```

```
}                          # end of the sections block
```

Linker Command File Syntax

This section explains some practical ways in which to use the commands of the linker command file to perform common tasks.

Alignment

To align data on a specific word-boundary, you use the ALIGN and ALIGNALL commands to bump the location counter to the preferred boundary. For example, the following fragment uses `ALIGN` to bump the location counter to the next 16-byte (word) boundary. A sample is in Listing 12.6 on page 266.

Listing 12.6 Sample ALIGN Command Usage

```
file.c (.text)
. = ALIGN (0x10);
file.c (.data)    # aligned on a 16-byte (word) boundary.
```

You can also align data on a specific word-boundary with `ALIGNALL as shown in` (Listing 12.7 on page 266).

Listing 12.7 Sample ALIGNALL Command Usage

```
file.c (.text)
ALIGNALL (0x10);  #everything past this point aligned on 16 bytes
(word)
file.c (.data)
```

Arithmetic Operations

Standard C arithmetic and logical operations may be used to define and use symbols in the linker command file. Table 12.1 on page 267 shows the order of precedence for each operator. All operators are left-associative.

Table 12.1 Arithmetic Operators

Precedence	Operators
1 (highest)	`- ˜ !`
2	`* / %`
3	`+ -`
4	`>> <<`
5	`== != > < <= >=`
6	`&`
7	`|`
8	`&&`
9	`||`

NOTE The shift operator shifts two-bits for each shift operation. The divide operator performs division and rounding.

Comments

Add comments by using the pound character (#) or C++ style double-slashes (//). C-style comments are not accepted by the LCF parser. Listing 12.8 on page 267 shows examples of valid comments.

Listing 12.8 Example Comments

```
#  This is a one-line comment
* (.text) // This is a partial-line comment
```

Deadstrip Prevention

The M56800 linker removes unused code and data from the output file. This process is called deadstripping. To prevent the linker from deadstripping unreferenced code and data, use the `FORCE_ACTIVE`, `KEEP_SECTION`, `and` `REF_INCLUDE` directives to preserve them in the output file.

Variables, Expressions and Integral Types

This section explains variables, expressions, and integral types.

Variables and Symbols

All symbol names within a Linker Command File (LCF) start with the underscore character (_), followed by letters, digits, or underscore characters. Listing 12.9 on page 268 shows examples of valid lines for a command file:

Listing 12.9 Valid Command File Lines

```
_dec_num = 99999999;
_hex_num_ = 0x9011276;
```

Variables that are defined within a `SECTIONS` section can only be used within a `SECTIONS` section in a linker command file.

Global Variables

Global variables are accessed in a linker command file with an 'F' prepended to the symbol name. This is because the compiler adds an 'F' prefix to externally defined symbols.

Listing 12.10 on page 268 shows an example of using a global variable in a linker command file. This example sets the global variable `_foot`, declared in C with the `extern` keyword, to the location of the address location current counter.

Listing 12.10 Using a Global Variable in the LCF

```
F_foot = .;
```

If you use a global symbol in an LCF, as in Listing 12.10 on page 268, it can be accessed from C program sources as shown in Listing 12.11 on page 268.

Listing 12.11 Accessing a Global Symbol From C Program Sources

```
extern unsigned long _foot;
void main( void ) {
  unsigned long i;
  // ...
  i = _foot;    // _foot value determined in LCF
  // ...
}
```

Expressions and Assignments

You can create symbols and assign addresses to those symbols by using the standard assignment operator. An assignment may only be used at the start of an expression, and a semicolon is required at the end of an assignment statement. An example of standard assignment operator usage is shown in Listing 12.12 on page 269.

Listing 12.12 Standard Assignment Operator Usage

```
_symbolicname = some_expression; # Legal
_sym1 + _sym2 =  sym3;           # ILLEGAL!
```

When an expression is evaluated and assigned to a variable, it is given either an absolute or a relocatable type. An absolute expression type is one in which the symbol contains the value that it will have in the output file. A relocatable expression is one in which the value is expressed as a fixed offset from the base of a section.

Integral Types

The syntax for linker command file expressions is very similar to the syntax of the C programming language. All integer types are `long` or `unsigned long`.

Octal integers (commonly know as base eight integers) are specified with a leading zero, followed by numeral in the range of zero through seven. Listing 12.13 on page 269 shows valid octal patterns you could put into your linker command file.

Listing 12.13 Sample Octal Patterns

```
_octal_number  = 012;
_octal_number2 = 03245;
```

Decimal integers are specified as a non-zero numeral, followed by numerals in the range of zero through nine. To create a negative integer, use the minus sign (-) in front of the number. Listing 12.14 on page 269 shows examples of valid decimal integers that you could write into your linker command file.

Listing 12.14 Sample Decimal Integers

```
_dec_num       = 9999;
_decimalNumber = -1234;
```

Hexadecimal (base sixteen) integers are specified as `0x` or `0X` (a zero with an X), followed by numerals in the range of zero through nine, and/or characters `A` through `F`.

Examples of valid hexadecimal integers you could put in your linker command file appear in Listing 12.15 on page 270.

Listing 12.15 Example Hexadecimal Integers

```
_somenumber        = 0x0F21;
_fudgefactorspace = 0XF00D;
_hexonyou          = 0xcafe;
```

File Selection

When defining the contents of a `SECTION` block, specify the source files that are contributing to their sections. The standard method of doing this is to list the files.

In a large project, the list can grow to become very long. For this reason, use the asterix (`*`) keyword. The asterix (`*`) keyword represents the filenames of every file in your project. Note, that since you have already added the `.text` sections from the files `main.c`, `file2.c`, and `file3.c`, the '`*`' keyword does not include the `.text` sections from those files again.

Function Selection

The OBJECT keyword allows precise control over how functions are placed within a section. For example, if the functions `pad` and `foot` are to be placed before anything else in a section, use code like the example in Listing 12.16 on page 270.

Listing 12.16 Sample Function Selection Using the Object Keyword

```
SECTIONS {
  .program_section :
    {
      OBJECT (Fpad, main.c)
      OBJECT (Ffoot, main.c)
      * (.text)
     } > ROOT
```

NOTE If an object is written once using the **Object** function selection keyword, you can prevent the same object from being written again using the '`*`' file selection keyword.

ROM to RAM Copying

In embedded programming, it is common to copy a portion of a program resident in ROM into RAM at runtime. For example, program variables cannot be accessed until they are copied to RAM.

To indicate data or code that is meant to be copied from ROM to RAM, the data or code is given two addresses. One address is its resident location in ROM (defined by the linker command file). The other is its intended location in RAM (defined in C code where you do the actual copying).

To create a section with the resident location in ROM and an intended location in RAM, you define the two addresses in the linker command file. Use the `MEMORY` segment to specify the intended RAM location, and the `AT(`*`address`*`)` parameter to specify the resident ROM address.

NOTE This method only works for copying from data ROM to data RAM.

For example, you have a program and you want to copy all your initialized data into RAM at runtime. Listing 12.17 on page 271shows you the LCF used to set up for writing initialized data to ROM.

NOTE If you want to write initialized data to program ROM, use the `WRITE` commands in the LCF. Also, write your own P to X memory copy routine in assembly to copy data from program ROM to data RAM at runtime.

Listing 12.17 LCF File to Prepare Data Copy From ROM to RAM

```
MEMORY {
  .text (RWX) : ORIGIN = 0x8000, LENGTH = 0x0    # code (P)
  .data (RW)  : ORIGIN = 0x3000, LENGTH = 0x0    # data (X)-> RAM
}

SECTIONS{

  F__ROM_Address = 0x1000;        # ROM Starting Address

  .main_application :
  {
    # .text sections

    *(.text)
    *(.rtlib.text)
    *(.fp_engine.txt)
    *(user.text)
  } > .text
```

```
  .data : AT( F__ROM_Address )  # Start data at 0x1000 -> ROM
  {
    # .data sections
    F_Begin_Data = .;           # Get start location for RAM
    *(.data)                    # Write data to the section (ROM)
    *(fp_state.data);
    *(rtlib.data);
    F_End_Data = .;             # Get end location for RAM

    # .bss sections
    * (rtlib.bss.lo)
    * (.bss)

  } > .data
}
```

To make the runtime copy the section from ROM to RAM, you need to know where the data start in ROM (`__ROM_Address`) and the size of the block in ROM you want to copy to RAM. In Listing 12.18 on page 272, all variables in the data section from ROM to RAM in C code are copied.

Listing 12.18 ROM to RAM Copy From C After Data-Flash Write

```
#include <stdio.h>
#include <string.h>

int GlobalFlash = 6;

// From linker command file
extern __Begin_Data, __ROMAddress, __End_Data;

void main( void )
{
  unsigned short a = 0, b = 0, c = 0;
  unsigned long dataLen  = 0x0;
  unsigned short __myArray[] = { 0xdead, 0xbeef, 0xcafe };

  // Calculate the data length of the X memory written to Flash
  dataLen = (unsigned long)&__End_Data -
             (unsigned long)&__Begin_Data;

  // Block move from ROM to RAM
  memcpy( (unsigned long *)&__Begin_Data,
           (const unsigned long *)&__ROMAddress,
           dataLen );
```

```
  a = GlobalFlash;

  return;
}
```

NOTE For this example to work, you must be writing to Flash with the CodeWarrior debugger and have your board jumpered to mode 0.

Stack and Heap

To reserve space for the stack and heap, arithmetic operations are performed to set the values of the symbols used by the runtime.

The Linker Command File (LCF) performs all the necessary stack and heap initialization. When Stationery is used to create a new project, the appropriate LCFs are added to the new project.

See any Stationery-generated LCFs for examples of how stack and heap are initialized.

Writing Data Directly to Memory

You can write directly to memory using the `WRITEx` command in the linker command file. The `WRITEB` command writes a byte, the `WRITEH` command writes two bytes, and the `WRITEW` command writes four bytes. You insert the data at the section's current address.

Listing 12.19 Embedding Data Directly Into the Output

```
.example_data_section :
{
   WRITEB 0x48;  //  'H'
   WRITEB 0x69;  //  'i'
   WRITEB 0x21;  //  '!'
}
```

Linker Command File Keyword Listing

This sections explains the keywords available for use when creating CodeWarrior Development Studio for Freescale 56800 applications with the linker command file. Valid linker command file functions, keywords, directives, and commands are described:

- . (location counter) on page 274

. (location counter)

The period character (`.`) always maintains the current position of the output location. Since the period always refers to a location in a `SECTIONS` block, it can not be used outside a section definition.

A period may appear anywhere a symbol is allowed. Assigning a value to period that is greater than its current value causes the location counter to move, but the location counter can never be decremented.

This effect can be used to create empty space in an output section. In the example below, the location counter is moved to a position that is 0x1000 words past the symbol `FSTART_`.

Example

```
.data :
{
      *(.data)
      *(.bss)
      FSTART_ = .;
      . = FSTART_ + 0x1000;
      __end = .;
} > DATA
```

ADDR

The `ADDR` function returns the address of the named section or memory segment.

Prototype

`ADDR (sectionName | segmentName)`

In the example below, `ADDR` is used to assign the address of `ROOT` to the symbol `__rootbasecode`.

Example

```
MEMORY{
     ROOT  (RWX) : ORIGIN = 0x8000, LENGTH = 0
}

SECTIONS{
  .code :
  {
  __rootbasecode = ADDR(ROOT);
      *(.text);
  } > ROOT
}
```

ALIGN

The `ALIGN` function returns the value of the location counter aligned on a boundary specified by the value of `alignValue`. The `alignValue` must be a power of two.

Prototype

```
ALIGN(alignValue)
```

Please note that `ALIGN` does not update the location counter; it only performs arithmetic. To update the location counter, use an assignment such as the following:

Example

```
. = ALIGN(0x10);     #update location counter to 16
                     #byte alignment
```

ALIGNALL

`ALIGNALL` is the command version of the ALIGN function. It forces the minimum alignment for all the objects in the current segment to the value of `alignValue`. The `alignValue` must be a power of two.

Prototype

```
ALIGNALL(alignValue);
```

Unlike its counterpart ALIGN, ALIGNALL is an actual command. It updates the location counter as each object is written to the output.

Example

```
.code :
{
  ALIGNALL(16);  // Align code on 16 byte boundary
   *     (.init)
   *     (.text)

  ALIGNALL(16);  //align data on 16 byte boundary
   *     (.rodata)
} > .text
```

FORCE_ACTIVE

The `FORCE_ACTIVE` directive allows you to specify symbols that you do not want the linker to deadstrip. You must specify the symbol(s) you want to keep before you use the `SECTIONS` keyword.

Prototype

```
FORCE_ACTIVE{ symbol[, symbol] }
```

INCLUDE

The `INCLUDE` command allows you to include a binary file in the output file.

Prototype

```
INCLUDE filename
```

KEEP_SECTION

The `KEEP_SECTION` directive allows you to specify sections that you do not want the linker to deadstrip. You must specify the section(s) you want to keep before you use the `SECTION` keyword.

Prototype

```
KEEP_SECTION{ sectionType[, sectionType] }
```

MEMORY

The `MEMORY` directive allows you to describe the location and size of memory segment blocks in the target. This directive specifies the linker the memory areas to avoid, and the memory areas into which it links the code and date.

The linker command file may only contain one `MEMORY` directive. However, within the confines of the `MEMORY` directive, you may define as many memory segments as you wish.

Prototype

`MEMORY { memory_spec }`

The `memory_spec` is:

```
segmentName (accessFlags) : ORIGIN = address, LENGTH = length
     [,COMPRESS] [> fileName]
```

`segmentName` can include alphanumeric characters and underscore '_' characters.

`accessFlags` are passed into the output ELF file (`Phdr.p_flags`). The `accessFlags` can be:

- R-read
- W-write
- X-executable (for P memory placement)

`address origin` is one of the following:

- **Memory address**

  Specify a hex address, such as `0x8000`.

- **AFTER command**

  Use the `AFTER(`*`name [,name]`*`)` command to instruct the linker to place the memory segment after the specified segment. In the example below, `overlay1` and `overlay2` are placed after the code segment. When multiple memory segments are specified as parameters for `AFTER`, the highest memory address is used.

Example

```
MEMORY{
code      (RWX)   : ORIGIN = 0x8000,      LENGTH = 0
overlay1  (RWX)   : ORIGIN = AFTER(code), LENGTH = 0
overlay2  (RWX)   : ORIGIN = AFTER(code), LENGTH = 0
data      (RW)    : ORIGIN = 0x1000,      LENGTH = 0
}
```

`ORIGIN` is the assigned address.

`LENGTH` is any of the following:

- A value greater than zero.

  If you try to put more code and data into a memory segment greater than your specified length allows, the linker stops with an error.

- Autolength by specifying zero.

When the length is 0, the linker lets you put as much code and data into a memory segment as you want.

NOTE There is no overflow checking with autolength. The linker can produce an unexpected result if you use the autolength feature without leaving enough free memory space to contain the memory segment. Using the `AFTER` keyword to specify origin addresses prevents this.

`> fileName` is an option to write the segment to a binary file on disk instead of an ELF program header. The binary file is put in the same folder as the ELF output file. This option has two variants:

- `> fileName`

  Writes the segment to a new file.

- `>> fileName`

  Appends the segment to an existing file.

OBJECT

The `OBJECT` keyword allows control over the order in which functions are placed in the output file.

Prototype

```
OBJECT (function, sourcefile.c)
```

It is important to note that if an object is written to the outfile using the `OBJECT` keyword, the IDE does not allow the same object to be written again by using the '*' wildcard selector.

REF_INCLUDE

The `REF_INCLUDE` directive allows you to specify sections that you do not want the linker to deadstrip, but only if they satisfy a certain condition: the file that contains the section must be referenced. This is useful if you want to include version information from your source file components. You must specify the section(s) you want to keep before you use the `SECTIONS` keyword.

Prototype

```
REF_INCLUDE{ sectionType [, sectionType]}
```

SECTIONS

A basic `SECTIONS` directive has the following form:

Prototype

```
SECTIONS { <section_spec> }
```

`section_spec` is one of the following:

```
sectionName : [AT (loadAddress)] {contents}  >
      segmentName
sectionName : [AT (loadAddress]] {contents} >>
      segmentName
```

Table 12.2 Sections Directive

`sectionName`	The section name for the output section. It must start with a period character. For example, `.mysection`.
`AT (loadAddress)`	An optional parameter that specifies the address of the section. The default (if not specified) is to make the load address the same as the relocation address.
`contents`	Made up of statements.

These statements can:

- assign a value to a symbol.
- describe the placement of an output section, including which input sections are placed into it.

`segmentName` is the predefined memory segment into which you want to put the contents of the section. The two variants are:

Table 12.3 segmentName Variants

`> segmentName`	Places the section contents at the beginning of the memory segment `segmentName`.
`>> segmentName`	Appends the section contents to the memory segment `segmentName`.

Here is an example section definition:

Example

```
SECTIONS {
  .text : {
            F_textSegmentStart = .;
            footpad.c (.text)
            . = ALIGN (0x10);
            padfoot.c (.text)
            F_textSegmentEnd = .;
  }
  .data : { *(.data) }
  .bss  : { *(.bss)
            *(COMMON)
  }
}
```

SIZEOF

The `SIZEOF` function returns the size of the given segment or section. The return value is the size in bytes.

Prototype

```
SIZEOF(segmentName | sectionName)
```

SIZEOFW

The `SIZEOFW` function returns the size of the given segment or section. The return value is the size in words.

Prototype

```
SIZEOFW(segmentName | sectionName)
```

WRITEB

The `WRITEB` command inserts a byte of data at the current address of a section.

Prototype

`WRITEB (expression);`

`expression` is any expression that returns a value `0x00` to `0xFF`.

WRITEH

The `WRITEH` command inserts two bytes of data at the current address of a section.

Prototype

`WRITEH (expression);`

`expression` is any expression that returns a value `0x0000` to `0xFFFF`.

WRITES

The `WRITES` command is a string of variables with maximum length of 255 characters.

You can use `DATE` and `TIME` in conjunction with the `WRITES` command.

`DATE` returns the current date as a C string (must be within parentheses).

`TIME` returns the current time as a C string (must be within parentheses).

Prototype

`WRITES (`*`string`*`);`

`string` is any string within parentheses.

Examples

```
WRITES ("Hello World").
WRITES ("Today is" DATE).
WRITES ("The time is " TIME).
```

WRITEW

The `WRITEW` command inserts 4 bytes of data at the current address of a section.

Prototype

`WRITEW (expression);`

`expression` is any expression that returns a value `0x00000000` to `0xFFFFFFFF`.

Sample M56800 Linker Command File

A sample M56800 linker command file is in Listing 12.20 on page 283. This is the typical linker command file.

Listing 12.20 Sample Linker Command File (DSP56805EVM)

```
# ---------------------------------------------------

# Metrowerks, a company of Freescale
# sample code

# linker command file for DSP56805EVM
# using
#     external pRAM
#     external xRAM
#     internal xRAM (0x30-40 for compiler regs)
#         mode 3
#          EXT 0

# revision history
# 011020 R4.1 a.h. first version
# 030220 R5.1 a.h. improved comments

# ---------------------------------------------------

# see end of file for additional notes
# additional reference: Freescale docs

# for this LCF:
# interrupt vectors --> external pRAM starting at zero
#      program code --> external pRAM
#         constants --> external xRAM
```

```
#      dynamic data --> external xRAM

# stack size is set to 0x1000 for external RAM LCF

# requirements: Mode 3 and EX=0
# note -- there is a mode OB but any Reset or COP Reset
#         resets the memory map back to Mode 0A.


# DSP56805EVM eval board settings:
#    OFF --> jumper JG7 (mode 0 upon exit from reset)
#    ON  --> jumper JG8 (enable external board SRAM)

# CodeWarrior debugger Target option settings
#   OFF --> "Use Hardware Breakpoints"
#    ON --> "Debugger sets OMR at Launch" option

# note: with above option on, CW debugger sets OMR as
# OMR:
#     0 --> EX bit (stay in Debug processing state)
#     1 --> MA bit
#     1 --> MB bit


# 56805
# mode 3 (development)
# EX = 0

MEMORY
{
  .p_interrupts_RAM     (RWX) : ORIGIN = 0x0000, LENGTH = 0x0080
  .p_external_RAM       (RWX) : ORIGIN = 0x0080, LENGTH = 0x0000
  .x_compiler_regs_iRAM (RW)  : ORIGIN = 0x0030, LENGTH = 0x0010
  .x_internal_RAM       (RW)  : ORIGIN = 0x0040, LENGTH = 0x07C0
  .x_reserved           (R)   : ORIGIN = 0x0800, LENGTH = 0x0400
  .x_peripherals        (RW)  : ORIGIN = 0x0C00, LENGTH = 0x0400
  .x_flash_ROM          (R)   : ORIGIN = 0x1000, LENGTH = 0x1000
  .x_external_RAM       (RW)  : ORIGIN = 0x2000, LENGTH = 0xDF80
  .x_core_regs          (RW)  : ORIGIN = 0xFF80, LENGTH = 0x0080
}


# we ensure the interrupt vector section is not deadstripped here
```

```
KEEP_SECTION{ interrupt_vectors.text }


# place all executing code & data in external memory

SECTIONS {

  interrupt_vectors_for_p_ram :{      # from 56805_vector.asm
                                                         *
  (interrupt_vectors.text)

  } > .p_interrupts_RAM


  .executing_code :
                                                     {
  # .text sections

  * (.text)
  * (rtlib.text)
  * (fp_engine.text)
  * (user.text)
  } > .p_external_RAM


  .data :
  {
    # .data sections

    * (.const.data)
    * (fp_state.data)
    * (rtlib.data)
    * (.data)



    # .bss sections

    * (rtlib.bss.lo)

    __bss_start = .;

    * (.bss)
```

```
__bss_end   = .;
__bss_size = __bss_end - __bss_start;


# setup the heap address

__heap_addr = .;
__heap_size = 0x1000; # larger heap for hostIO
__heap_end = __heap_addr + __heap_size;

. = __heap_end;


# setup the stack address

_min_stack_size = 0x0200;
__stack_addr = __heap_end;
__stack_end  = __stack_addr + _min_stack_size;
. = __stack_end;

# set global vars

# MSL uses these globals:
F_heap_addr  = __heap_addr;
F_heap_end   = __heap_end;
F_stack_addr = __stack_addr;


# stationery init code globals

F_bss_size      = __bss_size;
F_bss_addr      = __bss_start;

# next not used in this LCF
# we define anyway so init code will link
# these can be removed with removal of rom-to-ram
# copy code in init file

F_data_size     = 0x0000;
F_data_RAM_addr = 0x0000;
F_data_ROM_addr = 0x0000;

F_rom_to_ram    = 0x0000; # zero is no rom-to-ram copy
```

```
    } > .x_external_RAM
}


# -------------------------------------------------------
# additional notes:

# about the reserved sections
# for this external RAM only LCF:

# p_interrupts_RAM -- reserved in external pRAM
# memory space reserved for interrupt vectors
# interrupt vectors must start at address zero
# interrupt vector space size is 0x80

# x_compiler_regs_iRAM -- reserved in internal xRAM
# The compiler uses page 0 address locations 0x30-0x40
# as register variables. See the Target manual for more info.


# notes:
# program memory (p memory)
# (RWX) read/write/execute for pRAM
# (RX) read/execute for flashed pROM

# data memory (X memory)
# (RW) read/write for xRAM
# (R)  read for data flashed xROM

# LENGTH = next start address - previous
# LENGTH = 0x0000 means use all remaining memory
```

13

Command-Line Tools

This chapter contains the following sections:

Usage

To call the command-line tools, use the following format:

Table 13.1 Format

Tools	File Names	Format
Compiler	mwcc56800.exe	compiler-options [linker-options] file-list
Linker	mwld56800.exe	linker-options file-list
Assembler	mwasm56800.exe	assembler-options file-list

The compiler automatically calls the linker by default and any options from the linker is passed on by the compiler to the assembler. However, you may choose to only compile with the `–c` flag. In this case, the assembler will only assemble and will not call the linker.

Also, available are environment variables. These are used to provide path information for includes or libraries, and to specify which libraries are to be included. You can specify the variables listed in Table 13.2 on page 290.

Table 13.2 Environment Variables

Tool	Library	Description
Compiler	MWCM56800Includes	Similar to Access Paths panel; separate paths with ‘;’ and prefix a path with ‘+’ to specify a recursive path
Linker	MW56800Libraries MW56800LibraryFiles	Similar to MWC56800Includes List of library names to link with project; separate with ‘;’
Assembler	MWAsm56800Includes	(similar to MWC56800Includes)

These are the target-specific variables, and will only work with the DSP56800 tools. The generic variables **MWCIncludes**, **MWLibraries**, **MWLibraryFiles**, and **MWAsmIncludes** apply to all target tools on your system (such as Windows). If you only have the DSP56800 tools installed, then you may use the generic variables if you prefer.

Response File

In addition to specifying commands in the argument list, you may also specify a “response file”. A response file’s filename begins with an ‘@’ (for example, @file), and the contents of the response file are commands to be inserted into the argument list. The response file supports standard UNIX-style comments. For example, the response file @file, contain the following:

```
# Response file @file
-o out.elf          # change output file name to 'out.elf'
-g                  # generate debugging symbols
```

The above response file can used in a command such as:

mwcc56800 @file main.c

It would be the same as using the following command:

mwcc56800 –o out.elf –g main.c

Sample Build Script

This following is a sample of a DOS batch (BAT) file. The sample demonstrates:

- Setting of the environmental variables.
- Using the compiler to compile and link a set of files.

```
      REM *** set GUI compiler path ***
      set COMPILER={path to compiler}

      REM *** set includes path ***
      set MWCIncludes=+%COMPILER%\M56800 Support
      set MWLibraries=+%COMPILER%\M56800 Support
      set MWLibraryFiles=MSL C 56800.lib;FP56800.lib

      REM *** add CLT directory to PATH ***
      set
PATH=%PATH%;%COMPILER%\DSP56800_EABI_Tools\Command_Line_Tools\

      REM *** compile options and files ***
      set COPTIONS=-O3
      set CFILELIST=file1.c file2.c
      set LOPTIONS=-m FSTART_ -o output.elf -g
      set LCF=linker.cmd

      REM *** compile, assemble and link ***
      mwcc56800 %COPTIONS% %CFILELIST%
      mwasm56800 %AFILELIST%
      mwld56800 %LOPTIONS% %LFILELIST% %LCF%
```

Arguments

General Command-Line Options

```
-----------------------------------------------------------------------
General Command-Line Options

   All the options are passed to the linker unless otherwise noted.

   Please see '-help usage' for details about the meaning of this help.
-----------------------------------------------------------------------
  -help [keyword[,...]]      # global; for this tool;
                             #  display help
```

```
     usage                  #   show usage information
     [no]spaces             #   insert blank lines between options in
                            #   printout
     all                    #   show all standard options
     [no]normal             #   show only standard options
     [no]obsolete           #   show obsolete options
     [no]ignored            #   show ignored options
     [no]deprecated         #   show deprecated options
     [no]meaningless           #   show options meaningless for this
                            #   target
     [no]compatible         #   show compatibility options
     opt[ion]=name             #   show help for a given option; for
                            #   'name',
                            #   maximum length 63 chars
     search=keyword            #   show help for an option whose name
                            #   or help
                            #   contains 'keyword' (case-sensitive);

                        #   for  'keyword', maximum length 63 chars
     group=keyword      #   show help for groups whose names contain
                        #   'keyword' (case-sensitive); for 'keyword'
                        #   maximum length 63 chars
     tool=keyword[,...] #   categorize groups of options by tool;
                        #   default
     all                #   show all options available in this tool
     this               #   show options executed by this tool
                        #   default
     other|skipped      #   show options passed to another tool
     both               #   show options used in all tools
                        #
                        #
     -version           #   global; for this tool;
                        #   show version, configuration,and build date
     -timing            #   global; collect timing statistics
     -progress          #   global; show progress and version
     -v[erbose]         #   global; verbose information; cumulative;
                        #   implies -progress
     -search            #   global; search access paths for source
                        #   files
                        #   specified on the command line; may specify
                        #   object code and libraries as well; this
                        #   option provides the IDE's 'access paths'
                        #   functionality
     -[no]wraplines     #   global; word wrap messages; default
     -maxerrors max     #   specify maximum number of errors to
                        #   print, zero
                        #   means no maximum; default is 0
     -maxwarnings max   #   specify maximum number of warnings to
```

```
                          #  print, zero means no maximum; default is 0
     -msgstyle keyword    #  global; set error/warning message style
     mpw                  #  use MPW message style
     std                  #  use standard message style; default
     gcc                  #  use GCC-like message style
     IDE                  #  use CW IDE-like message style
     parseable            #  use context-free machine-parseable message
                          #  style
                          #
     -[no]stderr          #  global; use separate stderr and
                          #  stdout streams;
                          #  if using -nostderr, stderr goes
                          #  to stdout Compiler
----------------------------------------------------------------
Preprocessing, Precompiling, and Input File Control Options
----------------------------------------------------------------
     -c                   #  global; compile only, do not link
     -[no]codegen         #  global; generate object code
     -[no]convertpaths    #  global; interpret #include filepaths
                          # specified for a foreign operating system;
                          #  i.e., <sys/stat.h> or <:sys:stat.h>; when
                          #  enabled,
                          #  '/' and ':' will separate directories and
                          #  cannot be used in filenames (note: this is
                          #  not a problem on Win32, since these
                          #  characters are already disallowed in
                          #  filenames; it is safe to leave the option
                          #  'on'); default
     -cwd keyword           #  specify #include searching semantics:
                          #  before
                          #  searching any access paths, the path
                          #  specified by this option will be searched
     proj                 #  begin search in current working directory;
                          #  default
     source               #  begin search in directory of source file
     explicit             #  no implicit directory; only search '-I' or
                          #  '-ir' paths
     include              #  begin search in directory of referencing
                          #  file
                          #
     -D+ | -d[efine       #   cased; define symbol 'name' to 'value' if
     name[=value]         #  specified, else '1'
     -[no]defaults        #  global; passed to linker;
                          #  same as '-[no]stdinc'; default
     -dis[assemble]       # global; passed to all tools;
                          #  disassemble files to stdout
     -E                   #  global; cased; preprocess source files
     -EP                  #  global; cased; preprocess and strip out
```

```
                        #  line
                        #  directives
    -enc[oding] keyword#  specify default source encoding; compiler
                        #  will automatically detect UTF-8 header or
                        #  UCS-2/UCS-4 encodings regardless of setting
     [no]ascii          #  ASCII; default
     [no]autodetect |   #  scan file for multibyte_encoding  (slower)
     [no]multibyte      #
     [no]mb             #
     [no]ascii          #  ASCII;
     [no]system         #  use system locale
     [no]UTF[8|-8]      #  UTF-8
     [no]SJIS |         #  shift-JIS
     [no]Shift-JIS |    #
     [no]ShiftJIS       #
     [no]EUC[JP|-JP]    #  EUC-JP
     [no]ISO[2022JP|    #  ISO-2022-JP
    -2022-JP]           #
    -ext extension        #  global; specify extension for generated
                        #  object
                        #  files; with a leading period ('.'), appends
                        #  extension; without, replaces source file's
                        # extension; for 'extension', maximum length
14
                        # chars; default is none

-gccinc[ludes]          # global;adopt GCC #include semantics:add '-
                        # I' paths to system list if '-I-' is not
                        #   specified, and search directory of
                        #  referencing file first for #includes (same
                        # as'-cwd include')
    -i- | -I-           #  global; change target for '-I'
                        #  access paths to
                        #  the system list; implies '-cwd explicit';
                        #  while compiling, user paths then system
                        #  paths
                        #  are searched when using
                        #  '#include "..."; only
                        #  system paths are searched with '#include
                        #   <...>'
    -I+ | -i p          #  global; cased; append access path to
current
                        #  include list(see '-gccincludes' and '-I-'
    -include file       #  prefix text file or precompiled header onto
                        #  all source files
    -ir path            #  global; append a recursive access path to
                        #  current #include list
    -[no]keepobj[ects]  #  global; keep object files generated after
```

```
                        #  invoking linker; if disabled, intermediate
                        #  object files are temporary and deleted
after
                        #  link stage; objects are always kept when
                        #  compiling
     -M                 # global; cased; scan source files for
                        # dependencies and emit Makefile, do not
                        # generate object code
     -MM                # global; cased; like -M, but do not list
                        #  system
                        # include files
     -MD                # global; cased; like -M, but write dependency
                        # map to a file and generate object code
     -MMD               # global;cased; like -MD, but do not list
                        # system include files
     -make              # global;scan source files for dependencies
and
                          # emit Makefile, do not generate object
code  -nofail           #continue working after errors in earlier
files
     -nolink            # global; compile only, do not link
     -noprecompile      # do not precompile any files based on the
                        # filename extension
     -nosyspath         # global; treat #include <...> like #include
                        #"..."; always search both user and system
                        # path lists
     -o file|dir          #   specify output filename or directory for
                        #  object
                        # file(s) or text output, or output filename
                        #  for linker if called
     -P                 # global; cased; preprocess and send output to
                        #  file; do not generate code
     -precompile file|di#generate precompiled header from source;
                        #  write
                        # header to 'file' if specified, or put header
                        #  in 'dir'; if argument is "", write header
to
                        #  source-specified location; if neither is
                        #  defined, header filename is derived from
                        #  source filename; note: the driver can tell
                        #  whether to precompile a file based on its
                        #  extension; '-precompile file source' then
is
                        #  the same as '-c -o file source'
     -preprocess        # global; preprocess source files
     -ppopt keyword[,...] # specify options affecting the preprocessed
                        #   output
      [no]break         #  emit file/line breaks; default
```

```
      [no]line           #  emit #line directives, else comments
      [no]full[path]     #  emit full path of file, else base filename
       [no] pragma         #  keep #pragma directives, else strip them;
                         # default
      [no comment]       # keep comments, else strip them
      [no] space         #  keep whitespace, else strip it
                         #
     -prefix file        #  prefix text file or precompiled header
                         #  onto all
                         #  source files
     -S                  #  global; cased; passed to all tools;
                         #  disassemble and send output to file
  -[no]stdinc             #  global; use standard system include paths
                          #  (specified by the environment variable
                          #  %MWCIncludes%); added after all system '-
I'
                          #   paths; default
  -U+ | -u[ndefine] name # cased; undefine symbol 'name'

----------------------------------------------------------------------
Front-End C/C++ Language Options
----------------------------------------------------------------------
  -ansi keyword            # specify ANSI conformance options,
                           #   overriding the given settings
     off                   # same as '-stdkeywords off', '-enum min',
                           #  and '-strict off'; default
     on|relaxed            #  same as '-stdkeywords on', '-enum min',
                           # and '-strict on'
     strict                #  same as '-stdkeywords on', '-enum int',
                           # and '-strict on'
                           #
  -ARM on|off              # check code for ARM (Annotated C++
Reference
                           #   Manual) conformance; default is off
  -bool on|off             # enable C++ 'bool' type, 'true' and 'false'
                           # constants; default is off
  -char keyword            # set sign of 'char'
     signed                #    chars are signed; default
     unsigned              #    chars are unsigned
                           #
  -Cpp_exceptions on|off   # passed to linker;
                           # enable or disable C++ exceptions; default
                           # is
                           # on
  -dialect | -lang keyword # passed to linker;
                           #  specify source language
     c                     #  treat source as C always
     c++                   #  treat source as C++ always
```

```
     ec++                    #  generate warnings for use of C++ features
                             #  outside Embedded C++ subset (implies
                             # 'dialect cplus')
                             # ‘dialect cplus’)
      c99                    # compile with c99 extensions
                             #
   -enum keyword             # specify word size for enumeration types
      min                    #   use minimum sized enums; default
      int                    #   use int-sized enums
                             #
   -for_scoping on|off       # control legacy (non-standard) for-scoping
                               #  behavior; when enabled, varaibles
                               #  declared in ‘for’ loops are visible
                               #   to the enclosing scope; when disabled,
                                  #  such variables are scoped to the
loop                                #  only; default is off
   -fl[ag] pragma              #  specify an ‘on/off’ compiler #pragma;
                               # ‘-flag foo’ is the same as ‘#pragma
                               #  foo on’
                               #     ‘-flag no-foo’ is the same as
‘#pragma
                                  #  foo off’; use ‘-pragma’ option
                               #  for other cases
   -inline keyword[,...]       #  specify inline options
      on|smart                    #  turn on inlining for 'inline'
                               #  functions;
                               #  default
      none|off                 #  turn off inlining
      auto                     #  auto-inline small functions (without
                               #  'inline' explicitly specified)
      noauto                   #  do not auto-inline; default
      all                      #  turn on aggressive inlining: same as
                               #  '-inline on, auto'
      deferred                 # defer inlining until end of compilation
                               #unit; this allows inlining of functions
in
                               #  both directions
      level=n                  # cased; inline functions up to 'n' levels
                               #deep; level 0 is the same as '-inline
on';
                               #  for 'n', range 0 - 8
      [no]bottomup             #  inline bootom-up, starting from
                               #  leaves of the call graph rather
                               #  than the top-level funcion; default
                               #
   -iso_templates on|off       #enable ISO C++ template parser (note:
this
                               #  requires a different MSL C++ library);
```

```
                                #  default is off
  -[no]mapcr                    # reverse mapping of '\n' and '\r' so that
                                #  '\n'==13 and '\r'==10 (for Macintosh
MPW
                                #  compatability)
  -msext keyword                #  [dis]allow Microsoft VC++ extensions
     on                         #  enable extensions: redefining macros,
                                #  allowing XXX::yyy syntax when declaring
                                #  method yyy of class XXX,
                                #  allowing extra commas,
                                #  ignoring casts to the same type,
                                #  treating function types with equivalent
                                   #  parameter lists but different return
types
                                # as equal,
                                   # allowing pointer-to-integer
                                #  conversions,
                                #  and various syntactical differences
    off                         #  disable extensions; default on non-x86
                                #  targets
                                #
  -[no]multibyte[aware]         # enable multi-byte character encodings
for
                                #   source text, comments, and strings
  -once                    # prevent header files from being processed
more
                                #  than once
  -pragma                       # define a pragma for the compiler such as
                                #  "#pragma ..."
  -r[equireprotos]              # require prototypes
  -relax_pointers               # relax pointer type-checking rules
  -RTTI on|off              # select run-time typing information (for
C++);
                                #   default is on
  -som                        # enable Apple's Direct-to-SOM
implementation
  -som_env_check                # enables automatic SOM environment and
new
                                #   allocation checking; implies -som
  -stdkeywords on|off        # allow only standard keywords; default is
off
  -str[ings] keyword[,...]      # specify string constant options
     [no]reuse               #    reuse strings; equivalent strings are
the
                                #      same object; default
     [no]pool                   #    pool strings into a single data
object
     [no]readonly               #    make all string constants read-only
```

```
                            #
  -strict on|off         # specify ANSI strictness checking; default
is
                            #   off
  -trigraphs on|off   # enable recognition of trigraphs; default is
off
  -wchar_t on|off      # enable wchar_t as a built-in C++ type;
default
                            #   is on

----------------------------------------------------------------------
Optimizer Options

   Note that all options besides '-opt
off|on|all|space|speed|level=...' are
   for backwards compatibility; other optimization options may be
superceded
   by use of '-opt level=xxx'.
----------------------------------------------------------------------
  -O                        # same as '-O2'
  -O+keyword[,...]       # cased; control optimization; you may
combine
                            #   options as in '-O4,p'
     0                      #    same as '-opt off'
     1                      #    same as '-opt level=1'
     2                      #    same as '-opt level=2'
     3                      #    same as '-opt level=3'
     4                      #    same as '-opt level=4'
     p                      #    same as '-opt speed'
     s                      #    same as '-opt space'
                            #
  -opt keyword[,...]        # specify optimization options
     off|none               #    suppress all optimizations; default
     on                     #    same as '-opt level=2'
     all|full               #    same as '-opt speed, level=4'
     [no]space              #    optimize for space
     [no]speed              #    optimize for speed
     l[evel]=num            #    set optimization level:
                            #      level 0: no optimizations
                            #
                            #      level 1: global register
allocation,
                            #      peephole, dead code elimination
                            #
                            #      level 2: adds common subexpression
                            #      elimination and copy propagation
                            #
                            #      level 3: adds loop transformations,
```

```
                        #      strength reduction, loop-invariant code
                            #      motion
                            #
                            #      level 4: adds repeated common
                            #      subexpression elimination and
                            #      loop-invariant code motion
                        #      ; for 'num', range 0 - 4; default is 0
     [no]cse                #    common subexpression elimination
       [no]commonsubs       #
     [no]deadcode           #    removal of dead code
     [no]deadstore          #    removal of dead assignments
     [no]lifetimes          #    computation of variable lifetimes
     [no]loop[invariants]   #    removal of loop invariants
     [no]prop[agation]  #  propagation of constant and copy
assignments
     [no]strength       #   strength reduction; reducing
multiplication
                            #      by an index variable into addition
     [no]dead               #    same as '-opt [no]deadcode' and '-opt
                            #      [no]deadstore'
     display|dump           #    display complete list of active
                            #      optimizations
                            #

----------------------------------------------------------------------
DSP M56800 CodeGen Options
----------------------------------------------------------------------
  [no]DO                    # for this tool;
                            #   specify hardware DO loops
  [no]segchardata           # for this tool;
                            #   segregate character data
  [no]asmout                # for this tool;
                            #   assembly file output
  [no]peep                  # for this tool;
                            #   active peepholer;
  [no]NDelay                # for this tool;
                            #   adjust for delayed load of N register;
  [no]sched                 # for this tool;
                            #   activate scheduler
  [no]REP                   # for this tool;
                            #   specify REP instruction
  [no]cmp32                 # for this tool;
                            #   emit 32-bit compare;
  [no]rodata                # for this tool;
                            #   write constant data to .rodata
section;

----------------------------------------------------------------------
```

```
Debugging Control Options
----------------------------------------------------------------------
  -g                   # global; cased; generate debugging
information;
                            #   same as '-sym full'
  -sym keyword[,...]        # global; specify debugging options
     off                    #    do not generate debugging
information;
                            #      default
     on                     #    turn on debugging information
     full[path]             #    store full paths to source files
                            #
----------------------------------------------------------------------
C/C++ Warning Options
----------------------------------------------------------------------
  -w[arn[ings]]             # global; for this tool;
    keyword[,...]           #   warning options
     off                      #   passed to all tools;
                              #      turn off all warnings
     on                       #   passed to all tools;
                              #     turn on most warnings
     [no]cmdline              #  passed to all tools;
                              #    command-line driver/parser warnings
     [no]err[or] |            #  passed to all tools;
       [no]iserr[or]          #    treat warnings as errors
     all                      #turn on all warnings, require
prototypes
     [no]pragmas |            #    illegal #pragmas
       [no]illpragmas         #
     [no]empty[decl]          #    empty declarations
     [no]possible |           #    possible unwanted effects
       [no]unwanted           #
     [no]unusedarg            #    unused arguments
     [no]unusedvar            #    unused variables
     [no]unused               #  same as -w
[no]unusedarg,[no]unusedvar
     [no]extracomma |         #    extra commas
       [no]comma              #
     [no]pedantic |           #    pedantic error checking
       [no]extended           #
     [no]hidevirtual |        #    hidden virtual functions
       [no]hidden[virtual]    #
     [no]implicit[conv]       #    implicit arithmetic conversions
                                 #      'warn impl_float2int,
                              #      impl_signedunsigned'
     [no]impl_int2float       #   implicit integral to floating
                              #      conversions
     [no]impl_float2int       #   implicit floating to integral
```

```
                              #      conversions
     [no]impl_signed unsigned #   implicit signed/unsigned conversions
                              #
     [no]notinlined           #  'inline' functions not inlined
     [no]largeargs            # passing large arguments to
unprototyped
                              # functions
     [no]structclass             #inconsistent use of 'class' and
                              #   'struct'
     [no]padding              # padding added between struct members
     [no]notused                 #result of non-void-returning
function                               #    not used
     [no]missingreturn        #  return without a value in a
                              #      non-void-returning function
     [no]unusedexpr              #use of expressions as statements
                              #without side effects
     [no]ptrintconv              # lossy conversions from pointers to
                            #integers, and
                            #      vice versa
     [no]anyptrintconv      #any conversions from pointers to integers
     [no]undef[macro]          #use of undefined macros in #if/#elif
                            #conditionals
     [no]filecaps              #incorrect capitalization used in
                            #include"..."
     [no]sysfilecaps        #incorrect capitalization used in
                            #include<...>
     [no]tokenpasting       #token not formed by ## operator
     display|dump           #    display list of active warnings
                               #
```

Linker

```
----------------------------------------------------------------------
Command-Line Linker Options
```

```
  -dis[assemble]       # global; disassemble object code and do not
                       #   link; implies '-nostdlib'
  -L+ | -l path        # global; cased; add library search path; default
                       #   is to search current working directory and
                       #   then system directories (see '-defaults');
                       #   search paths have global scope over the
                       #   command line and are searched in the order
                       #   given
  -lr path             # global; like '-l', but add recursive library
```

```
                         #   search path
  -l+file                # cased; add a library by searching access paths
                         #   for file named lib<file>.<ext> where <ext>
is
                         #   a typical library extension; added before
                         #   system libraries (see '-defaults')
  -[no]defaults          # global; same as -[no]stdlib; default
  -nofail                # continue importing or disassembling after
                         #   errors in earlier files
  -[no]stdlib            # global; use system library access paths
                         #   (specified by %MWLibraries%) and add system
                         #   libraries (specified by %MWLibraryFiles%);
                         #   default
  -S                     # global; cased; disassemble and send output to
                         #   file; do not link; implies '-nostdlib'

----------------------------------------------------------------------
ELF Linker Options
----------------------------------------------------------------------
  -[no]dead[strip]        # enable dead-stripping of unused code; default
  -force_active           # specify a list of symbols as undefined;
useful
    symbol[,...]          #   to force linking of static libraries
                          #
  -keep[local] on|off     # keep local symbols (such as relocations and
                          #   output segment names) generated during
link;
                          #   default is on
  -m[ain] symbol          # set main entry point for application or
shared
                          #   library; use '-main ""' to specify no entry
                          #  point; for 'symbol', maximum length 63
chars;
                          #   default is 'FSTART_'
  -map [keyword[,...]]    # generate link map file
     closure              #    calculate symbol closures
     unused               #    list unused symbols
                          #
  -sortbyaddr             # sort S-records by address; implies '-srec'
  -srec                   # generate an S-record file; ignored when
                          #   generating static libraries
  -sreceol keyword        # set end-of-line separator for S-record file;
                          #   implies '-srec'
     mac                  #    Macintosh ('\r')
     dos                  #    DOS ('\r\n'); default
     unix                 #    Unix ('\n')
                          #
  -sreclength length      # specify length of S-records (should be a
```

```
                       #   multiple of 4); implies '-srec'; for
                       #   'length', range 8 - 252; default is 64
  -usebyteaddr         # use byte address in S-record file; implies
                       #   '-srec'
  -o file              # specify output filename

----------------------------------------------------------------------
DSP M56800 Project Options
----------------------------------------------------------------------
  -application         # global; generate an application; default
  -library             # global; generate a static library

----------------------------------------------------------------------
DSP M56800 CodeGen Options
----------------------------------------------------------------------
  -ldata | -largedata  # data space not limited to 64K

----------------------------------------------------------------------
Linker C/C++ Support Options
----------------------------------------------------------------------
  -Cpp_exceptions on|off       # enable or disable C++ exceptions;
                            #    default is on
  -dialect | -lang keyword  # specify source language
     c                         #  treat source as C++ unless its
                            #  extension is
                            #      '.c', '.h', or '.pch'; default
     c++                    #    treat source as C++ always
                            #

----------------------------------------------------------------------
Debugging Control Options
----------------------------------------------------------------------
  -g                   # global; cased; generate debugging
information;
                            #   same as '-sym full'
  -sym keyword[,...]        # global; specify debugging options
     off                    #    do not generate debugging
information;
                            #      default
     on                     #    turn on debugging information
     full[path]             #    store full paths to source files
                            #

----------------------------------------------------------------------
```

```
Warning Options
-----------------------------------------------------------------------
  -w[arn[ings]]              # global; warning options
    keyword[,...]            #
     off                     #    turn off all warnings
     on                      #    turn on all warnings
     [no]cmdline             #    command-line parser warnings
     [no]err[or] |           #    treat warnings as errors
       [no]iserr[or]         #
     display|dump            #    display list of active warnings
                             #

-----------------------------------------------------------------------
ELF Disassembler Options
-----------------------------------------------------------------------
  -show keyword[,...]        # specify disassembly options
     only|none               #    as in '-show none' or, e.g.,
                             #      '-show only,code,data'
     all                     #    show everything; default
     [no]code | [no]text   #  show disassembly of code sections;
default
     [no]comments          #  show comment field in code; implies '-
show
                             #      code'; default
     [no]extended           #    show extended mnemonics; implies '-
show
                             #      code'; default
     [no]data              #    show data; with '-show verbose', show
hex
                             #      dumps of sections; default
     [no]debug | [no]sym     #    show symbolics information; default
     [no]exceptions      #  show exception tables; implies '-show
data';
                             #      default
     [no]headers             #    show ELF headers; default
     [no]hex                 #    show addresses and opcodes in code
                         #    disassembly; implies '-show code';
default
     [no]names               #    show symbol table; default
     [no]relocs              #    show resolved relocations in code and
                             #      relocation tables; default
     [no]source          #    show source in disassembly; implies '-
show
                         #    code'; with '-show verbose', displays
                         #      entire source file in output, else
shows
                         #      only four lines around each function;
                             #      default
```

```
     [no]xtables             #    show exception tables; default
     [no]verbose         #  show verbose information, including hex
dump
                             #      of program segments in
applications;
                             #      default
                             #
```

Assembler

```
----------------------------------------------------------------------
Assembler Control Options
----------------------------------------------------------------------
  -[no]case                  # identifiers are case-sensitive; default
  -[no]debug                 # generate debug information
  -[no]macro_expand          # expand macro in listin output
  -[no]assert_nop      # add nop to resolve pipeline dependency;
default
  -[no]warn_nop              # emit warning when there is a pipeline
                             #   dependency
  -[no]warn_stall         # emit warning when there is a hardware stall
  -[no]legacy                # allow legacy DSP56800 instructions(imply
                             #   data/prog 16)
  -[no]debug_workaround       # Pad nop workaround debuggin issue in
some
                             #   implementation; default
  -data keyword              # data memory compatibility
     16                      #    16 bit; default
     24                      #    24 bit
                             #
  -prog keyword              # program memory compatibility
     16                      #    16 bit; default
     19                      #    19 bit
     21                      #    21 bit
                             #
----------------------------------------------------------------------
```

14

Libraries and Runtime Code

You can use a variety of libraries with the CodeWarrior™ IDE. The libraries include ANSI-standard libraries for C, runtime libraries, and other code. This chapter explains how to use these libraries for DSP56800 development.

With respect to the Metrowerks Standard Library (MSL) for C, this chapter is an extension of the *MSL C Reference*. Consult that manual for general details on the standard libraries and their functions.

This chapter contains the following sections:

MSL for DSP56800

This section explains the Metrowerks Standard Library (MSL) modified for use with DSP56800. CodeWarrior IDE for DSP56800 includes the source and project files for MSL so that you can modify the library if necessary.

Using MSL for DSP56800

CodeWarrior IDE for DSP56800 includes a version of the Metrowerks Standard Library (MSL). The MSL is a C library you can use in your embedded projects. All of the sources necessary to build MSL are included in CodeWarrior IDE for DSP56800, along with the project file and targets for different MSL configurations. If you already have a version of CodeWarrior IDE installed on your computer, the CodeWarrior installer adds the new files needed for building versions of MSL for DSP56800.

Do not modify any of the source files that support MSL.

Console and File I/O

DSP56800 Support provides standard C calls for I/O functionality with full ANSI/ISO standard I/O support with host machine console and file I/O for debugging sessions (Host I/O) through the JTAG port in addition to such standard C calls such as memory functions malloc() and free().

A minimal "thin" printf via "console_write" and "fflush_console" is provided in addition to standard I/O.

See the *MSL C Reference* manual (Metrowerks Standard Library).

MSL Configurations for DSP56800

There are two DSP56800 MSL libraries available. Both support standard C calls with optional I/O functionality. One library has a minimal `printf` function providing console output using debugger. The other library has full ANSI/ISO standard I/O support, including host machine console and file I/O for debugging sessions. The memory functions `malloc()` and `free()` are also supported for both libraries.

The two provided DPS56800 MSL libraries are:

MSL C 56800.lib

This library provides standard C library support without standard I/O. A minimal "thin" `printf` is provided but other `stdio` is stripped out in order to maximize performance. The `printf` sends characters to the CodeWarrior console window via the debugger. Use this library when you need minimal `printf` support for debugging and saving space.

MSL C 56800 host I/O.lib

This library adds ANSI/ISO standard I/O support through the debugger. The standard C library I/O is supported, including `stdio.h`, `sdderr.h`, and `stdin.h`. Use this library when you want to perform `stdio` calls, including CodeWarrior console `stdout/stdin`, and host machine file I/O, for debugging.

Host File Location

Files are created with `fopen` on the host machine as shown in Table 14.1 on page 308.

Table 14.1 Host File Creation Location

fopen filename parameter	host creation location
filename with no path	target project file folder
full path	location of full path

Binary and Text Files

`stdio` call `fopen` can open files as text or binary, depending on the open mode. For DSP56800 host I/O file operations, subsequent `stdio` calls treat the file as text or binary depending on how the file was originally opened with `fopen`.

NOTE You must decide whether to open the file as text or binary.

Binary and text files are handled differently because DSP56800 char (character) is 16-bits and x86 host char is 8-bits.

- Text file I/O operations are 1-to-2 mapping.
- Binary file I/O operations are 1-to-1 mapping.

Files are created with `fopen` on the host machine as shown in Table 14.2 on page 309.

Table 14.2 Host File Creation Location

file opened as	host elements	target elements
text	8-bit	16-bit
binary	16-bit	16-bit

Text File I/O

DSP56800 host I/O does 16-bit to 8-bit mapping for host text files. The host text file is handled as 8-bit elements with conversion to 16-bit elements on the target side.

For example, if you open the host file with the `fopen` mode "w", the file opens as new text file or a truncated existing text file of the file name. When `fwrite` is called, the host file writes the DSP56800 buffer of 16-elements of the host file as 8-bit elements.

Binary File I/O

DSP56800 host I/O does 16-bit to 16-bit mapping for binary files. The host binary file is handled as 16-bit elements.

Allocating Stacks and Heaps for the DSP56800

Stationery linker command files (LCF) define heap, stack, and BSS locations. LCFs are specific to each target board. When you use M56800 stationery to create a new project, CodeWarrior automatically adds the LCF to the new project.

See “ELF Linker” on page 263 for general LCF information. See each specific target LCF in Stationery for specific LCF information.

Definitions

Stack

The stack is a last-in-first-out (LIFO) data structure. Items are pushed on the stack and popped off the stack. The most recently added item is on top of the stack. Previously added items are under the top, the oldest item at the bottom. The "top" of the stack may be in low memory or high memory, depending on stack design and use. M56800 uses a 16-bit-wide stack.

Heap

Heap is an area of memory reserved for temporary dynamic memory allocation and access. MSL uses this space to provide heap operations such as `malloc`. M56800 does not have an operating system (OS), but MSL effectively synthesizes some OS services such as heap operations.

BSS

BSS is memory space reserved for uninitialized data. The compiler will put all uninitialized data here. The stationery `init` code zeroes this area at startup. See the 56824_`init.c` (startup) code example code in this chapter for general information and the stationery `init` code files for specific target implementation details.

NOTE Instead of accessing the original Stationery files themselves (in the Stationery folder), create a new project using Stationery (see “Creating a Project”) which will make copies of the specific target board files such as the LCF.

Variables defined by Stationery Linker Command Files

Each Stationery LCF defines variables which are used by runtime code and MSL. You can see how the values for these variables are calculated by examining any of the Stationery LCFs.

See Table 14.3 on page 310 for the variables defined in each Stationery LCF.

Table 14.3 LCF Variables and Address

Variables	Address
_stack_addr	The start address of the stack
_heap_size	The size of the heap

Table 14.3 LCF Variables and Address (*continued*)

Variables	Address
_heap_addr	The start address of the heap
_heap_end	The end address of the heap
_bss_start	Start address of memory reserved for uninitialized variables
_bss_end	End address of BSS

Additional Information and Specific Target Implementation Details

See each Stationery specific target board LCF for additional comments and implementation details. Perform a search for the variable name for quick access.

Depending on the target, implementation will be different between LCFs. For example, for targets using Host I/O, considerably more heap size is allocated in the LCF.

Runtime Initialization

The default `init` function is the bootstrap or glue code that sets up the DSP56800 environment before your code executes. This function is in the `init` file for each board-specific stationery project. The routines defined in the `init` file performs other tasks such as clearing the hardware stack, creating an interrupt table, and retrieving the stack start and exception handler addresses.

The default code in the `init` function also sets the addressing mode in the modifier register (`M01`) to 0xFFFF.

The final task performed by the `init` function is to call the `main()` function.

The starting point for a program is set in the **Entry Point** field in the **M56800 Linker on page 88** **Settings** panel.

When creating a project from R5.1 stationery, the `init` code is specific to the DSP56800 board. See the startup folder in the new project folder for the `init` code.

Listing 14.1 Sample Initialization File (DSP56803EVM)

```
/*
  56803_init.c

  Metrowerks, a Freescale Company
```

```
  sample code

  */



#include "DSP56F803_init.h"


extern _rom_to_ram;
extern _data_size;
extern _data_RAM_addr;
extern _data_ROM_addr;
extern _bss_size;
extern _bss_addr;



asm void init_M56803_()
{
bfset   #_32bit_compares,omr     //

//
  move   #-1,x0
  move   x0,m01                  // set the m reg to linear addressing

  move    hws,la                    // clear the hardware stack
  move    hws,la


// init registers

  move    #0,r1
  move    r1,x:IPR
  move    r1,x:COPCTL



// initialize compiler environment

CALLMAIN:

// setup stack
  move    #_stack_addr,r0      // get stack start address
  nop
```

```
  move    r0,x:<mr15           // set frame pointer to main stack top
  move    r0,sp                // set stack pointer too
  move    #0,r1
  move    r1,x:(r0)



// setup the PLL (phase locked loop)

  move    #pllcr_init,x:PLLCR    // set lock detector on and choose
core
                                //clock
  move    #plldb_init,x:PLLDB      // set to max freq
  move    #wait_lock,x0         // set x0 with timeout value
                                // timeout handles simulator case
pll_test_lock:                  // loop until PLL is locked
                                // or we reach timeout limit
  decw    x0                    // decrement our timeout value
  tstw     x0                   // test for zero
  beq      pll_timeout          // if timed-out, proceed anyway
  brclr
#pllsr_init,x:PLLSR,pll_test_lock
  pll_timeout:
// pll locked
  move  #pllcr_proceed,x:PLLCR       // set lock detector on, choose
                                     // PLL clock
  move    x:PLLSR,x0                 // clear pending clkgen
interrupts
  move     x0,x:PLLSR



// setup exception handler and interrupt levels
  move    M56803_int_Addr,r1      // address
  push    r1                      // establish exception handler
  bfset   #$0100,sr               // enable all levels of interrupts
  bfclr    #$0200,sr              // allow IPL 0 interrupts



// xrom-to-xram option

  move    -#_rom_to_ram,r0          // check for option
  tstw    r0
  beq     end_rom2ram
  move    #_data_size,r2            // set data size
  move    #_data_ROM_addr,r3        // src address -- xROM data start
```

```
  move     #_data_RAM_addr,r1              // dest address -- xRAM data
                                       // start
  do       r2,end_rom2ram              // copy for r2 times
  move     x:(r3)+,x0                  // fetch value at address r3
  move     x0,x:(r1)+                  // stash value at address r1
end_rom2ram:



// clear bss always

  move     #0,x0                          // set x0 to zero
  move     #_bss_size,r2                  // set bss size
  move     #_bss_addr,r1                  // dest address -- bss data start
  do       r2,end_bss_clear               // do for r2 times
  move     x0,x:(r1)+                     // stash zero at address
  nop
end_bss_clear:



// call main()
  move     #M56803_argc,y0                  // pass parameters to main()
  move     #M56803_argv,r2
  move      #M56803_arge,r3
  jsr       main                              // call the users program
  jsr     fflush
  debug
  rts
}
```

The startup folder includes the following:

- Stack setup
- PLL setup
- Exception handler and interrupt setup
- BSS zeroing
- Static initialization
- Jump to main

NOTE The original general-purpose runtime `init` code (FSTART) remains in the M56800 support library to provide compatibility for older projects. The MSL runtime project is: `CodeWarrior\56800`

`Support\msl\MSL_C\DSP_56800\Project\MSL C 56800.mcp`	See project group runtime: init, file FSTART.c.

15

Troubleshooting

This chapter explains common problems encountered when using the CodeWarrior™ IDE for DSP56800, and their possible solutions.

Troubleshooting Tips

This chapter contains the following sections:

If you are having trouble with CodeWarrior Development Studio for Freescale 56800 and this section does not help you, e-mail technical support at: `support@Freescale.com`

The Debugger Crashes or Freezes When Stepping Through a REP Statement

Due to the nature of DSP56800 instruction pipeline, do not set a breakpoint on a `REP` statement in the debugger. Doing so may cause the `REP` instruction to enter an infinite loop and freeze or crash the IDE.

"Can't Locate Program Entry On Start" or "Fstart.c Undefined"

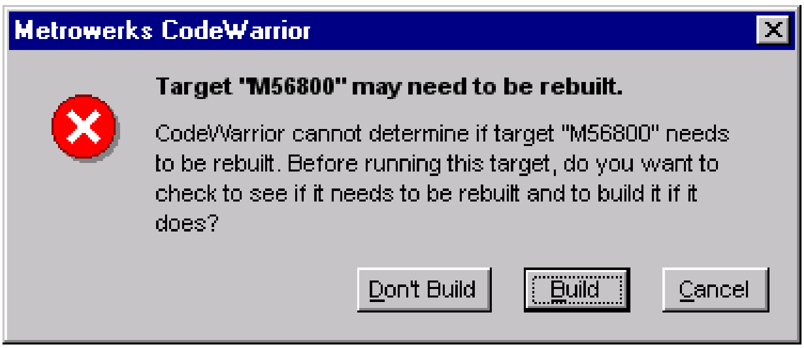
By default, the CodeWarrior stationery defines the entry point of program execution as `FSTART_`. The entry point is edited in the project target settings by selecting **Edit > M56800 Settings** from the menu bar of the Freescale CodeWarrior window and then M56800 Linker from the **Target Settings** panel. If the entry point is changed and not updated in the sources, linker errors are generated for undefined sources.

The FSTART.c program is defined in the MSL and may also generate errors if the CodeWarrior IDE cannot find the MSL path due to access path errors within a DSP56800 project.

When Opening a Recent Project, the CodeWarrior IDE Asks If My Target Needs To Be Rebuilt

If you open a recent project file and then select **Project > Debug** from the menu bar of the Freescale CodeWarrior window, the dialog box shown in Figure 15.1 on page 318 appears:

Figure 15.1 Rebuild Alert



This dialog box informs you that the software determines if your object code needs to be rebuilt. If you have made no changes since the last build, the CodeWarrior IDE does not change your object file when you select the **Build** option.

"Timing values not found in FLASH configuration file. Please upgrade your configuration file. On-chip timing values will be used which may result in programming errors"

This indicates you have an old flash configuration file that does not include timing information. If you continue to use this file, it could result in programming errors and a shorter life for the flash memory.

To upgrade your flash configuration file, replace the existing flash configuration file with the flash configuration file from the M56800 Support.

The flash configuration file is located in the following directory:

CodeWarrior\M56800 Support\initialization

IDE Closes Immediately After Opening

There may be a conflict with another version of the CodeWarrior IDE on your system. Running the `regservers.bat` file in the `Freescale/Bin` directory usually resolves this problem when there are different versions of the CodeWarrior IDE installed on the same computer.

Errors When Assigning Physical Addresses With The `Org` Directive

You cannot use the `ORG` directive with the CodeWarrior IDE DSP56800 assembler to specify physical addresses for program (P:) and data (X:) memory.

The Debugger Reports a Plug-in Error

When the CodeWarrior IDE debugger reports a plug-in error, a dialog box appears that reads “Embedded DSP Plug-in Error. Can’t connect to board.” If you see this dialog box, check the following:

- Verify that the hardware cards are installed and seated properly.
- Verify that all of the cables are connected properly.

- Verify that power is being supplied to the DSP hardware.

Windows Reports a Failed Service Startup

When the Windows Service Control Manager reports a failed service startup, the message box shown in Figure 15.2 on page 320 appears:

Figure 15.2 Service Control Manager Message Box

If you see the above message box, check the following:

- Ensure that you have not selected a conflicting address for use with the DSP hardware. The Resources Manager can help you determine whether or not there is a conflict.
- Check input/output addresses according to the operating system you are using:

  Windows 98

1. To access the Resources Manager, open the Control Panel and click the **Device Manager** tab.
2. Click **Properties** to display the **Computer Properties** window.
3. Click the **View Resources** tab in the **Computer Properties** window.
4. Click the **Input/Output** radio button to view all active input/output addresses.

  Windows NT

1. To access the Resources Manager, select **Start > Programs > Administrative Tools > Windows NT Diagnostics**.
2. Click the **Resources** tab in the Windows NT Diagnostics window.
3. Click **I/O Port** at the bottom of the tab to view all currently active input/output addresses.

No Communication With The Target Board

If you are unable to establish communication with the target DSP hardware, check the following:

- Verify that the hardware boards are properly connected to the computer. Follow the installation instructions in “Getting Started” on page 19.

- If you are using the Freescale ADS hardware with the ISA bus interface, ensure that you select the correct I/O address for the ISA card. If you have another device attempting to use this address, you must reconfigure that device to use another address or disable that device.
- Verify that all the hardware boards have power:
  - A green LED lights up on both the ADS and EVM boards.
  - A red LED and a yellow LED illuminate on the Domain Technologies SB-56K Emulator.
- Verify that all target settings are correct.

Downloading Code to DSP Hardware Fails

If you are unable to download code to the target DSP hardware, verify that the communications to the target hardware are working correctly.

The CodeWarrior IDE Crashes When Running My Code

Use one of the samples provided with CodeWarrior IDE for DSP56800 to verify that your system is working correctly.

The Debugger Acts Strangely

Sometimes DSP hardware can become corrupted and unusable, even after a soft reset. If the debugger has problems executing code, you might have to perform a hard reset of the DSP hardware.

To reset the EVM board, follow these steps:

1. Disconnect the power cable from the board.
2. Wait at least 5 seconds.
3. Reconnect the power supply to the EVM board. This reconnection step resets the board and clears its RAM.

To reset the ADS board, follow these steps:

1. Disconnect the power cable from the ADS board.
2. Wait at least 5 seconds.
3. Reconnect the power supply to the ADS board. This reconnection step resets the board and clears its RAM.

Problems With Notebook Computers

If you experience any problems downloading using the parallel port interface while using a notebook computer, ensure that the parallel port is set in bidirectional mode.

On Dell Latitudes, the ECP setting in CMOS has not emitted enough voltage through the parallel port. Increasing the ECP value may solve this problem.

How to make Parallel Port Command Converter work on Windows® 2000 Machines

If you encounter problems connecting to your Windows® 2000 machine using the parallel port command converter, check the following settings:

1. Verify LPT Port number matches the parallel port:
   a. Launch CCS.
   b. Select **File > Configure**.
   c. Ensure that the LPT port is set to parallel port and correct LPT number.
   d. Click **Save**.
2. Verify “Enable legacy Plug and Play” is enabled for the parallel port:
   a. Access the **Device Manager**.
   b. Access the LPT port settings window.
   c. Click the **Properties** button.
   d. In the **Properties** window, click the **Enable Legacy Plug and Play** box.
3. Verify the parallel port is set for “fast bi-directional transfer”:
   a. Access the BIOS settings.
   b. Set the parallel port for fast bi-directional transfers (EEP or ECP) instead of just bi-directional.

A

Porting Issues

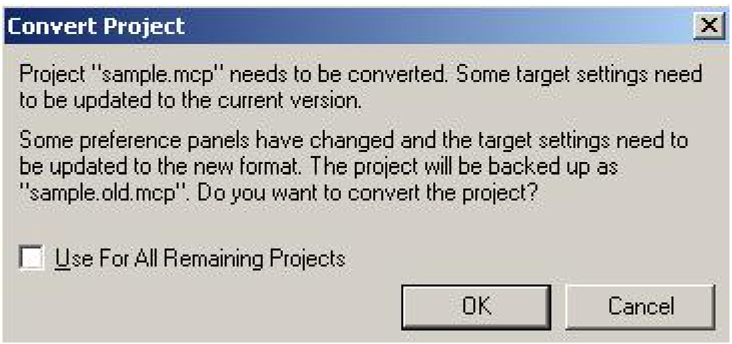
This appendix explains issues relating to successfully porting code to the most current version of the CodeWarrior Development Studio for Freescale 56800/E Digital Signal Controllers. This appendix lists issues related to successfully porting sources from the Suite56™ toolset and differences that occur between the CodeWarrior IDE and the Suite56 tools.

This appendix contains the following sections:

Converting the DSP56800 Projects from Previous Versions

When you open older projects in the CodeWarrior IDE, the IDE automatically prompts you to convert your existing project (Figure A.1 on page 323). Your old project will be backed up if you need to access that project file at a later time. The CodeWarrior IDE cannot open older projects if you do not convert them.

Figure A.1 Project Conversion Dialog



Removing “illegal object_c on pragma directive” Warning

If after porting a project to DSP56800 7.x, you get a warning that says `illegal object_c on pragma directive`, you need to remove it. To remove this warning:

1. Open the project preference and go to the C/C++ Preprocessor.
2. Remove the line `#pragma objective_con` from the prefix text field.

Setting-up Debugging Connections

In the DSP56800 7.x, debugging connections to the hardware or simulator are made using the Remote Debugging panel.

Older versions of the DSP56800 connected using other settings.

If you open a project created using a previous version of the CodeWarrior IDE, you must now set up the debugging connections using the new settings.

For more information on the Remote Debugging panel, see “Remote Debugging” on page 92.

Using XDEF and XREF Directives

The `XDEF` and `XREF` directives are not used with the CodeWarrior assembler. Use the `GLOBAL` directive to make symbols visible outside of a section.

Using the ORG Directive

Memory space and location counters cannot be updated with the ORG directive. You must use the linker command file to specify exact memory addresses rather than in the assembler. For example, if you declare:

```
ORG P:$0020
SECTION myISR_20
rti
ENDSEC
SECTION myISR_30
jsr foot
rti
ENDSEC
```

You would need to change your `ORG` directive to:

`ORG P:`

and your linker command file would be changed as follows:

```
MEMORY {
     .text (RWX)  : ORIGIN = 0x1000, LENGTH = 0x0
     .data (RW)   : ORIGIN = 0x2000, LENGTH = 0x0
     .text2(RWX)  : ORIGIN = 0x20,    LENGTH = 0x0
}

SECTIONS {
    .location_specific_code :
    {
      . = 0x20;
      *(myISR_20.text)
      . = 0x30;
      *(myISR_30.text)
    } > .text2

    .main_application :
    {
      *(.text)
      *(.rtlib.text)
      *(fp_engine.text)
      *(user.text)
    } > .text

    .main_application_data :
    {
      *(.data)
      *(fp_state.data)
      *(rtlib.data)
      *(rtlib.bss.lo)
      *(.bss)
    } > .data
```

B

DSP56800x New Project Wizard

This appendix explains the high-level design of the new project wizard.

Overview

The DSP56800x New Project Wizard supports the DSP56800x processors listed in Table B.1 on page 327.

Table B.1 Supported DSP56800x Processors for the New Project Wizard

DSP56800	DSP56800E
DSP56F801 (60 MHz)	DSP56852
DSP56F801 (80 MHz)	DSP56853
DSP56F802	DSP56854
DSP56F803	DSP56855
DSP56F805	DSP56857
DSP56F807	DSP56858
DSP56F826	MC56F8013
DSP56F827	MC56F8014
	MC56F8023
	MC56F8025
	MC56F8036
	MC56F8037
	MC56F8122
	MC56F8123

Table B.1 Supported DSP56800x Processors for the New Project Wizard (*continued*)

DSP56800	DSP56800E
	MC56F8145
	MC56F8146
	MC56F8147
	MC56F8155
	MC56F8156
	MC56F8157
	MC56F8165
	MC56F8166
	MC56F8167
	MC56F8322
	MC56F8323
	MC56F8335
	MC56F8345
	MC56F8346
	MC56F8356
	MC568357
	MC568365
	MC568366
	MC568367

Wizard rules for the DSP56800x New Project Wizard are described in these sub-sections:

- Page Rules on page 329
- Resulting Target Rules on page 330
- on page 331Rule Notes on page 331

Click on the following link for details about the DSP56800x New Project Wizard:

- DSP56800x New Project Wizard Graphical User Interface on page 332

Page Rules

The page rules governing the wizard page flow for the simulator and the different processors are shown in the Table B.2 on page 329, Table B.3 on page 329, Table B.4 on page 330, and Table B.5 on page 330.

Table B.2 Page Rules for the Simulator, DSP56F801 (60 and 80 MHz), DSP56F802, MC56F801x, MC56F802x, MC56F803x, MC56F812x, and MC56F832x

Target Selection Page	Next Page	Next Page
any simulator	Program Choice Page	Finish Page
DSP56F801 60 MHz		
DSP56F801 80 MHz		
DSP56F802		
MC56F801x		
MC56F802x		
MC56F803x		
MC56F812x		
MC56F832x		

Table B.3 Page Rules for the DSP56F803, DSP56F805, DSP56F807, DSP56F826, and DSP56F827

Target Selection Page	Next Page	Next Page	Next Page
DSP56F803	Program Choice Page	External/Internal Memory Page	Finish Page
DSP56F805			
DSP56807			
DSP56F826			
DSP56F827			

Table B.4 Page Rules for the DSP56852, DSP56853, DSP56854, DSP56855, DSP56857, and DSP56858

Target Selection Page	Next Page	Next Page
DSP56852	Program Choice Page	Finish Page
DSP56853		
DSP56854		
DSP56855		
DSP56857		
DSP56858		

Table B.5 Page Rules for the MC56F814x, MC56F815x, MC56F816x, MC56F833x, MC56F834x, MC56F835x, and MC56F836x

Target Selection Page	Next Page	Next Page	Next Page if Processor Expert Not Selected	Next Page
MC56F814x	Program Choice Page	Data Memory Model Page	External/Internal Memory Page	Finish Page
MC56F815x				
MC56F816x				
MC56F833x				
MC56F834x				
MC56F835x				
MC56F836x				

Resulting Target Rules

The rules governing possible final project configurations are shown in Table B.6 on page 331.

Table B.6 Resulting Target Rules

Target	Possible Targets
56800 Simulator	Target with Non-HostIO Library and Target with Host IO Library
56800E Simulator	Small Data Model and Large Data Model
DSP5680x	External Memory and/or Internal Memory with pROM-to-xRAM Copy
DSP5682x	External Memory and/or Internal Memory with pROM-to-xRAM Copy
DSP5685x	(Small Data Model and Small Data Model with HSST) or (Large Data Model and Large Data Model with HSST)
MC56F801x MC56F802x MC56F803x	Small Data Model Internal Memory with pROM-to-xRAM Copy
MC56F812x MC56F832x	Small Data Model or Large Data Model Internal Memory with pROM-to xRAM Copy
MC56F814x MC56F815x MC56F816x MC56F833x MC56F834x MC56F835x MC56F836x	(Small Data Memory External and/or Small Data Memory Internal with pROM-to-xRAM Copy) or (Large Data Memory External and/or Large Data Memory Internal with pROM-to-xRAM Copy)

Rule Notes

Additional notes for the DSP56800x New Project Wizard rules are:

- The DSP56800x New Project Wizard uses the DSP56800x EABI Stationery for all projects. Anything that is in the DSP56800x EABI Stationery will be in the wizard-created projects depending on the wizard choices.
- The DSP56800x EABI Stationery has all possible targets, streamlined and tuned with the DSP56800x New Project Wizard in mind.

- The DSP56800x New Project Wizard creates the entire simulator project with all the available targets in context of “Stationery as documentation and example.”

DSP56800x New Project Wizard Graphical User Interface

This section describe the DSP56800x New Project Wizard graphical user interface.

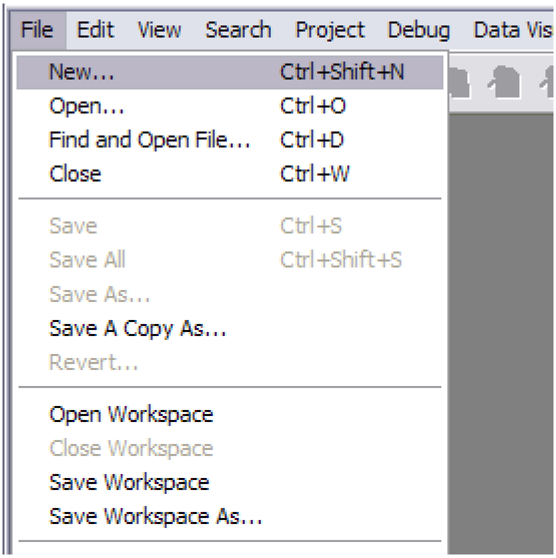
The subsections in this section are:

- Invoking the New Project Wizard on page 332
- New Project Dialog Box on page 333
- Target Pages on page 334
- Program Choice Page on page 343
- Data Memory Model Page on page 344
- External/Internal Memory Page on page 345
- Finish Page on page 346

Invoking the New Project Wizard

To invoke the New Project dialog box, from the Freescale CodeWarrior menu bar, select **File>New** (Figure B.1 on page 333).

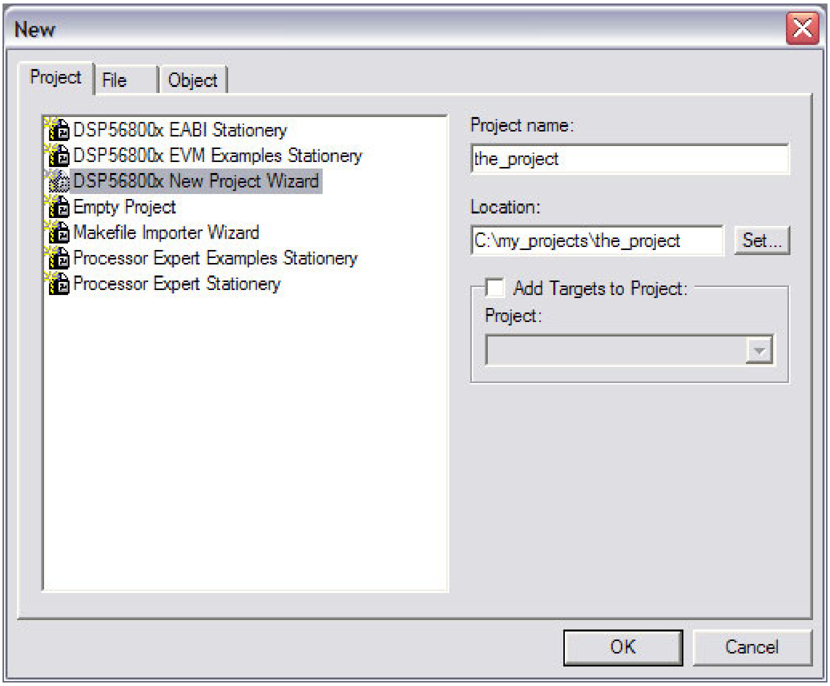
Figure B.1 Invoking the DSP56800x New Project Wizard



New Project Dialog Box

After selecting **File>New** from the Freescale CodeWarrior menu bar, the New project Dialog Box (Figure B.2 on page 334) appears. In the list of stationeries, you can select either the “DSP56800x New Project Wizard” or any of the other regular stationery.

Figure B.2 New Project Dialog Box



Target Pages

When invoked, the New Project Wizard first shows a dynamically created list of supported target families or simulators and processors. Each DSP56800x family is associated with a subset of supported processors (Figure B.3 on page 335, Figure B.4 on page 336, Figure B.5 on page 337, Figure B.6 on page 338, Figure B.7 on page 339, Figure B.8 on page 340, Figure B.9 on page 341, Figure B.10 on page 342, and Figure B.11 on page 343).

Figure B.3 DSP56800x New Project Wizard Target Dialog Box (DSP56F80x)

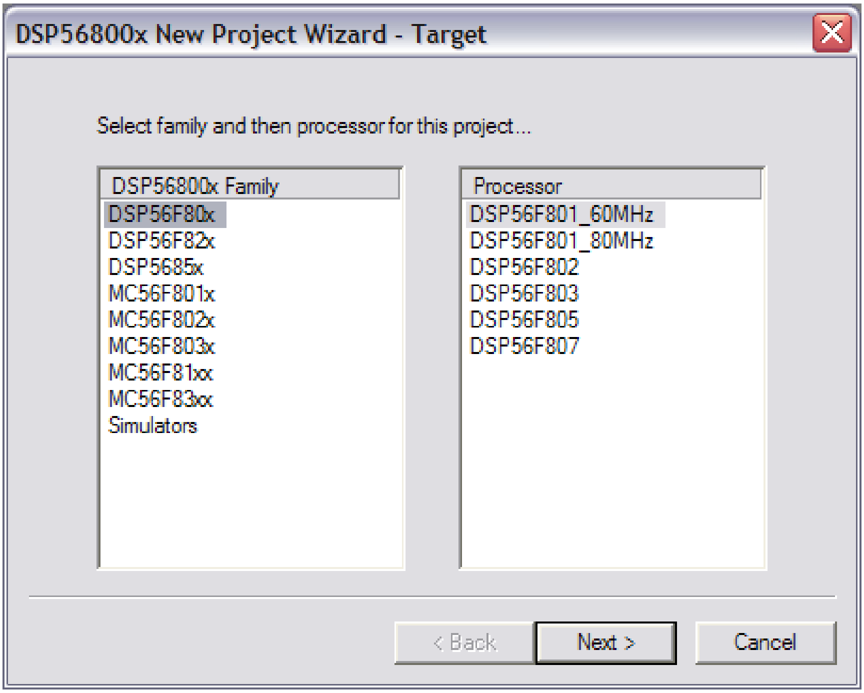


Figure B.4 DSP56800x New Project Wizard Target Dialog Box (DSP56F82x)

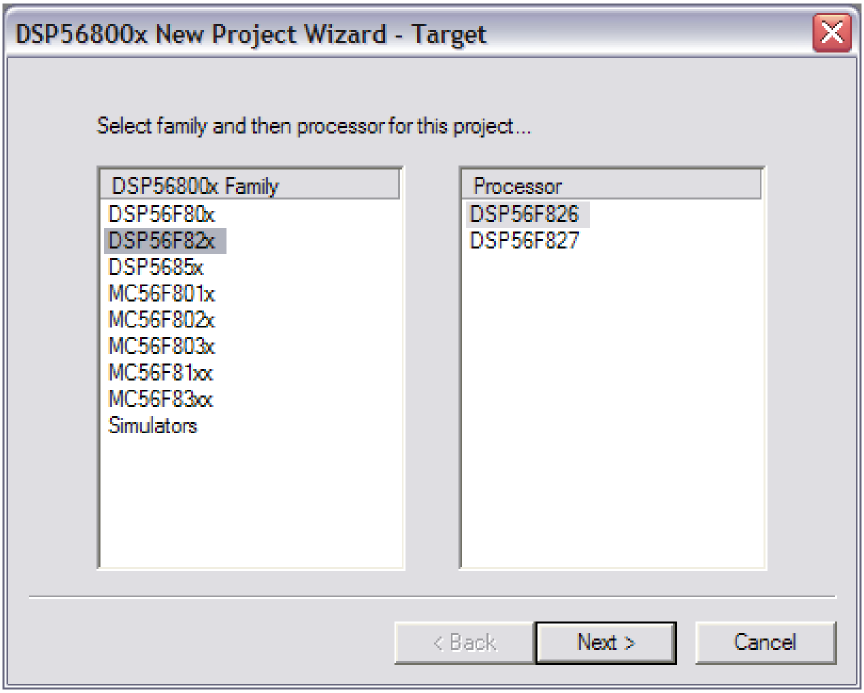


Figure B.5 DSP56800x New Project Wizard Target Dialog Box (DSP5685x)

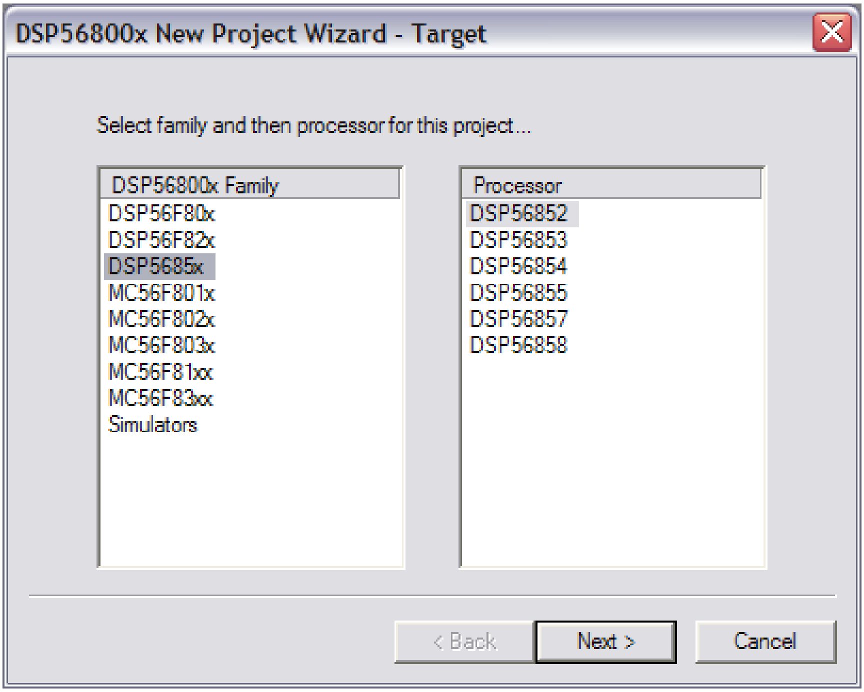


Figure B.6 DSP56800x New Project Wizard Target Dialog Box (MC56F801x)

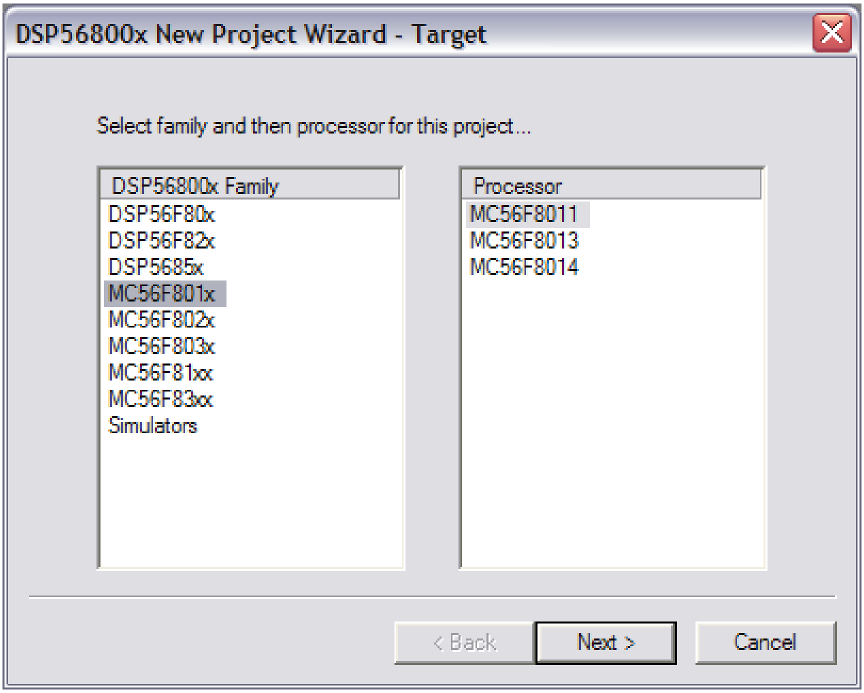


Figure B.7 DSP56800x New Project Wizard Target Dialog Box (MC56F802x)

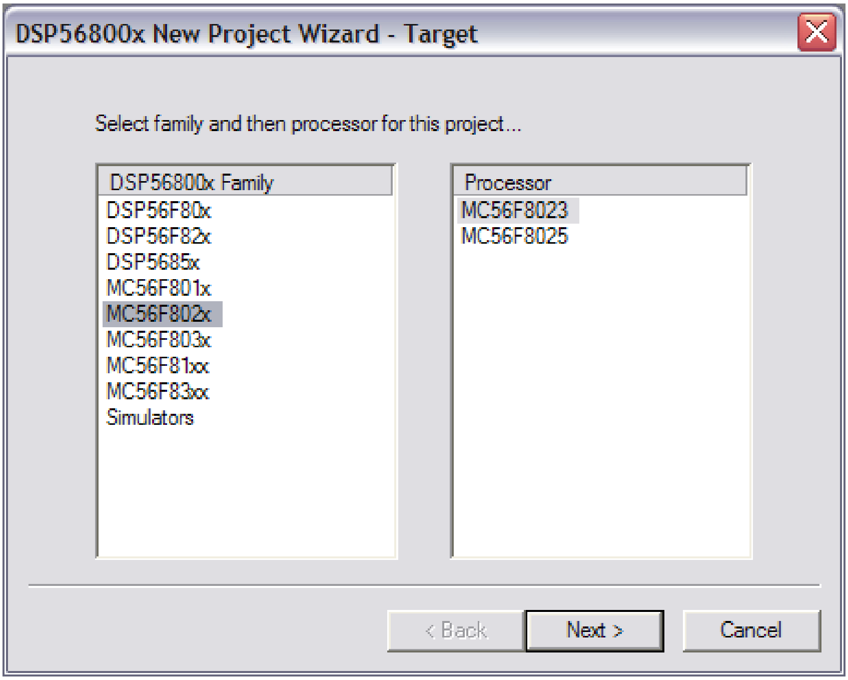


Figure B.8 DSP56800x New Project Wizard Target Dialog Box (MC56F803x)

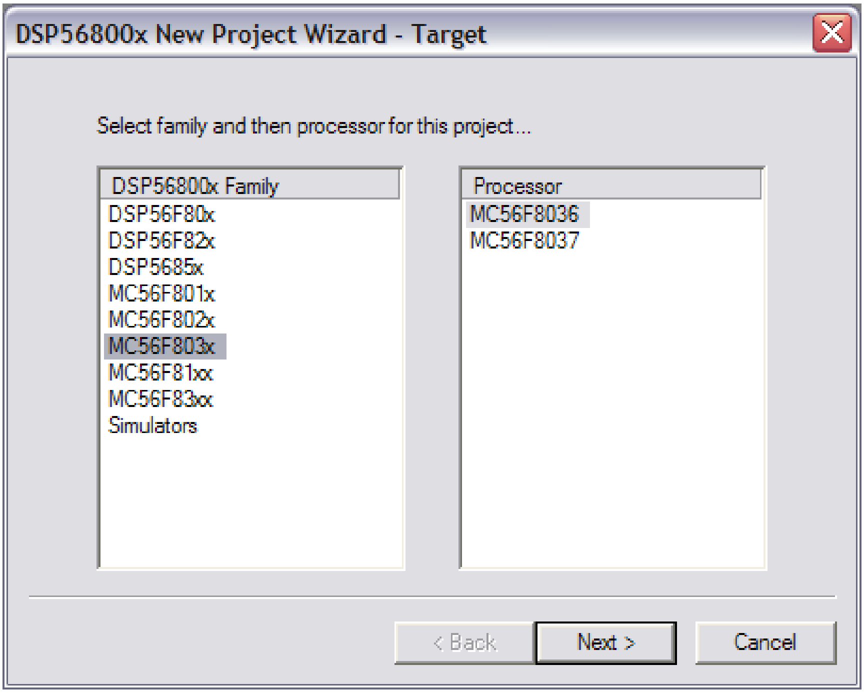


Figure B.9 DSP56800x New Project Wizard Target Dialog Box (MC56F81xx)

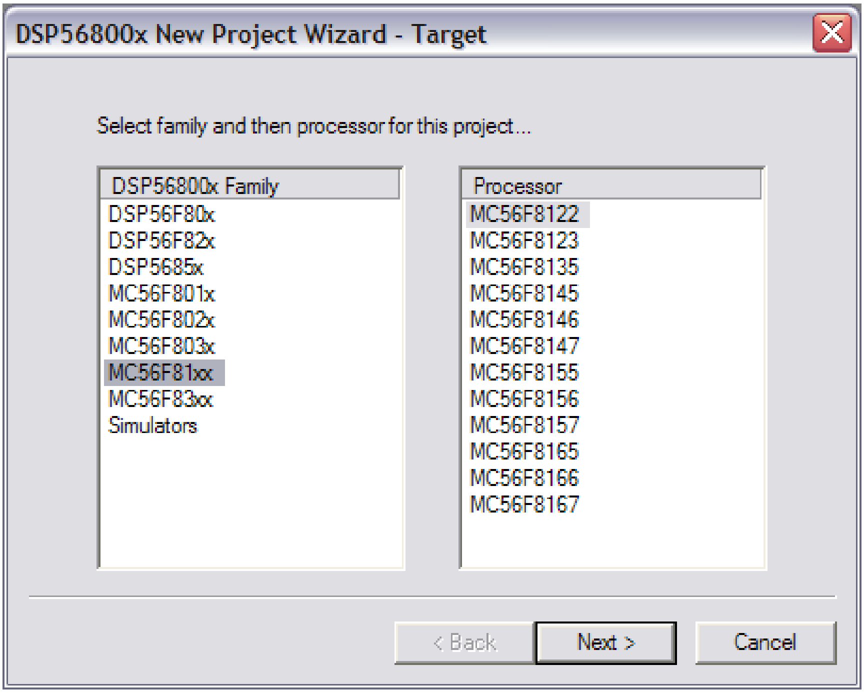


Figure B.10 DSP56800x New Project Wizard Target Dialog Box (MC56F83xx)

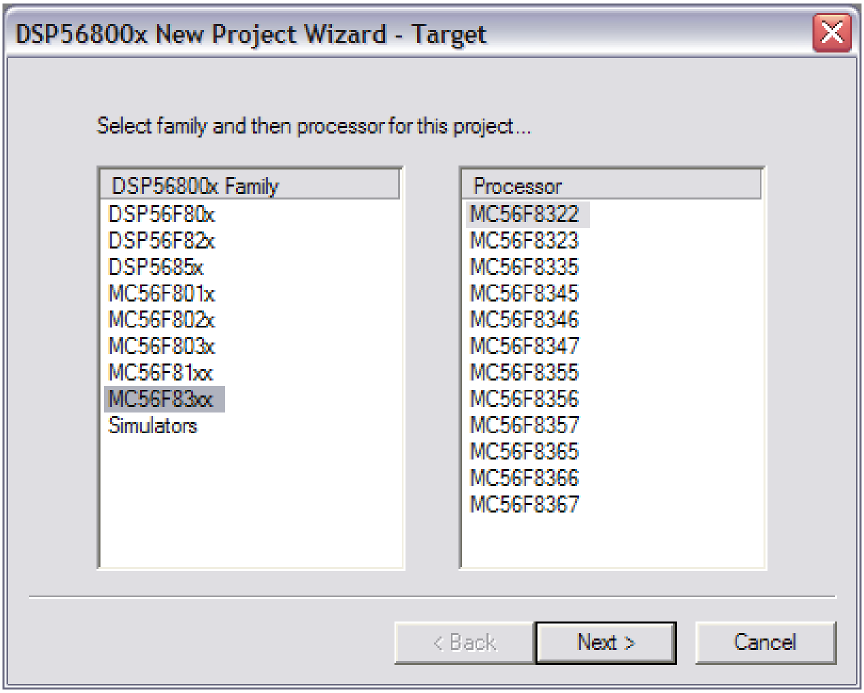
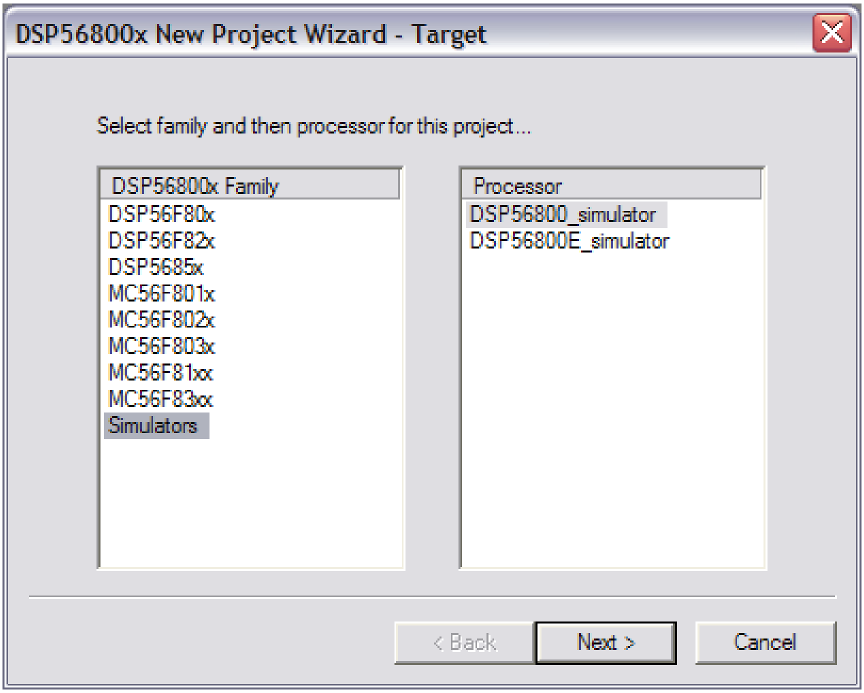


Figure B.11 DSP56800x New Project Wizard Target Dialog Box (Simulators)



One target family and one target processor must be selected before continuing to the next wizard page.

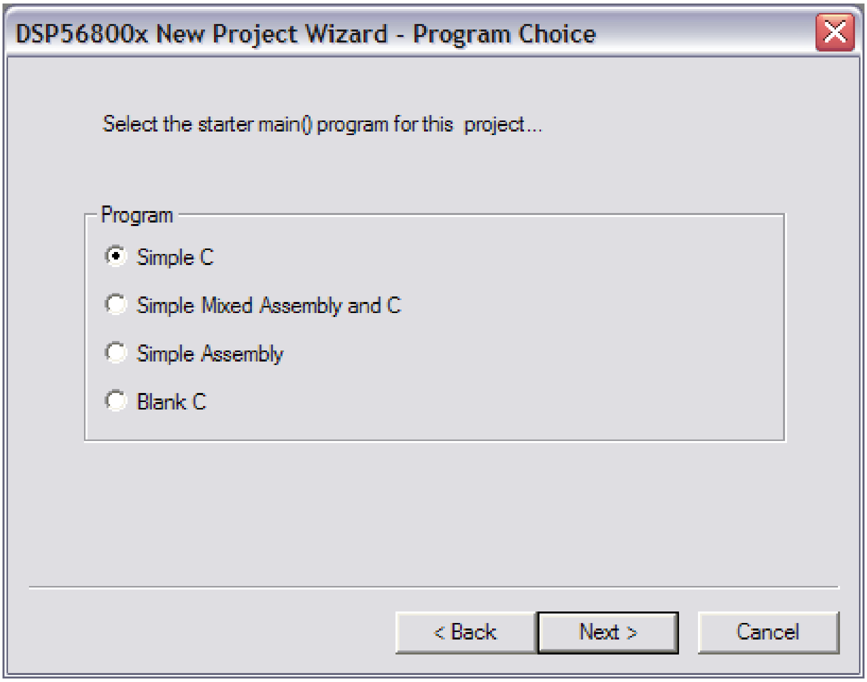
NOTE Depending on which processor you select, different screens will appear according to the “Page Rules” on page 329.

If you choose the simulator, then the DSP56800x New Project Wizard - Program Choice page appears.

Program Choice Page

If you chose either of the simulators, then Figure B.12 on page 344 appears and you can now choose what sort of main() program to include in the project.

Figure B.12 DSP56800x New Project Wizard - Target Choice

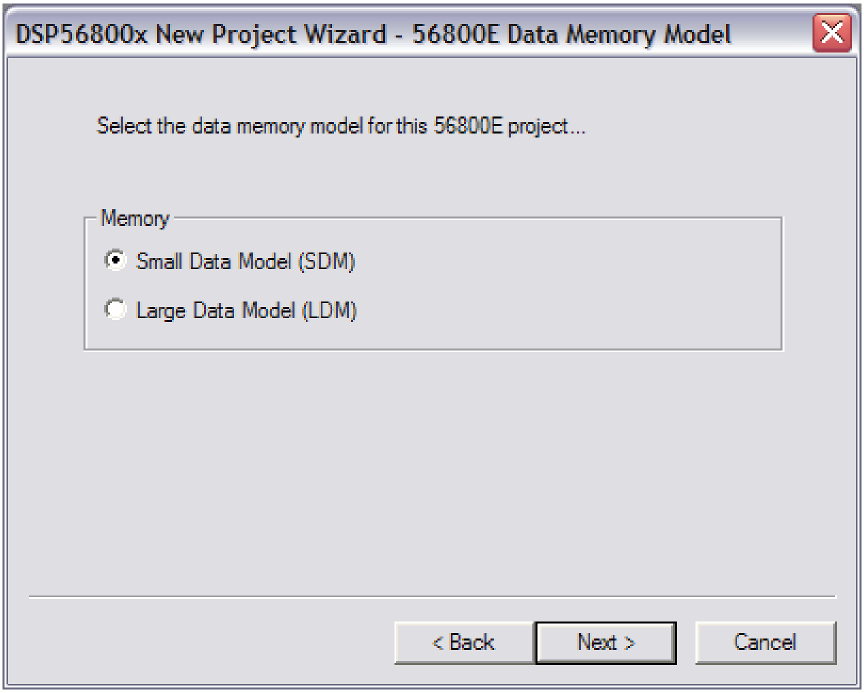


When you click **Next**, the Wizard jumps to the appropriate page determined by the "Page Rules" on page 329.

Data Memory Model Page

If you select a DSP56800E processor (56F83xx or 5685x family), then the Data Memory Model page appears (Figure B.13 on page 345) and you must select either the Small Data Model (SDM) or Large Data Model (LDM).

Figure B.13 DSP56800x New Project Wizard - 56800E Data Memory Model Page



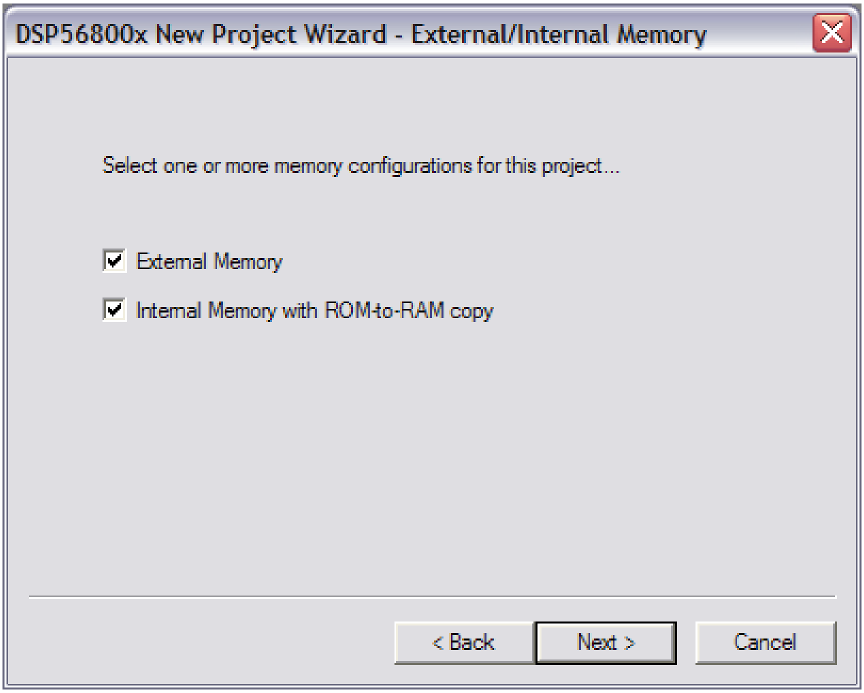
When you click **Next**, the Wizard jumps to the appropriate page determined by the “Page Rules” on page 329.

External/Internal Memory Page

Depending on the processor that you select, the External/Internal Memory page may appear (Figure B.14 on page 346) and you must select either external or internal memory.

NOTE Multiple memory targets can be checked.

Figure B.14 DSP56800x New Project Wizard - External/Internal Memory Page



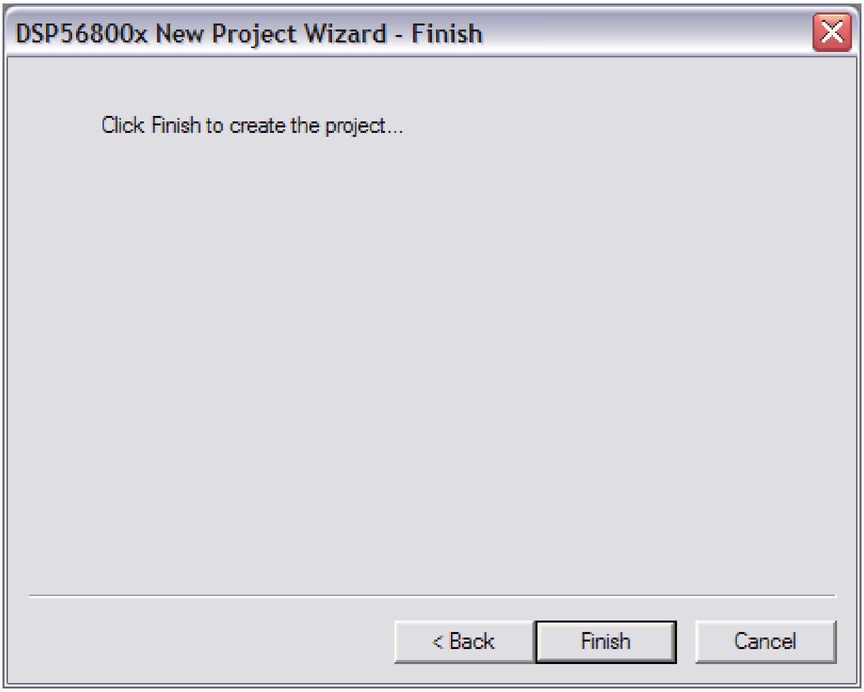
When you click **Next**, the Wizard jumps to the appropriate page determined by the "Page Rules" on page 329.

Finish Page

When you click the **Finish** button on the Finish Page (Figure B.15 on page 347), the project creation process start.

NOTE All target choices end on this page.

Figure B.15 DSP56800x New Project Wizard - Finish Page



Index

M

N

O

P

T