# Table of Contents

1 Introduction
- The CodeWarrior IDE and Its Documentation 9
- New In This Release 10
- References 11

2 Getting Started
- System Requirements 13
  - DSP56800 Hardware Requirements 14
  - Installing the CodeWarrior IDE for DSP56800 16
    - Installing the CodeWarrior IDE 16
    - What Gets Installed 18
  - Installing DSP56800 Hardware 19
    - Using Parallel Port 20
    - Installing the PCI Command Converter 21
    - Installing the Ethernet Command Converter 26
    - Installing ADS UCC with ISA Bus Interface 30
    - Using Serial Port to connect DSP568L811 Board 36
    - Using the Domain Technologies SB-56K 38

3 Development Tools
- Tools Overview 41
  - CodeWarrior IDE 41
  - CodeWarrior Compiler for DSP56800 42
  - CodeWarrior Assembler for DSP56800 42
  - CodeWarrior Linker for DSP56800 42
  - CodeWarrior Debugger for DSP56800 42
- The Development Process 43
  - Project Files versus Makefiles 43
  - Editing Code 44
  - Compiling 44
  - Linking 46
  - Debugging 46
  - Viewing Preprocessor Output 46
Table of Contents

4 Tutorial

CodeWarrior IDE for DSP56800 Tutorial ........................................ 47
Creating a Project ................................................................. 47
Working with the Debugger ..................................................... 59
References .............................................................. 69

5 Target Settings

Target ................................................................. 72
Target Settings Overview .................................................. 72
Displaying Target Settings Panel Window .............................. 72
Changing Target Settings ................................................... 73
Exporting and Importing Panel Options to XML Files .......... 74
Restoring Target Settings ................................................ 74
Target Settings .......................................................... 75
Access Paths ............................................................... 78
Build Extras .............................................................. 82
Runtime Settings ........................................................ 84
File Mappings ............................................................. 86
Source Trees ............................................................... 88
M56800 Target .......................................................... 91
Language ................................................................. 93
C/C++ Language ......................................................... 93
C/C++ Warnings ......................................................... 98
M56800 Assembler ....................................................... 101
Code Generation .......................................................... 103
ELF Disassembler ........................................................ 103
M56800 Processor ........................................................ 106
Global Optimizations .......................................................... 109
Linker ................................................................. 111
M56800 Linker ........................................................... 111
Editor ................................................................. 116
Custom Keywords Panel .................................................. 116
Debugger ................................................................. 120
Other Executables Panel .................................................. 120
Debugger Settings ......................................................... 120
M56800 Target Settings ................................................... 123
# 6 C for DSP56800

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Notes on C</td>
<td>129</td>
</tr>
<tr>
<td>Number Formats</td>
<td>129</td>
</tr>
<tr>
<td>DSP56800 Integer Formats</td>
<td>130</td>
</tr>
<tr>
<td>DSP56800 Floating-Point Formats</td>
<td>131</td>
</tr>
<tr>
<td>DSP56800 Fixed-Point Formats</td>
<td>131</td>
</tr>
<tr>
<td>Calling Conventions, Stack Frames</td>
<td>132</td>
</tr>
<tr>
<td>Calling Conventions</td>
<td>132</td>
</tr>
<tr>
<td>Stack Frame</td>
<td>133</td>
</tr>
<tr>
<td>Code and Data Storage</td>
<td>134</td>
</tr>
<tr>
<td>Optimizing Code</td>
<td>134</td>
</tr>
<tr>
<td>Page 0 Register Assignment</td>
<td>134</td>
</tr>
<tr>
<td>Register Coloring</td>
<td>134</td>
</tr>
<tr>
<td>Array Optimizations</td>
<td>136</td>
</tr>
<tr>
<td>Multiply and Accumulate (MAC) Optimizations</td>
<td>137</td>
</tr>
<tr>
<td>Pragma Directives</td>
<td>139</td>
</tr>
<tr>
<td>Description of Pragma Interrupt</td>
<td>139</td>
</tr>
<tr>
<td>Pragma Optimization</td>
<td>142</td>
</tr>
<tr>
<td>Constant Data Section</td>
<td>143</td>
</tr>
<tr>
<td>Linker Issues</td>
<td>145</td>
</tr>
<tr>
<td>Deadstripping Unused Code and Data</td>
<td>145</td>
</tr>
<tr>
<td>Link Order</td>
<td>145</td>
</tr>
</tbody>
</table>

# 7 Inline Assembly Language and Intrinsic Functions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working With DSP56800 Inline Assembly Language</td>
<td>147</td>
</tr>
<tr>
<td>Inline Assembly Language Syntax for DSP56800</td>
<td>147</td>
</tr>
<tr>
<td>Adding Assembly Language to C Source Code</td>
<td>149</td>
</tr>
<tr>
<td>Assembly Language Quick Guide</td>
<td>150</td>
</tr>
<tr>
<td>Creating Labels for M56800 Assembly</td>
<td>150</td>
</tr>
<tr>
<td>Using Comments in M56800 Assembly</td>
<td>150</td>
</tr>
<tr>
<td>Calling Assembly Language Functions from C Code</td>
<td>151</td>
</tr>
<tr>
<td>Calling Inline Assembly Language Functions</td>
<td>151</td>
</tr>
<tr>
<td>Calling Pure Assembly Language Functions</td>
<td>152</td>
</tr>
<tr>
<td>Calling Functions from Assembly Language</td>
<td>153</td>
</tr>
<tr>
<td>Intrinsic Functions for DSP56800</td>
<td>154</td>
</tr>
<tr>
<td>An Overview of Intrinsic Functions</td>
<td>154</td>
</tr>
<tr>
<td>Fractional Arithmetic</td>
<td>155</td>
</tr>
</tbody>
</table>
Table of Contents

Macros Used with Intrinsics ........................................ 156
List of Intrinsic Functions: Definitions and Examples ........ 156
Absolute/Negate ..................................................... 157
Addition/Subtraction .................................................. 158
Control ................................................................. 160
Conversion ............................................................ 161
Copy .................................................................... 163
Deposit/Extract ...................................................... 164
Division ............................................................... 166
Multiplication/MAC ................................................... 167
Normalization .......................................................... 171
Rounding ............................................................... 173
Shifting ................................................................. 173
Pipeline Restrictions ................................................. 177

8 Debugging .............................................................. 181
Target Settings for Debugging ...................................... 181
M56800 Target Settings .............................................. 182
Command Converter Server ........................................ 192
Essential Target Settings for Command Converter Server .... 192
Changing the Command Converter Server Protocol to Parallel Port ......................................................... 193
Changing the Command Converter Server Protocol to PCI .... 195
DSP56800 Menu ......................................................... 196
Load Default Target .................................................. 196
Load/Save Memory .................................................... 196
Fill Memory ............................................................ 199
Using DSP56800 Simulator ......................................... 201
Cycle/Instruction Count .............................................. 202
Memory Map ........................................................... 203
Watchpoints and Breakpoints ...................................... 203
Change of Flow FIFO Dump ......................................... 210
Global Variable Watchpoints ...................................... 212
Register Details Window ............................................ 215
Viewing Memory ...................................................... 216
Viewing X: Memory ................................................... 216
Viewing P: Memory ................................................... 218
<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
</table>

- Debugging on a Complex Scan Chain ........................................... 220
- Setting Up ....................................................................................... 220
- JTAG Initialization File ................................................................. 221
- Debugging a Loaded Target ............................................................. 221
- System-Level Connect ................................................................. 222

9 ELF Linker and Command Language ........................................ 223
- Structure of Linker Command Files ................................................. 223
- Memory Segment ............................................................................. 224
- Closure Blocks ............................................................................... 224
- Sections Segment ........................................................................... 226
- Linker Command File Syntax ........................................................... 226
- Alignment ......................................................................................... 226
- Arithmetic Operations ...................................................................... 227
- Comments ......................................................................................... 228
- Deadstrip Prevention ....................................................................... 228
- Variables, Expressions and Integral Types ...................................... 228
- File Selection ................................................................................... 231
- Function Selection ........................................................................... 231
- ROM to RAM Copying ....................................................................... 232
- Stack and Heap ................................................................................. 235
- Writing Data Directly to Memory ................................................... 235
- Linker Command File Keyword Listing ......................................... 236
- Sample M56800 Linker Command File ............................................ 246

10 Flash Programming ................................................................. 251
- Setting up the Debugger for Flash Programming .......................... 251
- Setting up the Linker Command File for Flash Programming .......... 252
- Specifying P: Memory ................................................................. 253
- Specifying X: Memory ................................................................. 253
- Preparing the Hardware for Flash Programming ............................ 253
- Flash Programming Tips ............................................................... 254
- Flash Programming the Reset and Interrupt Vectors ..................... 254
# Table of Contents

## 11 Libraries and Runtime Code
- MSL for DSP56800 ............................................. 255
- Using MSL for DSP56800 .................................... 255
- Allocating Stacks and Heaps for the DSP56800 .......... 258
- Runtime Initialization ........................................ 260

## 12 Troubleshooting
- Troubleshooting Tips ......................................... 263
- The Debugger Crashes or Freezes When Stepping Through a REP Statement .................................................. 264
- “Can’t Locate Program Entry On Start” or “Fstart.c Undefined” ................................................................. 264
- When Opening a Recent Project, the CodeWarrior IDE Asks If My Target Needs To Be Rebuilt .......................... 264
- "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" ................................. 265
- IDE Closes Immediately After Opening ........................ 265
- Errors When Assigning Physical Addresses With The \texttt{org} Directive ..................................................... 266
- The Debugger Reports a Plug-in Error ........................ 266
- Windows Reports a Failed Service Startup .................... 266
- No Communication With The Target Board .................... 267
- Downloading Code to DSP Hardware Fails .................... 268
- The CodeWarrior IDE Crashes When Running My Code .......................... 268
- The Debugger Acts Strangely ..................................... 268
- Problems With Notebook Computers ............................ 268
- How to make Parallel Port Command Converter work on Windows® 2000 Machines ................................ .... 269

## A Porting Issues
- Converting the DSP56800 3.x and 4.x Projects to 5.x Projects .......................... 271
- Porting Motorola 56824EVM projects to the CodeWarrior IDE .......................... 272
- Porting DSP56811 to DSP56824 Projects ........................ 277
- Using XDEF and XREF Directives ................................ 277
- Using the ORG Directive ......................................... 277

## Index
- ................................................................. 279

DSP–8

Targeting DSP56800
Introduction

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

This chapter contains the following sections:

- The CodeWarrior IDE and Its Documentation
- New In This Release
- References

The CodeWarrior IDE and Its Documentation

The CodeWarrior IDE has a simple graphical user interface and versatile tools for developing software. Using the CodeWarrior IDE, you can develop a program, plug-in, library, or other code.

The CodeWarrior IDE lets you assemble source-code files, resource files, library files, and configuration settings in a project without writing a complicated build script or makefile. You can also add or delete source-code files from a project using the 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 the host. From that host, you use the CodeWarrior IDE to develop code to target various platforms.

The term target has two meanings in the CodeWarrior IDE:

- Platform Target
  The operating system, processor, or microcontroller for which you write code. If you write code for a particular processor running a specific desktop operating system, you are creating code for a platform target.
• **Build Target**

The settings and files that determine the contents of your code, as well as the process in which this code compiles and links into the final output.

The IDE allows you to specify multiple build targets for a particular platform target. For example, you can compile a debugging version (build target) and an optimized version (build target) of your program for the Windows® operating system (platform target). The build targets can share files in the same project even though each build target uses its own settings. After debugging the program, generating a final version is as simple as selecting the project’s build target and using a single Make command.

The CodeWarrior IDE has compilers, linkers, a debugger, a source-code browser, and editing tools. You can edit, navigate, examine, compile, link, and debug code all within the CodeWarrior IDE. Options for code generation, debugging, and navigation of your project are all configurable in the CodeWarrior IDE.

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 those features unique to the CodeWarrior IDE for the DSP56800.

For a complete understanding of the CodeWarrior IDE, refer to the CodeWarrior IDE User’s Guide for general information and this manual for information specific to DSP56800.

**NOTE**
The CodeWarrior Release Notes contain information about new features, bug fixes, and incompatibilities that do not appear in the documentation because of release deadlines. Release Notes are on the CodeWarrior IDE CD.

**New In This Release**

Refinements in this version are:

• Addition of board-specific projects to the DSP56800 EABI stationery
• Automatic enabling of debugger
• Addition of Command Converter protocol
• Enhancement of pragma interrupt capability
• Addition of File Mappings panel
• Flagging of pipeline dependencies due to occurrence of errors in inline assembler
• Upgrade of stationeries to support all targets in both Flash and RAM
• Option for adjustment of assembler for delayed load of N-registers
• Enhanced setting of hardware breakpoints
• Automatic re-establishment of hardware breakpoints upon launching the debugger
• Addition of option for debugging a target chip within a JTAG chain

References

• The following manuals are included with this product:
  – Code Warrior IDE User Guide
  – Assembler Reference Manual
  – MSL C Reference (Metrowerks standard C libraries)
• To learn more about the DSP56800 processor, refer to the following manuals:
For more information on the various command converters supported by the CodeWarrior IDE for DSP56800, refer to the following manuals:


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

http://www.motorola.com
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**
- **Installing the CodeWarrior IDE for DSP56800**
- **Installing DSP56800 Hardware**

## System Requirements

This section lists system requirements for operating the CodeWarrior IDE for DSP56800 software.

Table 2.1 lists requirements for installing and using the CodeWarrior IDE and the DSP56800 Simulator software.

### Table 2.1 Windows Requirements for the CodeWarrior IDE

| Hardware          | • 133 MHz microprocessor: Intel® Pentium®, AMD™ K6™, or equivalent  
|                   | • 200 MB hard drive space  
|                   | • 32 MB RAM  
|                   | • CD-ROM drive for installation |
| Software          | Windows® 95/98/2000/NT (NT 4.0 recommended) |
DSP56800 Hardware Requirements

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

NOTE Each protocol in Table 2.2 is selected from the M56800 Target Settings panel.

Table 2.2 DSP56800 Hardware Requirements

<table>
<thead>
<tr>
<th>Target Connection</th>
<th>Boards Supported</th>
<th>Hardware Provided With Command Converter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel port on-board Command Converter</td>
<td>56824EVM</td>
<td>• 25-pin parallel-port interface cable</td>
</tr>
<tr>
<td></td>
<td>56F805EVM</td>
<td>• Power supply, 9–12 Vdc, 500 mA with 2.5 mm receptacle (inside positive)</td>
</tr>
<tr>
<td></td>
<td>56F803EVM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>56801EVM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>56807EVM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>56826EVM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>56827EVM</td>
<td></td>
</tr>
<tr>
<td>External Parallel Port Command Converter</td>
<td>56824EVM</td>
<td>• Motorola Parallel Port Command Converter</td>
</tr>
<tr>
<td></td>
<td>56F805EVM</td>
<td>• 25-pin parallel-port interface cable</td>
</tr>
<tr>
<td></td>
<td>56F803EVM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>56L811EVM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>56801EVM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>56807EVM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>56826EVM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>56827EVM</td>
<td></td>
</tr>
<tr>
<td>PCI Command Converter</td>
<td>56824EVM</td>
<td>• 25-pin OCD ribbon cable</td>
</tr>
<tr>
<td></td>
<td>56F805EVM</td>
<td>• Target Interface Module</td>
</tr>
<tr>
<td></td>
<td>56F803EVM</td>
<td>• JTAG 14-pin ribbon interface cable</td>
</tr>
<tr>
<td></td>
<td>56801EVM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>56807EVM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>56826EVM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>56827EVM</td>
<td></td>
</tr>
</tbody>
</table>
### System Requirements

#### Targeting DSP56800

<table>
<thead>
<tr>
<th>Target Connection</th>
<th>Boards Supported</th>
<th>Hardware Provided With Command Converter</th>
</tr>
</thead>
</table>
| Ethernet Command Converter | 56824EVM, 56F805EVM, 56F803EVM, 56F801EVM, 56F807EVM, 56F826EVM, 56F827EVM | • 10 base T RJ-45 cable  
• 25-pin OCD ribbon cable  
• JTAG 14-pin ribbon cable  
• Target Interface Module |
| Motorola Application Development System (ADS) Universal Command Converter (UCC) with the ISA bus interface | 56824ADS, 56F805ADS, 56F803ADS, 56L811ADS, 56801ADS, 56807ADS, 56826ADS, 56827ADS | • One open 16-bit ISA expansion slot  
• One free I/O address: $0100, $0200, or $0300  
• Motorola Application Development System  
• 37-pin ribbon interface cable (included with ADS)  
• 14-pin ribbon interface cable (included with ADS) |
| Domain Technologies SB-56K Multi-DSP Emulator | 56824EVM, 56F805EVM, 56F803EVM, 56L811EVM, 56801EVM, 56807EVM, 56826EVM, 56827EVM | • Domain Technologies SB-56K Multi-DSP Emulator. Refer to the following website for more information:  
• [http://www.domaintec.com](http://www.domaintec.com) |
| Serial port | 56L811EVM | • Motorola DSP56L811EVM board  
• 9-pin serial-port interface cable  
• Power supply, 9–12 Vdc, 500 mA with 2.5 mm receptacle (inside positive) |
Getting Started
Installing the CodeWarrior IDE for DSP56800

Table 2.3 Jumper Settings for Enabling JTAG Communication Interface

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Jumper</th>
<th>Setting from default</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP56L811EVM</td>
<td>JG7</td>
<td>5-6 closed</td>
</tr>
<tr>
<td>DSP56824EVM</td>
<td>JG6</td>
<td>1-2 closed</td>
</tr>
<tr>
<td>DSP56F805EVM</td>
<td>JG5</td>
<td>1-2 closed</td>
</tr>
<tr>
<td>DSP56F803EVM</td>
<td>JG2</td>
<td>1-2 closed</td>
</tr>
<tr>
<td>DSP56F801EVM</td>
<td>JG5</td>
<td>1-2 closed</td>
</tr>
<tr>
<td>DSP56F807EVM</td>
<td>JG4</td>
<td>1-2 closed</td>
</tr>
<tr>
<td>DSP56F826EVM</td>
<td>JG1</td>
<td>1-2 closed</td>
</tr>
<tr>
<td>DSP56F827EVM</td>
<td>JG1</td>
<td>1-2 closed</td>
</tr>
</tbody>
</table>

Installing the CodeWarrior IDE for DSP56800

The CodeWarrior installer automatically installs all the necessary components for you to begin development. If you have any questions about the installer, read the instructions on the CodeWarrior CD.

Installing the CodeWarrior IDE

Install the CodeWarrior IDE:

1. Insert the CodeWarrior CD into your computer’s CD-ROM drive. If Auto Install is disabled on your computer, run the setup.exe program at the root directory on the CD.

2. Follow the CodeWarrior software installation instructions. After installing the CodeWarrior IDE, restart your computer to ensure that the newly installed drivers are available for use.

3. Register the CodeWarrior software. Registered and 30-day evaluation users have separate registering procedures:

   • Registered Users
     Run the Register CodeWarrior program (MWRegister.exe) from the taskbar, located in the CodeWarrior group. This program is also run as part of the installation procedure as the last step.
• 30-day Evaluation Users

Visit the following website and enter your validation code (located on the CodeWarrior tools CD case):

http://www.metrowerks.com/key/eval/

You will receive a license key by email after you submit the evaluation form on the website.

4. Install the license key as follows:

   a. Locate the license.dat file in your CodeWarrior installation folder. The file is located at the root of the CodeWarrior installation directory, so if you have installed the CodeWarrior software at the default installation path, you can find it here:

   C:\Program Files\Metrowerks\CodeWarrior\license.dat

   It is important that this file remain at this location.

   b. Use any standard text editor to open this file. For example, NotePad.

   c. Copy or type the key starting on a new line at the bottom of this file. For example, if your dummy license key was:

   FEATURE dummykey metrowerks 1.000 permanent uncounted 0335B9E8897F\  VENDOR_STRING="Dummy key for placeholder" HOSTID=ANY

   And the existing IDE license key is:

   FEATURE Win32_CWISE_Limited metrowks 4.2 permanent uncounted \  2C3C43468173 HOSTID=ANY
   FEATURE Win32_CWISE_Unlimited metrowks 4.2 permanent uncounted \  D8C287BC5B1B HOSTID=ANY

   After you register with Metrowerks, Inc. (license@metrowerks.com), you will receive a new key. Replace the dummy license key with the new key. For example, if your new license key is:

   FEATURE Win32_Plugins_DSP56800 metrowks 5.0 1-feb-2002 uncounted \  61178DDE2D29 HOSTID=ANY
   FEATURE Win32_Plugins_DSP56800Comp metrowks 5.0 1-feb-2002 uncounted \  2BC235F509D8 HOSTID=ANY
After pasting or typing in the new key, the file contains:

FEATURE Win32_CWIDE_Limited metrowks 4.2 permanent uncounted \ 2C3C43468173 HOSTID=ANY
FEATURE Win32_CWIDE_Unlimited metrowks 4.2 permanent uncounted \ D8C287BC5B1B HOSTID=ANY
FEATURE Win32_Plugins_DSP56800 metrowks 5.0 1-feb-2002 uncounted \ 61178DDE2D29 HOSTID=ANY
FEATURE Win32_Plugins_DSP56800Comp metrowks 5.0 1-feb-2002 uncounted \ 2BC235F509D8 HOSTID=ANY

d. Any future keys can likewise be appended to the bottom of this file.

If you encounter difficulty in installing this key, please contact Metrowerks Customer support at:
Ph: (800) 377-5416
Fax: (512) 873-4901
e-mail: license@metrowerks.com

NOTE
Do not move the license.dat file after installation.

What Gets Installed

Table 2.4 describes the folders that are installed as part of the full standard CodeWarrior IDE for DSP56800 installation. Each folder is located in your CodeWarrior installation directory.

Table 2.4 Contents Installed with the CodeWarrior IDE for DSP56800

<table>
<thead>
<tr>
<th>Directory name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior Manuals</td>
<td>The CodeWarrior documentation tree.</td>
</tr>
<tr>
<td>CodeWarrior Examples</td>
<td>Target specific example projects and code.</td>
</tr>
<tr>
<td>CW Release Notes</td>
<td>Release notes for the CodeWarrior IDE and each tool.</td>
</tr>
</tbody>
</table>
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 User Manual for your product, unless otherwise stated.

**NOTE**
Parallel port connections with boards that support direct parallel port connections are not covered in this chapter. Refer to the Kit Installation Guide or Hardware User Manual for your specific board. You can use the DSP56800 Simulator provided with the CodeWarrior IDE instead of installing additional DSP568xx hardware.
Using Parallel Port

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

Connecting the Parallel Port Cable to 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 2.1).

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.

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

1. Enable the JTAG port.

   Table 2.3 shows 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 2.2).
3. Connect the 25-pin female connector at the other end of the parallel port cable to the 25-pin male connector on the Suite56™ Parallel Port Command Converter module.

4. Locate the 14-pin ribbon cable hanging from the Suite56™ 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 Motorola 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 2.3).
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® 95**

The required files are located in the following directory:

CodeWarrior\DSP EABI Support\Ads PCI Drivers\Win 95 98

1. Install CodeWarrior for DSP56800 Software Development Tools.
2. Shut down your computer.
3. Install the PCI command converter hardware into an empty PCI slot.
4. Turn on your computer.
5. The **Found New Hardware** window appears.
   a. Click the **Driver from Disk Provided from Hardware Manufacturer** box.
   b. Click OK.
6. The **Install from Disk** window appears. Browse to the following directory:
   C:\Program Files\Metrowerks\CodeWarrior\DSP EABI Support\Ads PCI Drivers\Win 95 98

   **NOTE**
   This is the default installation directory. If you changed this directory during the software installation, you will need to select your custom directory. Then, click the **Next** button.
8. Click the **Finish** button.
9. Copy **windrvr.sys** file to \Windows\System32\Drivers.
10. Copy **windrvr.vxd** file to \Windows\System\vmm32.
11. From the command prompt, change to the following directory:
    CodeWarrior\DSP EABI Support\Ads PCI Drivers\Win 95 98
12. Type the following:
    wdreg -name "Macraigor_PCI" -file windrvr install

**Windows® 98**

The required files are located in the following directory:
    CodeWarrior\DSP EABI Support\Ads PCI Drivers\Win 95 98

1. Install CodeWarrior for DSP56800 Software Development Tools.
2. Shut down your computer.
3. Install the PCI command converter hardware into an empty PCI slot.

4. Turn on your computer.

5. The **Add New Hardware Wizard** window appears. Click the **Next** button.

6. Check the **Search** button and then click the **Next** button.

7. Click the **Browse** button.

8. Select the following directory:
   - C:\Program Files\Metrowerks\CodeWarrior\DSP EABI Support\Ads PCI Drivers\Win 95 98

   **NOTE**
   
   This is the default installation directory. If you changed this directory during the software installation, you will need to select your custom directory. Then, click the **Next** button.

   Windows 98 finds the correct driver.

9. Copy the `windrvr.sys` file to `\Windows\System32\Drivers`

10. Copy the `windrvr.vxd` file to `\Windows\System\vmm32`.

11. From the command prompt, change to the following directory:
    - CodeWarrior\DSP EABI Support\Ads PCI Drivers\Win 95 98

12. Type the following:
    
    ```
    wdreg -name "Macraigor_PCI" -file windrvr install
    ```

**Windows NT® 4.0**

The required files are located in the following directory:

- CodeWarrior\DSP EABI Support\Ads PCI Drivers\Win NT

1. Copy the `raptor.inf` file to `/winnt/inf`.
2. Copy the `windrvr.vxd` file to `/winnt/system32/drivers`.
3. Copy the `windrvr.sys` file to `/winnt/system32/drivers`.
4. Install the `raptor.inf` file by right-clicking on this file and selecting the **Install** button.
5. From the command prompt, change to the following directory:
   CodeWarrior\DSP EABI Support\Ads PCI Drivers\Win NT
6. Type the following:
   wdreg -name "Macraigor_PCI" -file windrvr install
7. Shut down your computer.
8. Install the PCI command converter hardware into an empty PCI slot.
9. 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 22 before performing the steps in this section.

Connect the PCI Command Converter to your DSP568xxEVM board:

1. Enable the JTAG port.
   
   Table 2.3 shows the jumpers that you need change from the default configuration for your particular hardware. Refer to the Hardware User Manual or Kit Installation Guide for your particular board for information on default jumper settings.

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.
Installing the Ethernet Command Converter

Connect the Ethernet Command Converter and your Motorola DSP568xxEVM board to your computer as described below.

Configuring Network Settings for Ethernet Command Converter

Connect the Ethernet Command Converter hardware to your computer:

1. Shut down your computer.
2. Connect the 9-pin male connector at one end of an RS-232 serial cable to the 9-pin female connector on the Ethernet Command Converter (Figure 2.4).
3. Connect the 9-pin female connector at the other end to of the RS-232 serial cable to the 9-pin male connector on your computer.
4. Open Hyper Terminal or similar program (Figure 2.5) using the procedure appropriate for the operating system you are using:

- For NT workstations, select **Program > Accessories > Hyper Terminal** from the **Start** menu.
- For Windows® 95 and 98, select **Program > Communications > Hyper Terminal** from the **Start** menu.
5. Set up the Hyper Terminal with the following COM port settings:

Bits per second: 19200

Data bits: 8

Stop bit: 1

Flow control: Xon/Xoff or None

6. Plug the receptacle portion of detachable power cord into the power supply.

7. Insert the plug portion of the detachable power supply into a wall outlet.

8. Connect the 5V power supply cable to the OCDemon™ Ethernet Command Converter (black box).
After a delay of 15 to 20 seconds, the eDemon Command Menu appears. Figure 2.5 shows the eDemon Command Menu using HyperTerminal on Windows NT.

9. Follow the instructions from the eDemon Command Menu to enter the IP Address and other network settings.

Refer to the OCDemon™ Ethernet Command Converter User’s Manual for more information on testing your network connection and firmware upgrades for the Ethernet Command Converter module.

10. After setting up your network settings for the Ethernet Command Converter, disconnect the RS-232 cable from the Command Converter.

**Connecting the Ethernet Command Converter to the DSP568xxEVM Board**

To connect your DSP568xxEVM with the Ethernet Command Converter, follow the steps explained in “Configuring Network Settings for Ethernet Command Converter” on page 26 before performing the steps in this section.

Connect the Ethernet Command Converter to your DSP568xxEVM board:

1. Enable the JTAG port.

   You must enable the JTAG/OnCE port on your hardware. Table 2.3 shows 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 the back panel of the OCDemon™ Ethernet Command Converter (Figure 2.6).

3. Connect the 25-pin female connector at the other end of the 25-pin parallel port cable to the 25-pin male connector on the OCDemon™ target interface module.

4. Locate the 14-pin ribbon cable on the OCDemon™ target interface 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. Connect one end of the RJ-45 base T cable to the Ethernet Command Converter (black box).

6. Connect the other end of the RJ-45 base T cable to the network.

7. Apply power to the DSP568xxEVM.

8. Plug the power supply into a wall socket.

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

   The green LED next to the power connector lights up. The board is now connected.

**Installing ADS UCC with ISA Bus Interface**

Connect the Application Development System (ADS) Universal Command Converter (UCC) and your computer using the Industry Standard Architecture (ISA) bus interface as described below.
Install the Universal Command Converter and ISA Bus

Install the ADS UCC hardware:
1. Place the Motorola Universal Command Converter card on a static-proof mat.
2. Locate an empty card slot in your computer.
3. Insert the Motorola Universal Command Converter card in the empty card slot.
4. Find an open I/O address for the ADS card.
5. Find an open I/O address according to your operating system.

Windows 95 and Windows 98
1. From the Start menu, access the Control Panel. Double-click Systems. Select the Device Manager tab.
2. Click the Properties button. The Computer Properties window appears (Figure 2.7).

Figure 2.7 Computer Properties Window
3. In the **Computer Properties** window, click the **Input/Output** radio button.

4. In the address list, verify that one of the following addresses is unused:
   - 0100 – 0102
   - 0200 – 0202
   - 0300 – 0302

5. If all of these addresses are used, reconfigure your system to accept the ADS card.

6. Close the **Computer Properties** window.

**Windows NT**

1. Click **Start > Programs > Administrative Tools > Windows NT Diagnostics** to open the Windows NT Diagnostics window.

2. Click the **Resources** tab.

3. Click the **I/O Port** button.
4. In the address list (Figure 2.8), verify that one of the following addresses is unused:

- 0100 - 0102
- 0200 - 0202
- 0300 - 0302

If all of these addresses are used, you must reconfigure your system to accept the ADS card as follows:

a. If your Windows NT installation directory is `c:\winnt`, copy the `mdsp.sys` file to the directory:

`c:\winnt\system32\drivers\`

The `mdsp.sys` file is in the following path:
b. On a DOS command line, type \texttt{regini address}, where \texttt{address} reflects the empty address you selected for the card. For example, type \texttt{regini 100} to use I/O address $0100$, which is the default.

5. Shut down your computer.

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

\textbf{Adjusting Jumpers and Making Connections}

1. Adjust the jumper settings on the ISA card.
   
   Adjust the jumper group JG2 to match the I/O address you determined in step 4.
   
   For example, if you want to use $0100 - 0102$, close all jumpers other than jumper A8. If you want to use $0200 - 0202$, close all jumpers other than jumper A9. Refer to Table 2.5.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
To use this I/O address... & CLOSE all JG2 jumpers except... \\
\hline
0100 - 0102 & A8 \\
0200 - 0202 & A9 \\
0300 - 0302 & A8 and A9 \\
\hline
\end{tabular}
\caption{ISA Card Jumper Settings}
\end{table}

2. Verify that all the IRQ jumpers (JG1) are open.

3. Open your computer and locate an empty card slot in your computer.

4. Insert the Motorola ISA interface card into the empty card slot and close your computer.

5. Connect the 37-pin female connector at one end of a 37-pin ribbon cable to the 37-male connector on the ISA card (Figure 2.9).
6. Connect the 37-pin female connector at the other end of the 37-pin ribbon cable to the 37-pin male connector on the ADS Universal Command Converter card.

7. Arrange the Command Converter jumpers according to Table 2.6.

Table 2.6 ADS Universal Command Converter Jumper Settings

<table>
<thead>
<tr>
<th>ADS UCC Jumper Location</th>
<th>Settings Known to Work With CodeWarrior IDE for DSP56800</th>
</tr>
</thead>
<tbody>
<tr>
<td>JG1</td>
<td>Use the factory defaults.</td>
</tr>
<tr>
<td>JG2</td>
<td>1-2, 3-4, and 5-6 CLOSED</td>
</tr>
<tr>
<td>JG3</td>
<td>2-3 CLOSED</td>
</tr>
</tbody>
</table>

8. Turn on your computer.

   The green LED on the ADS Universal Command Converter lights up.

Connect ADS UCC and ISA Bus to DSP568xxEVM Board

To connect your DSP568xxEVM with the ADS Universal Command Converter, follow the steps in “Install the Universal Command Converter and ISA Bus” on page 31 before performing the steps in this section.

Connect the ADS Universal Command Converter to your DSP568xxEVM board (Figure 2.9):

1. Enable the JTAG port.

   You must enable the JTAG/OnCE port on your hardware. Table 2.3 shows the jumpers that you need to change from the default configuration for your particular hardware.

2. Connect 14-pin female connector at one end of a 14-pin ribbon cable to the 14-pin JTAG male connector on the ADS UCC board.

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

3. Connect the 14-pin female connector at the other end of the cable to the 14-pin JTAG male connector on the DSP568xxEVM card.
Ensure that the red stripe on the ribbon cable corresponds to pin 1 on the DSP568xxEVM card.

4. Plug in the power supply into a wall socket.

5. Connect the power supply to the power connector on the DSP568xxEVM card.

The green LED next to the power connector lights up. The board is now connected.

**Figure 2.9** Complete Setup for ADS UCC Setup

**Using Serial Port to connect DSP568L811 Board**

Connect the serial port cable and Motorola DSP568L811 board to your computer as described below.

**Connecting the Serial Port Cable to DSP568L811EVM Board**

1. Arrange all the jumpers as shown in Table 2.7.
Table 2.7  Motorola DSP56L811EVM Jumper Settings

<table>
<thead>
<tr>
<th>Jumper location</th>
<th>Settings Known to Work With CodeWarrior IDE for DSP56800</th>
</tr>
</thead>
<tbody>
<tr>
<td>JG1</td>
<td>1-2 CLOSED; 3-4 OPEN</td>
</tr>
<tr>
<td>JG2</td>
<td>1-2 CLOSED; 3-4 OPEN</td>
</tr>
<tr>
<td>JG3</td>
<td>1-2 OPEN; 3-4 CLOSED</td>
</tr>
<tr>
<td>JG4</td>
<td>1-2 CLOSED; 3-4 OPEN</td>
</tr>
<tr>
<td>JG5</td>
<td>1-2, 3-4, 5-6, and 7-8 CLOSED</td>
</tr>
<tr>
<td>JG6</td>
<td>1-2 OPEN</td>
</tr>
<tr>
<td>JG7</td>
<td>1-3, 2-4 CLOSED, 5-6 OPEN</td>
</tr>
<tr>
<td>JG8</td>
<td>1-2 OPEN</td>
</tr>
<tr>
<td>JG9</td>
<td>2-3 CLOSED</td>
</tr>
<tr>
<td>JG10</td>
<td>1-2 OPEN</td>
</tr>
<tr>
<td>JG11</td>
<td>1-2 CLOSED</td>
</tr>
<tr>
<td>JG12</td>
<td>1-2 OPEN</td>
</tr>
</tbody>
</table>

2. Connect the 9-pin female connector at one end of a 9-pin serial cable to the 9-pin male connector on your computer.

3. Connect the 9-pin male connector at the other end of the serial cable to the 9-pin female connector on the DSP56L811EVM card (Figure 2.10).
Getting Started
Installing DSP56800 Hardware

Figure 2.10 Connecting the DSP56L811EVM Card to Serial Port

4. Plug in the power supply into a wall socket.
5. Connect the power supply to the power connector on the DSP56L811EVM card.

The green LED next to the power connector lights up. The board is now connected.

Using the Domain Technologies SB-56K

Connect the Domain Technologies SB-56K Multi-DSP Emulator and your Motorola DSP568xxEVM board to your computer:

Connect the Domain Technologies SB-56K

1. Locate an empty 9-pin COM port on your computer.
2. Connect one end of a 9-pin female serial cable to the 9-pin male connector on your computer (Figure 2.11).
3. Connect the 9-pin male connector at the other end of the serial cable to the 9-pin female connector on the SB-56K Emulator.

4. Locate the 14-pin ribbon cable hanging from the SB-56K Emulator. 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 in the power supply into a wall socket.
Programming for a DSP56800 board is like programming for any other platform target. If you have never used the CodeWarrior IDE™ before, familiarize yourself with these tools:

- CodeWarrior IDE
- CodeWarrior Compiler Architecture
- CodeWarrior Assembler for DSP56800
- CodeWarrior Linker for DSP56800
- CodeWarrior Debugger for DSP56800

If you are an experienced CodeWarrior IDE user, review the DSP56800 runtime software environment.

Tools Overview

**CodeWarrior IDE**

The CodeWarrior IDE allows you to 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 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, Mac® OS, 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.

**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 further information, refer to the Assembler Reference Manual.

**CodeWarrior Linker for DSP56800**

The CodeWarrior linker for Motorola DSP56800 is an Executable and Linker Format (ELF) linker. This linker lets you generate an ELF file (the default output file format) for your application. This linker also lets you 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. You use the debugger to locate 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, and inspect the contents of the processor’s registers.
For general information about the debugger, including its general features and its visual interface, refer to the IDE User Guide.

The Development Process

While working with the CodeWarrior IDE, you proceed through the development stages familiar to all programmers: write code, compile and link code, and debug code. For complete information on performing tasks like editing, compiling, linking, and debugging, refer to the IDE User Guide.

The difference between the CodeWarrior IDE and traditional command-line environments is in how the software (in this case the IDE) helps you manage your work more effectively. If you are unfamiliar with an integrated development environment in general, or with the CodeWarrior IDE in particular, you will find the topics in this section helpful.

Read these topics to find out how using the CodeWarrior IDE differs from command-line programming:

- Project Files versus Makefiles
- Editing Code
- Compiling
- Linking
- Debugging
- Viewing Preprocessor Output

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 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 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 manage files common to multiple build targets.

The IDE automatically handles the interdependencies 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 IDE also stores compiler and linker settings for each build target. You can modify these settings by using the IDE or by using #pragma statements in your code.

### Editing Code

The CodeWarrior IDE features a text editor designed for programmers. It handles text files in MS-DOS/Windows, UNIX, and Mac OS 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 Metrowerks CodeWarrior IDE window.

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

### Compiling

You can 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 Metrowerks 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 Metrowerks 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.

**CodeWarrior Compiler Architecture**

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. The CodeWarrior IDE build system is depicted in Figure 3.1.

![CodeWarrior Build System](image)

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. Metrowerks publishes this API, allowing developers to create custom or proprietary tools. For more information, go to the Metrowerks Support on the World Wide Web at this URL:

http://www.metrowerks.com/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**

Linking object code into a final binary file is easy: select Project > Make from the menu bar of the Metrowerks CodeWarrior Window. The Make command brings the active project up to date, then links the resulting object code into a final output file.

The IDE controls the linker through 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 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 Metrowerks CodeWarrior Window.

**Viewing Preprocessor Output**

To view preprocessor output, select the file in the project window and click Project > Preprocess from the main menu. The CodeWarrior IDE displays a 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.
This chapter gives you a quick start at learning how to use the CodeWarrior™ IDE for DSP56800.

**CodeWarrior IDE for DSP56800 Tutorial**

This chapter provides a tour of the software development environment of the CodeWarrior IDE for DSP56800. 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
- Working with the Debugger

**Creating a Project**

In this section of the tutorial, you work with the CodeWarrior IDE to create a project.
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 Metrowerks CodeWarrior window appears with a menu bar at the top (Figure 4.1).

Figure 4.1 Metrowerks CodeWarrior Window
Create a new project from project stationery:

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

   The **New** window appears with a list of options in the **Project** tab (Figure 4.2).

![Figure 4.2 New Window](image)

2. Select **DSP56800 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, do not use an empty project because of the complexities involved in using the correct libraries and files and selecting the correct build target settings.

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 Metrowerks CodeWarrior project file. In this tutorial, the file name is `sample.mcp`.

4. Set the location for the project use.

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 4.3)* appears:

![Create New Project Dialog Box](image)

   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.
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 4.4) with a list of board-specific project stationeries.

**Figure 4.4 New Project Window**

6. Select M56824 as the Project Stationery for your target.

7. Click OK in the New Project window.

   A project window appears (Figure 4.5). This window displays all the files and libraries that are part of the project stationery.
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

8. View a source file.

   a. Select the **Files** tab in the project window.
   
   b. Open source and startup files.

   Hierarchical controls are displayed next to folders (groups) in the project window. You can expand or collapse a view.
Click the hierarchical controls next to ‘code’ and ‘support’ to expand and view their contents (Figure 4.6).

**Figure 4.6** CodeWarrior Project Window with Expanded Hierarchical Folders

![CodeWarrior Project Window with Expanded Hierarchical Folders](image)

c. Double-click the **M56800_main.c** file in the project window, the source code in the file is displayed in a CodeWarrior source-code editor window (Figure 4.7).
9. Set the build target.

The CodeWarrior IDE allows you to write code for a variety of microprocessors and operating systems. These are called “build targets.” When you work with a new CodeWarrior project, the first thing you do is specify what your build target is.

a. To specify a build target, double-click the Settings icon in the Project window (see Figure 4.5 for location of icons in the Project window).

The Target Settings window (external RAM (mode 3) Settings in sample) appears (Figure 4.8).
This window contains several different panels. In Figure 4.8, the Target Settings Panels is displayed in the Target Settings window.

**Figure 4.8  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.

If you select M56800 Linker from the Linker list box, the CodeWarrior IDE recognizes that the code you are writing is intended for DSP56800 processors.

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 **M56800 Target** 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 4.9).

![Figure 4.9 M56800 Target Settings Panel](image)

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 4.10).
12. Examine the default settings and select the options according to your requirements. Close the Target Settings window when you are finished by clicking the OK button.


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 4.11). 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.
14. Compile the code using either of the following options:
   - From the menu bar of the Metrowerks CodeWarrior window, select Project > Make.
   - In the project window, double-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 Metrowerks 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 Metrowerks 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 IDE for DSP56800, 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.

3. Access the **Target Settings** window ([Figure 4.9](#)).

4. 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 4.12](#)).

---

**Figure 4.12** Selecting Debugger Settings

![Figure 4.12 selecting debugger settings](image)

5. Select the correct protocol in the **M56800 Settings Panel**:
   - **ADS Command Converter**
Select the **ADS Command Converter** protocol if you are using the ADS Universal Command Converter (UCC).

- **Serial - EVM**
  Select the **Serial - EVM** protocol if you are only using the DSP56L811EVM board with serial interface.

- **Serial - SB56K**
  Select the **Serial - SB56K** protocol if you are using the Domain Technologies SB-56K Multi-DSP Emulator.

- **Parallel Port - ADS or EVM**
  Select the **Parallel Port - ADS or EVM** protocol if you are using the external Motorola Suite56™ Parallel Command Converter or the on-board parallel port interface of the EVM board.

- **Simulator**
  Select the **Simulator** protocol if you want to run your code on the DSP56800 Simulator instead of downloading the code to actual hardware.

- **PCI**
  Select the **PCI** protocol if you are using the Motorola Suite56™ PCI Command Converter.

- **Ethernet**
  Select the **Ethernet** protocol if you are using the Motorola Suite56™ Ethernet Command Converter.

- **Command Converter Server**
  Select the **Command Converter Server** protocol if you want to debug a target with a complex chain locally or to debug remotely.

6. **Set protocol specific options:**

- **Parallel Port - ADS or EVM**
  Select from LPT1, LPT2, LPT3, or LPT4 depending on which parallel port you have used to connect the DSP568xxEVM card.

- **Serial - EVM or SB56K**
  Select from COM 1, COM 2, COM 3, or COM 4, depending on which serial port you used to connect the DSP56L811EVM card or SB-56K Emulator.
• **ADS Command Converter**

  Select the **ADS Base Address**, which is either 0x100, 0x200, or 0x300, depending on the address of the ISA ADS card.

• **Simulator**

  Select the **Simulator** to simulate the DSP56800 processor. The simulator allows selecting bandwidth for CPU usage. Options are Low, Medium, and High.

• **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**

Note that this option is not displayed if you select **Simulator** from the **Protocol** menu.

7. Debug the project by using either of the following options:

• From the Metrowerks 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:

  Downloading external RAM (mode 3) Settings .elf

  When the download to the board is complete, the IDE displays the **Program** window (**external RAM_mode 3 .elf** in sample) shown in **Figure 4.13**.

---

**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 4.13 Program Window

8. 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.

9. 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 4.14), confirming you have set your breakpoint.
To remove the breakpoint, click the red dot. The red dot disappears.

10. View and edit register values.

11. Registers are platform-specific. Different chip architectures have different registers.
   a. From the menu bar of the Metrowerks CodeWarrior window, select View > Registers.

   In this tutorial, the General Purpose Registers window appears (Figure 4.15).
b. To edit values in the register window, double-click a register value. Change the value as you wish.

12. 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 Metrowerks CodeWarrior window.

The Memory window appears (Figure 4.16).

b. Locate the Page list box at the bottom of the View Memory window. Select X Memory from the Page list box.
13. 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 DATA section in the EVM board’s memory. The memory address location for DATA (and CODE) are set in the Memory Segment and Sections Segment 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.


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

The **Memory** window appears (Figure 4.17).

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 4.17 shows Raw Data.

**Figure 4.17** View P:Memory Window

15. Run the debugger.

a. From the menu bar of the Metrowerks CodeWarrior window, select **Project > Run**.

This command executes your code until a breakpoint is reached in the Program window.

b. Display local variables by selecting one from the list and clicking the control to the left of that variable in the Program window (Figure 4.13).

Local variables are displayed in the top-right pane of the Program window.
16. Quit the application.
   
a. Use either of the following options:
   
   • Select **Project > Run**.
   • Click the **Run** icon in the toolbar of the **Program** window.
     
     The code runs to its conclusion. You can now exit the debugger.
   
   b. From the menu bar of the Metrowerks CodeWarrior window, select **Debug > Kill**. This stops the code execution and quits debugging.

**References**

You have completed the tutorial and used the basic elements of the CodeWarrior IDE for DSP56800.

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

Each build target in a CodeWarrior™ project has its own settings. This chapter explains the target settings for DSP56800 software development. These settings affect the CodeWarrior™ DSP56800 compiler, linker, and assembler. Some of the target settings panels are explained in greater detail in the IDE User Guide.

This chapter contains the following sections:

- Target Settings Overview
- Target Settings
- Access Paths
- Build Extras
- Runtime Settings
- File Mappings
- Source Trees
- M56800 Target
- C/C++ Language
- C/C++ Warnings
- M56800 Assembler
- ELF Disassembler
- M56800 Processor
- Global Optimizations
- M56800 Linker
- Custom Keywords Panel
- Other Executables Panel
- Debugger Settings
- M56800 Target Settings
Target

Target Settings Overview

These settings control:

- Compiler options
- Linker output
- Assembler 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.

Displaying Target Settings Panel Window

To display any settings panel for an open project, use either of the following options:

- Select Target Settings from the Edit menu of the Metrowerks CodeWarrior window, where Target is the name of the current build target in the CodeWarrior project.
- Click the Target Settings icon in the project window.

The Target Settings window appears (Figure 5.1).
The **Target Settings** panel window is the most important window in the CodeWarrior IDE. This is where your target operating system or microprocessor is selected.

The left side of the **Target Settings** window contains a list of target settings panels that apply to the current build target. Select the preferred target settings panel. The CodeWarrior IDE displays the target settings panel that you selected.

### Changing Target Settings

To change target settings:

1. Select Edit > Target Name Settings.
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.

**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.

**Restoring Target Settings**

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

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

Only panels appropriate for the current build target are available. The current build target is displayed in a menu in the toolbar of the project window.

**NOTE**

Use the DSP56800 project stationery to create a new project. The stationery already has reasonable or default values for all of the settings panels. You can also create stationery files with your own pre-
ferred settings. Modify a new project to suit your requirements, then save it inside the stationery folder.

Target Settings

The **Target Settings** panel (**Figure 5.2**), 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 Settings Panel](image)

**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 IDE for DSP56800 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 IDE for DSP56800 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** button. The **Please Select an Access Path** dialog box appears (**Figure 5.3**).
Use the dialog box to select the directory to which you want the IDE to save the executable file.

You can specify how the CodeWarrior IDE stores an access path by selecting any of the following options in the Path Type list box of the Please select an access path dialog box:

- **Absolute Path**
  This option allows the IDE to store the access path from the root level of the startup hard drive to the folder whose access path you want to add, including all folders in between. You must update absolute access paths if you move the project to another system, rename the hard disk, or rename any of the folders along the access path.

- **Project Relative**
  This option allows the IDE to store the access path from the folder that contains the project to the folder whose access path you want to add. You do not need to update the project relative access paths if you move a project, as long as the hierarchy of the relative path is the same. You cannot create a relative path to a folder on a different hard drive than where your project file resides.
• Compiler Relative
  This option allows the IDE to store the access path that contains the CodeWarrior IDE to the folder whose access path you want to add. You do not need to update the compiler relative access paths if you move a project, as long as the hierarchy of the relative path is the same. You cannot create a relative path to a folder on a different hard drive than where your CodeWarrior IDE resides.

• Systems Relative
  This option allows the IDE to store the access path from the operating system’s base folder to the folder whose access path you want to add. You do not need to update the systems relative access paths if you move a project, as long as the hierarchy of the relative path is the same. You cannot create a relative path to a folder on a different hard drive than where your operating system’s base folder resides.

4. Click **OK**.

**Save Project Entries Using Relative Paths**

When you check this check box in the **Target Settings** panel, the IDE uses relative paths to locate the files in your project. Relative paths are useful for distinguishing between two or more files with identical names.

**Access Paths**

Use the **Access Paths** panel ([Figure 5.4](#)) for the CodeWarrior IDE to search for additional access paths while compiling and linking. Search works top-down through the access paths. You can add paths that are Absolute, Project Relative, Compiler Relative, or System Relative. If you cannot change the location of your source and library files, you may need to change your access paths if they do not fall into one of the current locations in the **Access Paths** panel.
**User Paths**

Click this radio button to display the User Paths pane in the **Access Paths** panel.

**System Paths**

Click this radio button to display the System Paths pane in the **Access Paths** panel.

**Always Search User Paths**

Enable this check box to search for system header files or interface files.
Add Default

Click this button to restore access paths in the User Paths pane or System Paths pane after you delete them. The restored path appears in the active pane.

Host Flags

Select any of the following options:

- All. Allows all host platforms to use the access path.
- None. Prevents any host from using the access path.
- Mac OS. Allows the IDE to search the access path only on a Mac OS computer.
- Windows. Allows the IDE to search the access path only on a Windows PC.
- UNIX. Allows the IDE to search the access path only on a UNIX workstation.

Add Access Path

To add a new access path, perform the following steps:

1. Select the User Paths pane or System Paths pane in the Access Path panel.
2. Click the Add button.

The Browse for Folder dialog box (Figure 5.5) appears:
3. Use the dialog box to select the folder to which you want to add an access path.

4. You can specify how the CodeWarrior IDE stores an access path by selecting one of the following options in the Path Type list box of the Browse for Folder dialog box (Figure 5.5):
   - Absolute Path
   - Project Relative
   - Compiler Relative
   - Systems Relative

**Change Access Path**

If you change the location of your source and library files, you may need to change your access paths if they do not fall into one of the current selections in the Access Paths panel.

To change an access path, perform the following steps:
1. Select the path in the User Paths pane or System Paths pane.
2. Click the Change button.

   The Browse for Folder dialog box (Figure 5.5) appears.
3. Use this dialog box to select a new access path.

**Remove Access Path**

To remove an access path, perform the following steps:

1. Select the path that you want to remove from the User Paths pane or System Paths pane.
2. Click the **Remove** button to delete the path from the pane.

**Build Extras**

The **Build Extras** panel (**Figure 5.6**) contains various options that affect the way the IDE builds a project, including the use of a third-party debugger.

**Figure 5.6  Build Extras Panel**
**Target Settings**

**Target**

**Extras**

**Use modification date caching**

Check this check box to enable the IDE to check the modification date of each project prior to making the project. Checking the check box causes the IDE to cache the modification dates of the files in a project.

**Cache Subprojects**

Check this check box to improve multi-project updating and linking. The option also allows the IDE to generate symbolics information for both the build targets and the subprojects within each build target. Disable this check box to reduce the amount of memory required by the CodeWarrior IDE.

**Activate Browser**

Check the check box to allow the IDE to generate symbolics information for the project during each build. The symbolics information is required for opening browser windows.

**Dump internal browse information after compile**

Check this check box to review the raw browser information that a plug-in compiler or linker provides for the IDE.

**Use External Debugger**

Check the check box to use an external debugger in place of the CodeWarrior debugger.

**Application**

Click the **Browse** button for the Application box. The **Open** dialog box appears. Locate and select the debugger application from the dialog box.

You must check the check box for **Use External Debugger** to enable the **Browse** button.

**Arguments**

Type command-line arguments to pass to the project at the beginning of a debugging session.
Target Settings
Target

Initial Directory

Click the Browse button. The Please Select an Access Path dialog box appears. Locate and select the path to the external debugger from the dialog box.

You must check the check box for Use External Debugger to enable the Browse button.

Runtime Settings

The Runtime Settings panel (Figure 5.7) includes options for specifying a debugging application for non-executable files, defining a working directory, listing program arguments, and creating environment variables.

Figure 5.7 Runtime Settings Panel
Host Application for Libraries & Code Services

Use the text box for Host Application for Libraries & Code Services to specify a host application when debugging a non-executable field, such as shared library, dynamic link library (DLL), or code resource. The application that you specify is not the debugger application, but rather the application with which the non-executable file interacts.

General Settings

Working Directory

Use this text box to specify the default directory to which the current project has access. Debugging occurs in this location. If you do not specify a directory, debugging occurs in the same field as the executable file.

Program Arguments

Use this text box to type command-line arguments to pass to the project at the beginning of a debugging session. The program receives these arguments after you select Project > Run from the menu bar of the Metrowerks CodeWarrior window.

Environment Settings

This section allows you to specify environment variables to pass to your program as part of the environment parameter in your program’s main() function, or as part of environment calls. These variables are only available to the target program. When your program terminates, the settings are no longer available.

To enable the Environment Settings text, you must make entries in the Variable and Value text boxes at the same time.

Variable

Type a name for the environment variable.

Value

Type a value for the environment variable.
**Add an Environment Variable**

To add an environment variable, perform the following steps:

1. Type a name in the **Variable** text box.
2. Type a value in the **Variable** text box.
3. Click the **Add** button.

The new environment variable appears in the **Environment Settings** text box.

**Change an Environment Variable**

To change an environment variable, perform the following steps:

1. Select an environment variable that you want to change from the **Environment Settings** text box.
2. Change the name in the **Variable** text box.
3. Change the value in the **Value** text boxes.
4. Click the **Change** button.

The changes appear in the **Environment** text box.

**Delete an Environment Variable**

Select an environment variable that you want to remove from the **Environment** text box.

To delete an environment variable, perform the following steps:

1. Select an environment variable that you want to remove from the **Environment Settings** text box.
2. Change the name in the **Variable** text box.
3. Click the **Remove** button. The environment variable is removed from the **Environment Settings** text box.

**File Mappings**

The File Mappings panel ([Figure 5.8](#)) is used to associate a file name extension, such as “.c” or “.p” with a plug-in compiler. The CodeWarrior IDE assigns a compiler to process files with matching file-name extensions.
**Figure 5.8 File Mappings Panel**

**Mapping Info**

*File Type*

Use this text box to type the file type for the selected file mapping in the File Mappings list.

*Extension*

Use this text box to enter a file-name extension, such as the “.c” or “.h” extensions, for a selected file type in the File Mapping list.

Refer to a list of default file-name extensions in Appendix A.

*Compiler*

Use this list box to select a compiler for the selected **File Type** in the File Mapping list.
**Target Settings**

**Target**

**Edit Language**

Select any of the following from the Edit Language list box:

- None
- C/C++
- Disassembly
- Java
- Pascal
- PEF Export List
- Rez
- XML

**Flags**

- Resource File
  
  Select this file to cause the IDE to include in your finished product the resources from the files with the selected file mapping.

- Launchable
  
  Select this file to cause the IDE to open the source-code file with the application that created it after you double-click the file in the Project window.

- Precompiled
  
  Select this file to cause the IDE to compile files with the selected mapping before compiling other files.

- Ignored by Make
  
  Select this file to cause the IDE to ignore files with the selected file mapping when compiling or linking the project.

**Source Trees**

The Source Trees settings panel (Figure 5.9) allows you to define project-specific source trees (root paths). You can define your project’s access paths and build-target output in terms of source
trees. Using this approach, you can share projects across various hosts.

**Figure 5.9 Source Trees Settings Panel**

### Source Trees List

**Name**

This column shows the name of each source tree. When you define access paths in terms of source trees, you use this name in your access path’s definition.

**Path**

This column shows the path to each source tree. You might need to modify the paths of the source trees when you transfer your project to a new host.
Source Tree Info

Name Source Tree

Type a name for a new source tree in the text box, or to change the name of an existing source tree.

Type

Use the list box to select the type of source tree:

- Absolute Path
  This type defines a path from the root level of the hard drive to a desired folder, including all intermediate folders.

- Environment Variable
  This type defines an environment variable in the operating environment (Windows, Solaris, and Linux).

- Registry Key
  This type defines a key entry in the operating-environment registry.

Add Source Tree

To add a new source tree, perform the following steps:

1. Select the type of source tree from the Type menu.
2. Enter a name for the new source tree in the Name text box.
3. If you select Absolute Path in the Type list box, the Choose button is enabled.
   a. Click the Choose button to select a path using a standard dialog box (Figure 5.5).
   b. Click the Add button. The new source tree appears in the Source Trees list.
4. Click Save in the Target Settings window to save your changes.

Change Source Tree

To change a source tree, perform the following steps:

1. Select the source tree that you want to change from the Source Trees list.
2. Click the Change button.
3. Change the name in the Name text box.
4. Change the type of source tree in the Type text boxes.
   The modified source tree name and path for the source tree are displayed in the Source Trees list.
5. Click Save in the Target Settings window to save your changes.

Remove Source Tree

To remove a source tree, perform the following steps:

1. Select the source tree that you want to remove from the Source Trees list.
2. Click the Remove button to delete the source tree.
3. Click Save in the Target Settings window to save your changes.

M56800 Target

The M56800 Target panel (Figure 5.10) 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.
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 5.10**).

**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 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.

## Language

### C/C++ Language

Settings in the **C/C++ Language** panel ([Figure 5.11](#)), only affect C language features implemented for the DSP56800.

The following options are not applicable to the DSP56800 compiler. Disable the options at all times:

- Activate C++ Compiler
- ARM Conformance
- Enable C++ Exceptions
- Enable RTTI
- Pool Strings
- Enable bool support
- Enable wchar_t Support
- Multi-Byte Aware
- EC++ Compatibility Mode
- Enable Objective C
The **C/C++ Language** panel options are:

- **Inline Depth**

  Select this function if you want the compiler to determine whether to inline a function based on the settings of **ANSI Keywords Only** and the **Inline Depth** and **Auto-inline** options.

  **NOTE** When you call an inline function, the compiler inserts the function’s code instead of issuing instructions to call that function. Inline functions makes your programs faster because you execute the function’s code immediately without a function call, but possibly larger because the function’s code may be repeated in several different places.

  If you do not select **ANSI Keywords Only** option, you can declare C functions to be **inline**. The list box for the
Inlining options allows you to select inline no functions, only functions declared inline, or all small functions as shown in Table 5.1.

Table 5.1 Options for Inline Depth Menu

<table>
<thead>
<tr>
<th>Options</th>
<th>Inline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Don’t Inline</td>
<td>Does not inline functions, not even C or C++ declared inline.</td>
</tr>
<tr>
<td>Smart</td>
<td>Inline small functions to a depth of 2 to 4 inline functions deep.</td>
</tr>
<tr>
<td>1 to 8</td>
<td>Always inlines functions to the depth specified by the numerical selection.</td>
</tr>
<tr>
<td>Always Inline</td>
<td>Always inlines functions, no matter the depth.</td>
</tr>
</tbody>
</table>

- **Auto-Inline**
  Select this option to allow the compiler to choose which functions to inline.

- **Deferred Inlining**
  Select this option if you want the compiler to allow inlining of inline and auto-inline functions that are called before these functions are declared.
  The compiler requires more memory for this option.

- **Don’t Reuse Strings**
  Select this option if you want the compiler to store each string literal separately.
  When you do not select this option, the compiler stores only one copy of identical string literals. This option helps you save memory if your program contains identical string literals that you would not modify.
  If you do not select this option and if you change one of the strings, all the strings will be changed.

- **Require Function Prototypes**
  Select this option if you want the compiler to generates an error if you use a function that is defined after it is referenced and does not have a prototype. If the function is implicitly defined, that is, defined before it is referenced, and does not
Target Settings

Language

have a prototype, then the compiler will issue a warning when this option is on. This option helps you to prevent errors that occur when you call a function before you declare or define it. For example, without a function prototype, you may pass data of the wrong type. As a result, your code may not work as you expect even though it compiles without error.

• ANSI Strict

This option affects several extensions to the C language supported by the CodeWarrior compiler. The extensions are:

– C++ style comments
– Unnamed arguments in functions definitions
– A # not followed by argument in a macro
– Using an identifier after #endif
– Using typecasted pointers as values
– Converting pointers to types of the same size
– Arrays of zero length in structures
– The “D” constant suffix

In each case the extension is available only if the option is not selected. If the option is selected, then these extensions to the ANSI C standard are disabled.

• ANSI Keywords Only

Select this option if you want the compiler to generate an error if it encounters any of the CodeWarrior C additional keywords. Use this option if you are writing code that must strictly follow the ANSI/ISO standard.

When this option is not selected, the following additional keywords are available to you:

– asm
  This keyword allows you to use the compiler’s built-in inline assembler.
– inline
  This keyword allows you to declare a C function as inline.

• Expand Trigraphs

Select this option if you want the C compiler to ignore trigraph characters. Many common character constants look
like trigraph sequences (specially on Mac OS), and this extension allows you to use them without including escape characters.

If you are writing code that must follow the ANSI/ISO standard strictly, select this option.

**NOTE** If this option is on, exercise caution when you initialize strings or multi-character constants that contain question marks.

- **Map newlines to CR**
  When you select this option, the C compiler allows you to choose how to interpret the newline (\'\n\') and return (\'\r\') characters.

  In most compilers, ‘\r’ is translated to the value 0x0D, the standard value for carriage return, and ‘\n’ is translated to 0x0A, the standard value for linefeed.

  However, the C compiler in the Macintosh Programmers Workshop, known as MPW C, ‘\r’ is translated to 0x0A and ‘\n’ is translated to 0x0D - the opposite of the typical behavior.

  If you select this option, the compiler uses the MPW C conventions for ‘\n’ and ‘\r’ characters.

  If you do not select this option, the compiler uses the CodeWarrior C and C++ conventions for the ‘\n’ and ‘\r’ characters.

- **Relaxed Pointer Type Rules**
  When you select this option, the compiler treats char* and unsigned char* as the same type. While prototypes are checked for compatible pointer types, direct pointer assignments are allowed.

  This option is useful for if you are using code written before the ANSI/ISO standard. Old source code frequently uses these types interchangeably.

- **Enum Always Int**
  When you select this option, the underlying type is always signed int. All enumerators must be no larger than a signed int. If an enumerated constant is larger than an int, the compiler generates an error.'
However, if you do not select this option, enumerators that can be represented as an unsigned int are implicitly converted to signed int. The compiler chooses the integral data type that supports the largest enumerated constant. The type could be as small as a char or as large as long int.

- **Use Unsigned Chars**

  Select this option to allow the C compiler to treat a char declaration as an unsigned char declaration.

  To check whether this option is on, use _option(unsigned char). By default, this option is off.

  **NOTE** If you select this option, your code may not be compatible with libraries that were compiled with this option turned off.

---

**C/C++ Warnings**

Settings in the C/C++ Warnings panel (Figure 5.12), affect only C language features implemented for DSP56800.

There are no C++ warning features applicable to DSP56800 development.

The following options are not applicable to the DSP56800E compiler. Disable the options at all times:

- Hidden Virtual Functions
- Inconsistent use of ‘class’ and ‘struct’ Keywords
The **C/C++ Language** panel options are:

- **Illegal Pragmas**

  When you select this option, the compiler displays a warning if it encounters an illegal pragma.

**Listing 5.1  Example of Pragma Statements that generate Warnings**

```c
#pragma near_data off     // WARNING: near data is not a pragma
```

- **Empty Declarations**

  When you select this option, the compiler declares a warning if it encounters a declaration with no variable name.
Target Settings
Language

Listing 5.2  Example of Empty Declarations that generate Warnings

<table>
<thead>
<tr>
<th>int ;        // WARNING</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i;       // OK</td>
</tr>
</tbody>
</table>

- Possible Errors
  Select this option if you want the compiler check to for some common typographical mistakes that are legal C syntax, but that may have unwanted side effects, such as putting in unintended semicolons or confusing = and ==. The compiler generates a warning if it encounters one of these:
  - An assignment in a logical expression or the condition in a while, if, or for expression. This check is useful if you frequently use = when you meant to use ==.
  - An equal comparison in a statement that contains a single expression. This check is useful if you frequently use == when you meant to use =.
  - A semicolon (;) directly after a while, if, or for statement.

- Unused Variables
  When you select this option, compiler generates a warning when it encounters a variable that you declare, but do not use. This check helps you find misspelled variable names and variables you have written out of your program.

  If you want to use this warning, but need to declare a variable that you do not use, use the pragma unused statement.

- Unused Arguments
  When you select this option, the compiler generates a warning when it encounters an argument you declare but do not use. This check helps you find misspelled argument names and arguments you have written out of your program.

  There are two ways to avoid this warning:
  - Use the pragma unused statement.
  - You can turn off the ANSI Strict option in the C/C++ Language Panel and not assign a name to the unused argument.

- Extra Commas
When you select this option, the compiler generates a warning when it encounters an extra comma. For example, this statement is legal in C, but it causes a warning when this option is on:

```c
int a[] = {1, 2, 3, 4, }; // ^ WARNING: Extra comma after 4
```

### Listing 5.3  Example of Extra Comma that generate a Warning

- **Extended Error Checking**
  
  When you select this option, the compiler generates a warning (not an error) if it encounters one of the following syntax problems:
  
  - A non-void function that does not contain a `return` statement.
  
  - An integer or floating-point value assigned to an `enum` type.
  
  - An empty `return` statement (`return;`) in a function that is not declared `void`.

- **Implicit Arithmetic Conversions**
  
  When you select this option, the compiler issues a warning if the destination of an operation is not large enough to hold all possible results. For example, assigning the value of a variable type `long` to a variable of type `char` results in a warning if this option is on.

- **Non-Inlined Functions**
  
  Select this option if you want the compiler to issue a warning when it is unable to inline a function.

  If you want to check if the option is on, use `_option`.

### M56800 Assembler

The M56800 Assembler panel ([Figure 5.13](#)) determines the format used for the assembly source files and the code generated by the DSP56800 assembler.
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 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
Target Settings

Code Generation

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 an .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 move X:(Rntoffset),N, N is not available in the instruction following immediately. This option allows the assembler to insert NOP instructions to resolve the restrictions in 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.

**Code Generation**

**ELF Disassembler**

The **ELF Disassembler** panel (Figure 5.14) controls settings related to the disassembly view, which appear when you when you disassemble object files. To view the disassembly of a module, select **Project > Disassemble from the menu bar of the Metrowerks CodeWarrior window.**
The items in this panel are:

**Information on ELF Files**

**Show Headers**

The **Show Headers** option determines whether or not 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 table for the disassembled module.

**Verbose Info**

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

Show Relocations

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

Show Code Modules

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

When 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 for the disassembled module.

This option is displayed only if Show Code Modules 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 Show Code Modules is enabled.

Show Source Code

The Show Source Code option determines whether the disassembler lists the source code for the module presented. 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 **Show Code Modules** is enabled.

Show Data Modules

The **Show Data Modules** option determines whether the disassembler outputs any ELF data sections (such as `.rodata` and `.bss`) for the module that was disassembled.

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 not enabled for DSP56800 because exception tables are not supported.

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

Show Debug Info

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

M56800 Processor

The **M56800 Processor** settings panel (Figure 5.15) determines the kind of code the compiler creates. This panel is available only when the current build target uses the M56800 Linker.
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 Emits 32-bit CMP

This option allows the compiler to use a 32-bit compare instruction. Enable this option for all target CPUs except for the DSP56811.
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.

Global Optimizations

Use the Global Optimizations panel (Figure 5.16), to configure how the compiler rearranges its object code to produce smaller and faster object-code. Some optimizations remove redundant operations in a program, while other optimizations analyze an item’s use in a program. The goal of these optimizations is to improve performance.

NOTE

Use compiler optimizations only after you have debugged your software. Optimizing may break the one-to-one relationship between source and object code.
Optimizing for Space or Speed

You can optimize for space (size of the code in memory) or speed. Although some code optimizations can reduce both size and execution time, there is often a trade-off between these two factors. The Smaller Code Size and Faster Execution Speed settings allow you to control these trade-offs.

For example, if you click the Faster Execution Speed radio button, then the compiler optimizes code for speed even if it adversely affects the code size.

Faster Execution Speed

This option improves the execution speed of object code. With this option on, object code is faster but may be larger. Click the Smaller Code Size radio button to reduce the effect that this option has on a size of a program.
Smaller Code Size

This setting reduces the size of object code that the compiler produces. If you select this radio button, object code is smaller but may be slower.

Linker

M56800 Linker

The M56800 Linker panel (Figure 5.17), controls the behavior of the linker. This panel is available only when the current build target uses the M56800 Linker.

Figure 5.17  M56800 Linker Settings Panel
Target Settings

Linker

**Generate Symbolic Info**

The **Generate Symbolic Info** option controls whether the linker generates debugging information.

When you enable this option, the linker generates debugging information included in the linked ELF file. This setting does not generate a separate file.

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

**NOTE**

If you disable the **Generate Symbolic Info** option, you cannot debug your project using the CodeWarrior debugger. For this reason, the option is enabled 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.

When the option is enabled, the linker includes full path names to the source files. When the option is disabled, the linker uses only the file names. In typical usage, this option is enabled.

This option is available only if you enable **Generate Symbolic Info**.

**Generate Link Map**

The **Generate Link Map** option controls whether the linker generates a link map. The file name for the link map adds the extension **xMAP** to the generated file name. The IDE places the link map in the Debug subdirectory.

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. You can use that information to remove the symbols from the source manually and rebuild the relocatable file in order to make your final process image smaller.

**List Unused Objects**

The **List Unused Objects** option controls whether the linker includes unused objects in the link map.

Enable this 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.

**Show Transitive Closure**

The **Show Transitive Closure** option recursively lists in the link map file all of the objects referenced by `main()`. **Listing 6.1** shows some sample code. To show the effect of the **Show Transitive Closure** option, you must compile the code.

### Listing 5.4  Sample Code to Show Transitive Closure

```c
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 as shown in **Listing 6.2**.
Target Settings

Linker

Listing 5.5  Effects of Show Transitive Closure in Link Map File

# Link map of FSTART_
  1) FSTART_ (func,global) found in MSL C 56800.Lib FSTART.c
  2) FIntVec (notype,global) found in MSL C 56800.Lib Init56811.asm
  2) F_stack_addr (object,global) found in dsp568_heap_stack.c
  2) FInt_Addr (notype,global) found in MSL C 56800.Lib Init56811.asm
  2) Fmain (func,global) found in M56800_main.c
  3) Ffoot (func,global) found in M56800_main.c
  3) Fpad (func,global) found in M56800_main.c
  2) Ffflush (notype,global) found in MSL C 56800.Lib console.asm
  3) __stdout_ready (object,global) found in MSL C 56800.Lib console.c
  3) rtlib.bss.lo (section,local) found in MSL C 56800.Lib console.asm
  3) rtlib.data (section,local) found in MSL C 56800.Lib console.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 generates 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.

When this option is disabled, the linker displays warnings in the **Message** window. When this option is disabled, the linker does not display warnings.

In typical usage, this option is disabled.
**Generate S-Record File**

This option controls whether the linker generates an S-Record file based on the application object image.

The file name for the S-Record adds the `.s` extension to the generated file name. The linker generates S3 type S-Records.

**Sort By Address**

This option enables the compiler to sort S-records generated by the linker using byte address.

**Generate Byte Address**

This option enables the linker to generate S-records in bytes.

**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 you enable **Generate S-Record File**. The maximum value allowed for an S-Record length is 256 bytes.

---

**NOTE**

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

**EOL Character**

The **EOL Character** menu defines the end-of-line character for the S-record file. This field is available only if you enable **Generate S-Record File**. The end-of-line characters are:

- `<cr>` `<lf>` for DOS
- `<cr>` for Mac OS
- `<lf>` for Unix
Entry Point

The **Entry Point** field specifies the function that the linker first uses when the program runs. This function is the program’s starting point.

The default `FSTART_` function is the IDE’s bootstrap or glue code that sets up the DSP56800 environment before your code executes. This function is in the `FSTART.asm` file, which is part of the Metrowerks Standard Library for DSP56800. The `FSTART` function performs other tasks, such as clearing the hardware stack, creating an interrupt table, and fetching the stack start and exception handler addresses.

The final task performed by `FSTART_` is to call your `main()` function.

For the DSP56800 development environment, the `FSTART.asm` file is located in the following path:

`\M56800 Support\Msl\Msl_c\DSP_56800\Src\FSTART.asm`

Force Active Symbols

The **Force Active Symbols** text field allows 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.

Editor

Custom Keywords Panel

The CodeWarrior IDE can use different colors for each type of text. To change these colors, select the **Custom Keywords** panel ([Figure 5.18](#)). This can configure as many as four keyword sets, each with a list of keywords, and syntax coloring for a project.
Figure 5.18  Custom Keywords Panel

Keyword Set

You have two options to select the color:

- **Color** dialog box
- Custom Keywords dialog box

Color Dialog Box

Click the color swatch. The **Color** dialog box appears (Figure 5.19):
**Basic Colors**

Select the color by clicking the rectangular box in the **Color** dialog box.

**Color/Solid**

When you select a color, the selected color appears in the color strip adjacent to the right side of the dialog box. Drag the arrow located on the right of the color strip to change the color shade. Click the **Add to Custom Color** button to save the chosen color shade. The new shade of color appears in one of the rectangle under **Custom Colors**.

**Custom Keywords Dialog Box**

You can define a collection of keywords to which you can assign a unique color. This custom keyword set can include functions, types, and other names that you want to highlight with a particular color. You can enter four sets as shown in the **Custom Keywords** panel (Figure 5.18). Use the **Custom Keywords** dialog box to define, modify, export, and import the sets for use within the IDE.

Click the **Edit** button adjacent to the swatch. The **Custom Keywords** dialog box appears (Figure 5.20).
Target Settings
Editor

Figure 5.20 Custom Keywords Dialog Box

Custom Keywords

Add

1. Type the custom keyword in the text box at the top of the Custom Keywords dialog box. The Add button is enabled.

2. Click the Add button. The custom keyword appears in the dialog box. You can enter up to four sets.

3. Click Done to save the sets and close the dialog box.

Import from file

Using the Import from file option, you can import existing custom keyword sets for use in the IDE.

Export to file

Using the Export to file option, you can export your custom keyword sets for use on another IDE host.
Debugger

Other Executables Panel

The Other Executables panel (Figure 5.21) does not apply to the DSP56800 chip.

![Other Executables Settings Panel](image)

Debugger Settings

The Debugger Settings panel (Figure 5.22) includes options to log activities, change data-update intervals, and set other related options.
Figure 5.22 Debugger Settings Panel

Location of Relocated Libraries and Code Resources

Use any of the following options to enter a path in the Location of Relocated Libraries and Code Resources text box.

- Type the path name in the Location of Relocated Libraries and Code Resources text box.
- Click the Choose button to display a standard dialog box (Figure 5.23) and use the dialog box to select the path.
Choose the Alternate Executable Dialog Box

The path to the selected executable field then appears in the Location of Relocated Libraries and Code Resources text box.

Stop on application launch

Check this check box to stop program execution at a specified temporary breakpoint at the beginning of a debugging session.

Select any of the following three options to stop program execution:

- Click the Program entry point radio button to halt program execution upon entering the program.
- Click the Default language entry point radio button to always stop at the main() function.
- Click the User specified radio button and type the field at which you want to stop.

Other Settings

Auto-target Libraries

This check box applies to the current project when you debug a non-project file. For example, this situation can occur when you attach a running process.
Check the **Auto Target Libraries** check box to allow the IDE to attempt to debug dynamically linked libraries (DLL) loaded by the target application. The IDE attempts to automatically debug the loaded DLLs for which symbolics information is available.

**Cache symbolics between runs**

Check this check box to allow the debugger to cache a project’s symbolics information and refer to that cached information during subsequent debugging sessions. Leave the check box unchecked to force the debugger to discard the project’s symbolics information after each debugging session. Enabling the symbolics cache is useful for improving the performance of successive debugging sessions.

**Log System Messages**

Check this check box to log all system messages to a file. Leave the check box unchecked if you do not wish to create a log file.

**Stop at Watchpoints**

Check this check box to halt a program’s execution when the debugger encounters a watchpoint, regardless of whether the watched value changes. Leave the check box unchecked to halt execution only when the watched value changes.

**NOTE**

Watchpoints always stop regardless of the settings.

**Update Data every x seconds**

Check this check box to update the information in debugging windows while the target is running after a specified time interval. Type in an interval in the **Update Data every x seconds** text box, where x represents the number of seconds you wish to elapse before the next update. Leave the check box unchecked if you do not wish to update the debugging information. In this case, debugging-window information stays the same throughout the debugging session.

**M56800 Target Settings**

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

- **ADS Command Converter**
  Select the **ADS Command Converter** protocol if you are using the ADS Universal Command Converter (UCC).

- **Serial - EVM**
  If you are only using the DSP56L811EVM card with serial interface.

- **Serial - SB56K**
  If you are using the Domain Technologies SB-56K Multi-DSP Emulator.

- **Parallel Port - ADS or EVM**
  Select the **Parallel Port - ADS or EVM** protocol if you are using the external Motorola Suite56™ Parallel Command Converter or the on-board parallel port interface of the EVM board.
• **Simulator**

  If you want to run your code on the DSP56800 Simulator instead of downloading the code to actual hardware.

• **PCI**

  If you are using the Motorola Suite56™ PCI Command Converter with parallel port interface.

• **Ethernet**

  If you are using the Motorola Suite56™ Ethernet Command Converter.

• **Command Converter Server**

  Select the **Command Converter Server** protocol if you want to debug a target with a complex chain locally or to debug remotely.

• **Simulator**

  Select the **Simulator** to simulate the DSP56800 processor. The simulator allows selection of bandwidth for CPU usage. Options are Low, Medium, and High.

### Always Reset on Download

**Always reset on download** determines whether the debugger always resets the DSP hardware before starting a debugging session. If enabled, the debugger automatically resets the hardware before each session. If disabled, the debugger does not reset the hardware.

### Use Flash Config File

When the **Use Flash Config File** option is enabled, you can specify the use of a flash configuration file (**Listing 6.3**) 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 5.25**).
Target Settings
M56800 Target Settings

Figure 5.25 Choose File Dialog Box

Listing 5.6 Flash Configuration File Format in Column Major

<table>
<thead>
<tr>
<th>baseAddr</th>
<th>Addr</th>
<th>startAddr</th>
<th>endAddr</th>
<th>progMem</th>
<th>regBaseAddr</th>
<th>Terase</th>
<th>Tme</th>
<th>Tnvs</th>
<th>Tpgs</th>
<th>Tprog</th>
<th>Tnvh</th>
<th>Tnvh1</th>
<th>Trcv</th>
</tr>
</thead>
</table>

Where:

- 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
Target Settings
M56800 Target Settings

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 Listing 6.4. Do not change the contents of this file.

Listing 5.7  Sample Flash Configuration File for DSP56F803/5

<table>
<thead>
<tr>
<th>Value</th>
<th>Offset</th>
<th>Value</th>
<th>Offset</th>
<th>Value</th>
<th>Offset</th>
<th>Value</th>
<th>Offset</th>
<th>Value</th>
<th>Offset</th>
<th>Value</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0004</td>
<td>0x7dff</td>
<td>1</td>
<td>0x0f40</td>
<td>0x0002</td>
<td>0x0006</td>
<td>0x001A</td>
<td>0x0033</td>
<td>0x0066</td>
<td>0x001A</td>
<td>0x019A</td>
</tr>
<tr>
<td>0</td>
<td>0x8000</td>
<td>0x87ff</td>
<td>1</td>
<td>0x0f80</td>
<td>0x0002</td>
<td>0x0006</td>
<td>0x001A</td>
<td>0x0033</td>
<td>0x0066</td>
<td>0x001A</td>
<td>0x019A</td>
</tr>
<tr>
<td>0</td>
<td>0x1000</td>
<td>0x1ffe</td>
<td>0</td>
<td>0x0f60</td>
<td>0x0002</td>
<td>0x0006</td>
<td>0x001A</td>
<td>0x0033</td>
<td>0x0066</td>
<td>0x001A</td>
<td>0x019A</td>
</tr>
</tbody>
</table>

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.

Debugger sets OMR on launch

Enable Debugger sets OMR on launch to put the board into development mode (setting the OMR register to 0x103). Otherwise, the OMR value is 0x100. This is necessary for boards that do not have jumpers to set the development mode.

NOTE  If you are using Flash ROM, do not enable Debugger sets OMR on launch.

Always Load Program at Debugger Launch

When the Always load program at debugger launch option is enabled, the debugger downloads your object code to the target hardware when you select Project > Debug from the menu bar of the Metrowerks CodeWarrior window. Disable this option when you want to debug a target that is already loaded.
Use Hardware Breakpoints

Enabling the **Use hardware breakpoints** option lets you set a hardware breakpoint in your C source code at one location. To set a hardware breakpoint, enable the **Use hardware breakpoints** option and set a breakpoint in the same manner as setting a software breakpoint.

Use hardware breakpoints only for flash debugging. Using a hardware breakpoint halts execution after an instruction goes into the pipeline. This causes the execution point to "skid" a few instructions past the breakpoint.

**NOTE**

Enabling **Use hardware breakpoints** disables all previously set breakpoints.

Auto-clear previous breakpoint on new breakpoint request

This option is only available when you enable the **Use hardware breakpoints** option. When you also enable the **Auto-clear previous breakpoint on new breakpoint request** and set a breakpoint, the original breakpoint is automatically cleared and the new breakpoint is immediately set. If you disable the **Auto-clear previous breakpoint on new breakpoint request** option and attempt to set another breakpoint, you will be prompted the following message:

![Hardware Breakpoint Already Set](image)

If you click the **Yes** button, the previous breakpoint is cleared and the new breakpoint is set.

If you click the **Yes to all** button, the **Auto-clear previous breakpoint on new breakpoint request** option is enabled and the previously set breakpoint is cleared out without prompting for every subsequent occurrence.

If you click the **No** button, the previous breakpoint is kept and the new breakpoint request is ignored.
This chapter explains the CodeWarrior™ compiler and linker for DSP56800.

This chapter contains the following sections:

- General Notes on C
- Number Formats
- Calling Conventions, Stack Frames
- Code and Data Storage
- Optimizing Code
-Pragma Directives
- Linker Issues

General Notes on C

Note the following on the DSP56800 processors:

- C++ language is not supported.
- Floating-point 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 compiler 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 in the Metrowerks Standard Library (MSL) folder for DSP.

DSP56800 Integer Formats

Table 6.1 shows the sizes and ranges of the data types for the DSP56800 compiler.

Table 6.1 Data Type Ranges

<table>
<thead>
<tr>
<th>Type</th>
<th>Option Setting</th>
<th>Size (bits)</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>n/a</td>
<td>16</td>
<td>true or false</td>
</tr>
<tr>
<td>char</td>
<td>Use Unsigned Chars is disabled in the C/ C++ Language settings panel</td>
<td>16</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td></td>
<td>Use Unsigned Chars is enabled</td>
<td>16</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>signed char</td>
<td>n/a</td>
<td>16</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>unsigned char</td>
<td>n/a</td>
<td>16</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>short</td>
<td>n/a</td>
<td>16</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>n/a</td>
<td>16</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>int</td>
<td>n/a</td>
<td>16</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>unsigned int</td>
<td>n/a</td>
<td>16</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>long</td>
<td>n/a</td>
<td>32</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>unsigned long</td>
<td>n/a</td>
<td>32</td>
<td>0 to 4,294,967,295</td>
</tr>
</tbody>
</table>
DSP56800 Floating-Point Formats

Table 6.2 shows the sizes and ranges of the floating-point types for the DSP56800 compiler.

**Table 6.2 DSP56800 Floating-Point Types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Option Setting</th>
<th>Size (bits)</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer</td>
<td>small-model enabled</td>
<td>16</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td></td>
<td>small-model disabled (not available)</td>
<td>32</td>
<td>0 to 4,294,967,295</td>
</tr>
</tbody>
</table>

DSP56800 Fixed-Point Formats

Table 6.3 shows the sizes and ranges of the fixed-point types for the DSP56800 compiler.

**Table 6.3 DSP56800 Fixed-Point Types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Declared As</th>
<th>Size (bits)</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixed</td>
<td><strong>fixed</strong></td>
<td>16</td>
<td>(-1.0 &lt;= x &lt; 1.0)</td>
</tr>
</tbody>
</table>
NOTE
For compatibility reasons, preferably use DSP intrinsics instead of fixed-point types in Table 6.3 for fractional arithmetic.

### Calling Conventions, Stack Frames

The CodeWarrior IDE for Motorola 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 long fixed-point value is placed in A.
2. The first two 16-bit fixed-point values are placed in Y0 and Y1.
3. The first two 16-bit addresses are placed in R2 and R3.
4. If there are no long fixed-point parameters, the first non-fixed-point 32-bit value or 19-bit address is placed in A.
5. If there are no 16-bit fixed-point parameters, the first two 16-bit non-fixed-point, non-address values are placed in Y0 and Y1 (and Y0 receives the single value when only one is passed).

All remaining parameters are pushed onto the stack, beginning with the rightmost parameter. Multiple-word parameters (19- and 32-bit values) have their 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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Declared As</th>
<th>Size (bits)</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>short fixed</td>
<td><strong>shortfixed</strong></td>
<td>16</td>
<td>(-1.0 &lt;= x &lt; 1.0)</td>
</tr>
<tr>
<td>long fixed</td>
<td><strong>longfixed</strong></td>
<td>32</td>
<td>(-1.0 &lt;= x &lt; 1.0)</td>
</tr>
</tbody>
</table>
The registers \( A, R0, R2, \) and \( Y0 \) are used to return function results as follows:

- Long fixed-point values are returned in \( A \).
- All 19-bit addresses and 32-bit values are returned in \( A \).
- 16-bit addresses are returned in \( R2 \).
- All 16-bit non-address values are returned in \( Y0 \).

### Stack Frame

The stack frame is generated as shown in Figure 6.1. The stack grows upward, meaning that pushing data onto the stack increments the address in the stack pointer.

![Figure 6.1 The Stack Frame](image)

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.
Code and Data Storage

There are two memory sections for the DSP56800: CODE (P memory) and DATA (X memory) (Table 6.4). The compiler places code and data in the appropriate sections. You may need to specify how the program-defined sections map to real memory by using the ELF Linker and Command Language and the M56800 Linker settings panel.

Table 6.4  Code and Data Memory

<table>
<thead>
<tr>
<th>Section</th>
<th>Size</th>
<th>Range (Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE</td>
<td>64K x 16 bit</td>
<td>0000 - FFFF</td>
</tr>
<tr>
<td>DATA</td>
<td>64K x 16 bit</td>
<td>0000 - FFFF</td>
</tr>
</tbody>
</table>

Optimizing Code

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

Page 0 Register Assignment

The compiler uses page 0 address locations 0x30 - 0x40 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 0x30 - 0x37 (page 0 registers MR0-MR7) are volatile registers and can be overwitten. 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.

Register Coloring

The C compiler performs an optimization called register coloring. In this optimization, the compiler may assign two or more register variables to the same register. The compiler does this optimization if the code is not using the two variables at the same time. In this example, the compiler could place i and j in the same register.
Listing 6.1 Register Coloring Example

```c
short i;
int j;

for (i=0; i<100; i++) { MyFunc(i); }
for (j=0; j<100; j++) { MyFunc(j); }
```

However, if a line of code like the one below is placed anywhere in the function, the compiler would realize that you are using \( i \) and \( j \) at the same time, and would place the variables in different registers.

```c
MyFunc (i + j);
```

Register coloring reduces code size and has no effect on execution time.

Using the DSP56800 development tools, you can instruct the compiler to:

1. Store all local variables on the stack.
   - The compiler loads and stores local variables when you read and write to them. This behavior is standard for the compiler.
   - If desired, set the optimization level to Off, level 1, or level 2 in the **Global Optimizations** settings panel.

2. Place as many local variables as possible in registers.
   - In this case, two or more variables whose lifetimes do not overlap can be placed in the same register.
   - If desired, set the optimization level to 3 or 4 in the **Global Optimizations** settings panel.

**NOTE**

At optimization level 3 or 4, you can assign local variables to different registers in different sections of your code. This behavior could produce unexpected results if you compile your code at optimization level 3 or 4 and then attempt to debug your code.

Variables that are declared `volatile` (or those that have the address taken) are not kept in registers.
Array Optimizations

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

In Listing 6.2, the i index is optimized out and the operation performs with address registers.

Listing 6.2  C Code Example for Array Optimizations

```c
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 6.3) and after (Listing 6.4) optimizations are turned on.

Listing 6.3  Array Example Before Optimizations - Mixed View

```assembly
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)
```

DSP–136  Targeting DSP56800
The optimization level has been set to 3 (Listing 6.4). Note that \( i \) is optimized out and the operation is now performed with address registers.

**Listing 6.4  Array Example After Optimizations - Mixed View**

```c
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 6.5 and Listing 6.6.

**Listing 6.5  Sample Multiply and Accumulate Operation**

```c
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 6.6  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 6.7):

Listing 6.7  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)+
}
Pragma Directives

A pragma is a method for modifying compiler settings from the source code rather than the preference panels. Typically, you would use the settings panels to set global options and use pragmas for special cases.

Description of Pragma Interrupt

The pragma interrupt directive controls the compilation of object code for interrupt routines. The compiler generates a special prologue and epilogue for functions so that they may be used to handle interrupts. The contents of the epilogue and prologue vary depending on the mode selected.

The compiler also emits an RTI or RTS for the return statement depending upon the mode selected. The SA, R, and CC bits of the OMR register are set to system default.

To use the interrupt pragma, place the pragma interrupt inside your interrupt handler as shown in Listing 6.8. This causes the function to return with an RTI instruction instead of an RTS.

Listing 6.8 Syntax and Examples

```c
void myIntHandler(){
  #pragma interrupt
  //code...
}
```

There are several ways to use this pragma as described below.

```c
#pragma interrupt saveall|called[warn]
```

The compiler performs the following using the `pragma interrupt [warn]` option:

- Sets M01 to –1 if M01 is used by ISR.
- Sets OMR to system default:
  - Convergent rounding
  - No saturation mode
  - 32-bit compares
• Saves/restores only registers used by ISR.
• Generates an RTI to return from interrupt.
• If \([\text{warn}]\) is present, then emits warnings if this ISR makes calls to functions that have not been defined with pragma called.

**NOTE**
You must use the \([\text{warn}]\) argument only within the scope of function body.

```c
pragma interrupt saveall [warn]
```

The compiler performs the following using the `pragma interrupt saveall[warn]` argument:

• Always sets M01 to –1.
• Sets OMR to system default 0x103:
  – Convergent rounding
  – No saturation mode
  – 32-bit compares
• Saves/restores hardware stack via a runtime call.
• Generates an RTI to return from interrupt
• If \([\text{warn}]\) is present, then emits warnings if this ISR makes calls to functions that have not been defined with pragma called.

**NOTE**
You must use the `saveall[warn]` argument only within the scope of function body.

```c
pragma interrupt called
```

The compiler performs the following using the `pragma interrupt called` argument:

• Saves/restores only registers used by routine.
• Generates an RTS to return from function.

Note the following for the `called` argument:

• You can use the argument on function declarations or within scope of function body.
• You must use this argument before interrupt body is compiled.
• Use this pragma for all functions called by an interrupt routine.

Listing 6.9  Sample Code - #pragma interrupt saveall|called [warn]

```c
#pragma interrupt called
void calledfunction();

int irq1( void )
{
#pragma interrupt warn
...irq1 code
} /* end of ISR--pragma interrupt state is turned off automatically */

int irq2( void )
{
#pragma interrupt saveall warn
...irq2 code
calledfunction();
} /* end of ISR--pragma interrupt state is turned off automatically */

void calledfunction()
{
...code
}

int main ( void )
{
irq1();
irq2();
}
```

NOTE  The end of a function always turns off the pragma interrupt directive.
Pragma Optimization

Listing 6.10  Synopsis - Pragma Optimization

#pragma optimization_level [<level> | reset]

<level> is an integer number between 0 and 4 inclusive

The optimization_level pragma controls the global optimization level programmatically through the #pragma preprocessor statement. The optimization level may be set to any legal value (0-4) using this method. The ‘reset’ option resets the optimization level to its prior value before the #pragma is encountered. This pragma does not affect the interactive preference panel settings for global optimization levels. If this pragma occurs in the middle of a function definition, the entire function is compiled as if the pragma had occurred before the function definition since the actual compilation of the function is deferred until the function is parsed entirely.

Listing 6.11  Sample Code - Pragma Optimization

// Function definition
int afunc ( void )
{

    // Function statements...
    #pragma optimization_level 0
    // Remaining function statements...

    // Restore optimization level to its previous value
    #pragma optimization_level reset

    The entire function 'afunc' is compiled under optimization level 0.
Constant Data Section

By default, the compiler emits const defined data to the .data section. There are two ways to cause the compiler to emit const defined data to the .rodata section:

1. Setting the “write const data to .rodata section” option in the M56800 Processor Settings panel.
   This method emits all const-defined data to the .rodata section.

2. Using #pragma use_rodata [on | off | reset].
   on                           Write const data to .rodata section.
   off                           Write const data to .data section.
   reset                        Toggle pragma state.

To use this pragma, place the pragma before the const data that you wish the compiler to emit to the .rodata section. This method overrides the target setting and allows a subset of constant data to be emitted to or excluded from the .rodata section.

Listing 6.12  Synopsis - Pragma use_rodata

#pragma use_rodata on|off|reset

By default, the compiler emits const defined data to the .data section. There are two ways to cause the compiler to emit const defined data to the .rodata section:

1. Setting the “write const data to .rodata section” option in the M56800 Processor Settings panel.
   This method emits all const-defined data to the .rodata section.

2. Using #pragma use_rodata [on | off | reset].
   on                           Write const data to .rodata section.
   off                           Write const data to .data section.
   reset                        Toggle pragma state.

To use this pragma, place the pragma before the const data that you wish the compiler to emit to the .rodata section. This method
overrides the target setting and allows a subset of constant data to be emitted to or excluded from the .rodata section.

Listing 6.13  Sample Code _ Pragma use_rodata

```c
const UInt16 len_l_mult_ls_data = sizeof(l_mult_ls_data) / sizeof(Frac32) ;
const Int16 g = a+b+c;

#pragma use_rodata on
const Int16    d[]={0xdddd};
const Int16    e[]={0xeeee};
const Int16    f[]={0xffff};
#pragma use_rodata off

main()
{
     // ... code
}
```

You must add .rodata section information to the linker command file.

Listing 6.14  Sample Linker Command File - Pragma use_rodata

```c
MEMORY {
   .text   (RWX) : ORIGIN = 0x2000, LENGTH = 0x00000000
   .data   (RW)  : ORIGIN = 0x3000, LENGTH = 0x00000000
   .rodata (R)  : ORIGIN = 0x5000, LENGTH = 0x00000000
}
SECTIONS {
   .main_application :
   {
      # .text sections
   } > .text

   .main_application_data :
   {
      # .data sections
      # .bss sections
   } > .data
```
Linker Issues

This section explains background information on the DSP56800 linker and its operation.

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.

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

Working With DSP56800 Inline Assembly Language

This section explains how to use the CodeWarrior compiler’s for inline assembly language programming, including assembly language syntax.

This chapter contains the following sections:

• Working With DSP56800 Inline Assembly Language
• Calling Assembly Language Functions from C Code
• Calling Functions from Assembly Language
• Intrinsic Functions for DSP56800

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 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 7.1.

**Listing 7.1  Function-level Syntax**

```
asm <function header> 
    {  
        <local declarations> 
        <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 7.2. The inline assembly language statement is any valid assembly language statement.

**Listing 7.2  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 parenthesis "()" 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 7.3 and Listing 7.7. Note that if you are using this method, you must define local variables within the function.

**Listing 7.3  Defining a Function with `asm`**

```c
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 7.4. Note that if you are using this method, you must not define local variables within the inline `asm` statement.

**Listing 7.4  Inline Assembly with `asm`**

```c
long MyInlineAsmFunction(void)
{
    asm { move x:(r0)+,x0 }
}
```
Assembly Language Quick Guide

Keep these points in mind as you write 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 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 7.5  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 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 7.6.

Listing 7.6  Comments Allowed in M56800 In-line Assembly Language

move x:(r3),y0     # ERROR
add  x0,y0         //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 pure 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 7.7 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 7.7  Sample Code - Creating an Inline Assembly Language Function**

```c
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 7.8.
Listing 7.8  Sample Code - Calling an Inline Assembly Language Function

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

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

### Calling Pure 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 7.9. 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 7.9  Sample Code - Creating an Assembly Language Function

```assembly
;"my_assym.asm"

SECTION user
ORG P:                         ;map to user defined section in CODE
                                 ;put the following program in P
                                 ;memory

GLOBALF pmemwrite             ;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
```
In the source code, the compiler prepends the letter 'F' to every function label name. You can now call your assembly language function from C, as shown in Listing 7.10.

Listing 7.10 Sample Code - Calling an Assembly Language Function from C

```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:

```c
void foot( void ) {
    /* Do something */
```
Intrinsic Functions for DSP56800

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

- An Overview of Intrinsic Functions
- Fractional Arithmetic
- Macros Used with Intrinsics

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 are an extension provided by the CodeWarrior compiler.

**Fractional Arithmetic**

Many of the intrinsic functions for Motorola 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:

\[
\text{Fractional Value} = \frac{\text{Integer Value}}{2^{15}}
\]

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

\[
\text{Fractional Value} = \frac{\text{Long Integer Value}}{2^{31}}
\]

**Table 7.1** shows how both 16 and 32-bit values can be interpreted as either fractional or integer values.

<table>
<thead>
<tr>
<th>Type</th>
<th>Hex</th>
<th>Integer Value</th>
<th>Fixed-point Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>short int</td>
<td>0x2000</td>
<td>8192</td>
<td>0.25</td>
</tr>
<tr>
<td>short int</td>
<td>0xE000</td>
<td>-8192</td>
<td>-0.25</td>
</tr>
<tr>
<td>long int</td>
<td>0x20000000</td>
<td>536870912</td>
<td>0.25</td>
</tr>
<tr>
<td>long int</td>
<td>0xE000000</td>
<td>-536870912</td>
<td>-0.25</td>
</tr>
</tbody>
</table>
Macros Used with Intrinsics

These macros are used in intrinsic functions:

- Word32. A macro for signed long.

List of Intrinsic Functions: Definitions and Examples

The intrinsic functions supported by the DSP56800 are shown in Table 7.2. However, please refer to intrinsics_56800.h for any last minute changes.

Table 7.2 Intrinsic Functions for DSP56800

<table>
<thead>
<tr>
<th>Category</th>
<th>Function</th>
<th>Category</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute/Negate</td>
<td>__abs</td>
<td>Multiplication/MAC</td>
<td>__mac_r</td>
</tr>
<tr>
<td></td>
<td>_negate</td>
<td></td>
<td>__msu_r</td>
</tr>
<tr>
<td></td>
<td>_L_negate</td>
<td></td>
<td>__mult</td>
</tr>
<tr>
<td>Addition/Subtraction</td>
<td>__add</td>
<td></td>
<td>__mult_r</td>
</tr>
<tr>
<td></td>
<td>_sub</td>
<td></td>
<td>_L_mac</td>
</tr>
<tr>
<td></td>
<td>_L_add</td>
<td></td>
<td>_L_msu</td>
</tr>
<tr>
<td></td>
<td>_L_sub</td>
<td></td>
<td>_L_mult</td>
</tr>
<tr>
<td>Control</td>
<td>__stop</td>
<td></td>
<td>_L_mult_l</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>_s</td>
</tr>
</tbody>
</table>
Absolute/Negate

- __abs
- __negate
- _L_negate

__abs

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

**Assumption**

**Prototype**

```c
int __abs( int );
```
**Inline Assembly Language and Intrinsic Functions**

*List of Intrinsic Functions: Definitions and Examples*

---

**Example**

```c
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**
```c
Word16 __negate(Word16 svar1)
```

**Example**
```c
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**
```c
Word32 _L_negate(Word32 lvar1)
```

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

---

### Addition/Subtraction

- __add
- __sub
- _L_add
- _L_sub

---

DSP–158  
Targeting DSP56800
## __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
Inline Assembly Language and Intrinsic Functions
List of Intrinsic Functions: Definitions and Examples

_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

DSP–160  Targeting DSP56800
__stop

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

Prototype  void __stop(void)

Usage  __stop();

Conversion

- __fixed2int
- __fixed2long
- __fixed2short
- __int2fixed
- __labs
- __long2fixed
- __short2fixed

__fixed2int

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

Prototype  int __fixed2int ( __fixed__ );

Example  int  i;
          __fixed__  i_fix = 0.645;
          i = __fixed2int( i_fix ); /* Returns 21135 */

__fixed2long

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

Prototype  long __fixed2long ( __longfixed__ );
### __fixed2short

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

**Prototype**
```
short __fixed2short ( __shortfixed__ );
```

**Example**
```c
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**
```c
int i = 2; __fixed__ ifix;

/* Returns 0.0000610 (2^{-14})*/
ifix = __int2fixed( i );
```

### __labs

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

**Prototype**
```
long __labs ( long );
```

**Example**
```c
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
- __strcpy

---

**__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 );

_DEFINITION

__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
- __extract_l
- _L_deposit_h
- _L_deposit_l
**__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
• __div_ls

__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 positive of 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**

```c
Word16 __div_s(Word16 s_denominator, Word16 s_numerator)
```

**Example**

```c
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
- __msu_r
- __mult
- __mult_r
- _L_mac
- _L_msu
- _L_mult
- _L_mult_ls

**__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 = 0x0x0000FFFF;

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: 0.625 = 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**
```c
Word32 _L_mac(Word32 laccum, Word16 sinp1, Word16 sinp2)
```

**Example**
```c
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**
```c
Word32 _L_msu(Word32 laccum, Word16 sinp1, Word16 sinp2)
```

**Example**
```c
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
```
Inline Assembly Language and Intrinsic Functions
List of Intrinsic Functions: Definitions and Examples

_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: 0.625 = 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(l1,s2);
         // Expected value of result: 0.625 = 0x08000000

Normalization
•  __norm_l
__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_s which is more optimal but generates a different value for the case where the input == 0x0000.

Prototype  Word16 __norm_s(Word16 ssrcl)

Example    short s1 = 0x2000; /* .25 */
short result;

result = __norm_s(s1);
// Expected value of result: 1

## Rounding

__round

### __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 16bit = 0xFFFF > 0x8000 then add 1 */
short result;

result = __round(l);
// Expected value of result: 0x1235

## Shifting

- __shl
- __shr
- __shr_r
- _L_shl
- _L_shr
- _L_shr_r
__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**
```c
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**
```c
short result;
```

DSP–174

Targeting DSP56800
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.
Inline Assembly Language and Intrinsic Functions
List of Intrinsic Functions: Definitions and Examples

Note
This operation is not optimal on the DSP56800 because of the saturation requirements and the bidirectional capability. See the intrinsic \_L\_shl or result = shlfts(l, s1); 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 access 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
```
This chapter explains the generic features of the CodeWarrior™
debugger and additional features specific to DSP56800 debugging.

This chapter contains the following sections:

- **Target Settings for Debugging**
- **Command Converter Server**
- **DSP56800 Menu**
- **Using DSP56800 Simulator**
- **Watchpoints and Breakpoints**
- **Global Variable Watchpoints**
- **Register Details Window**
- **Viewing Memory**
- **Debugging on a Complex Scan Chain**
- **System-Level Connect**

**Target Settings for Debugging**

This section explains how to control the debugger’s behavior by
modifying the appropriate settings panels.

To properly debug DSP56800 software, you must set certain
preferences in the **Target Settings** window.

The **M56800 Target** settings panel, which is described below, is
specific to DSP56800 development. The remaining settings panels
are generic to all build targets. In addition, other settings panels can
affect debugging. **Table 8.1** lists these panels.
Table 8.1 Settings Panels That Can Affect Debugging

<table>
<thead>
<tr>
<th>This panel…</th>
<th>Affects this…</th>
<th>See…</th>
</tr>
</thead>
<tbody>
<tr>
<td>M56800 Linker</td>
<td>Symbolics, linker warnings</td>
<td>“M56800 Linker,”</td>
</tr>
<tr>
<td>M56800 Processor</td>
<td>Optimizations</td>
<td>“M56800 Processor,”</td>
</tr>
<tr>
<td>Debugger Settings</td>
<td>Debugging options</td>
<td>“Debugger Settings”</td>
</tr>
</tbody>
</table>

M56800 Target Settings

The M56800 Target settings panel is unique to DSP56800 debugging. The options available in this panel depend on the communications protocol you use.

The Protocol menu determines the communications protocol the debugger uses when downloading your application into memory. The options available depend on the protocol you are using.

Table 8.2 lists the protocols supported by the CodeWarrior debugger for DSP56800. Refer to a section for settings specific to your chosen protocol.

Table 8.2 Debugging Protocols Supported

<table>
<thead>
<tr>
<th>Debug protocol option</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADS Command Converter</td>
<td>Select this option if you are communicating through the JTAG interface with the UCC. The Target Settings window presents the options shown in Figure 8.1.</td>
</tr>
<tr>
<td>Serial - EVM</td>
<td>Select this option if you use the DSP56L811EVM board. The Target Settings window presents the options shown in Figure 8.2.</td>
</tr>
<tr>
<td>Serial - SB56K</td>
<td>Select this option if you are using the Domain Technologies SB-56K Multi-DSP Emulator. The Target Settings window presents the options shown in Figure 8.3.</td>
</tr>
<tr>
<td>Simulator</td>
<td>Select this option if you want to use the DSP56800 Simulator. The Target Settings window presents the options shown in Figure 8.4.</td>
</tr>
<tr>
<td>Parallel Port - ADS or EVM</td>
<td>Select this option if you are using either the Motorola Parallel Port Command Converter or a direct parallel port connection. The Target Settings window presents the options shown in Figure 8.5.</td>
</tr>
<tr>
<td>PCI Command Converter</td>
<td>Select this option if you are using the PCI Command Converter. The Target Settings window presents the options shown in Figure 8.6</td>
</tr>
</tbody>
</table>
### Target Settings for Debugging

**Targeting DSP56800**

**NOTE** Refer to section on “Debugger Settings,” for detail description of options common to all the protocol boards.

<table>
<thead>
<tr>
<th>Debug protocol option</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ethernet Command Converter</td>
<td>Select this option if you are using the Ethernet Command Converter. The <strong>Target Settings</strong> window presents the options shown in Figure 8.7</td>
</tr>
<tr>
<td>Command Converter Server</td>
<td>Select this option if you are using the Command Converter Server. The <strong>Target Settings</strong> window presents the options shown in Figure 8.8</td>
</tr>
</tbody>
</table>

**ADS Command Converter**

*Figure 8.1* shows the **M56800 Target Settings** panel after you select **ADS Command Converter** from the **Protocol** menu.

The **ADS Base Address** menu lets you select an I/O address for the ISA card. This address can be 0×100, 0×200, or 0×300, depending on the address you selected when installing the card.
Serial - EVM

Figure 8.2 shows the **M56800 Target Settings** panel when you select **Serial - EVM from the Protocol menu**. This panel provides the same options when you select **Serial - SB56K**.

The **COM Port** menu lets you select the serial port for the DSP568xx card or the Domain Technologies SB-56K Emulator. This port can be COM 1, COM 2, COM 3, or COM 4, depending on the port you selected when installing the hardware.
Target Settings for Debugging

Figure 8.2 M56800 Target Settings Panel - Serial - EVM Connections

Figure 8.3 shows the M56800 Target Settings panel when you select Serial - SB56K from the Protocol menu. This panel provides the same options when you select Serial - EVM.

The COM Port menu lets you select the serial port for the DSP568xx card or the Domain Technologies SB-56K Emulator. This port can be COM 1, COM 2, COM 3, or COM 4, depending on the port you selected when installing the hardware.
Figure 8.3  M56800 Target Settings Panel - Serial SB - 56K Connections

Simulator

Figure 8.4 shows the M56800 Target Settings panel when you select Simulator from the Protocol menu.

Figure 8.4  M56800 Target Settings Panel - Simulator
Parallel Port - ADS or EVM

Figure 8.5 shows the M56800 Target Settings panel when you select Parallel Port - ADS or EVM from the Protocol menu.

The Parallel Port menu lets you select the port for the Motorola Parallel Port Command Converter. This port can be LPT 1, LPT 2, LPT 3, or LPT 4, depending on the port you selected when installing the hardware.
**Debugging**  
*Target Settings for Debugging*

**Figure 8.5** M56800 Target Settings Panel - Parallel Port - ADS or EVM

**PCI Command Converter**

*Figure 8.6* shows the **M56800 Target Settings** panel when you select **PCI Command Converter** from the **Protocol** menu.

When using the PCI Command Converter, no additional selections need to be made as the CodeWarrior debugger automatically detects this device when this protocol is chosen.
Figure 8.6 M56800 Target Settings Panel - PCI Command Converter

Ethernet Command Converter

Figure 8.7 shows the M56800 Target Settings panel when you select Ethernet Command Converter from the Protocol menu.

Once you have flashed an IP address into the Ethernet Command Converter, you can specify that address in the IP Address text box. Any valid local or remote IP address is acceptable.
Command Converter Server

Figure 8.8 shows the M56800 Target Settings panel when you select Command Converter Server from the Protocol menu.
Connect to Remote CCS

Select this checkbox to specify that the CodeWarrior IDE should connect to a remote command converter server. Otherwise, the IDE starts the command converter server locally.

- **IP Address**
  
  Use this text box to specify the IP address where the command converter server resides when running the command converter server from a remote location on the network.

- **Port**
  
  The port to which the command converter server listens. This field defaults to port 41475.

If you check the **Custom JTAG** checkbox, the following options appear:
• JTAG Init. File
  A JTAG initialization file specifies the name and order of the boards you are debugging.
• JTAG Target Core Index
  This value specifies the location of your target processor in the JTAG chain. The first board in the chain has an index value of 0, the second board has an index value of 1, and so on.

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, if you wish to start CCS without launching a debug session, you may do so by selecting Command Converter Server from the Protocol text box in the M56800 Target Settings panel.

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 Metrowerks Command Converter Server window (Figure 8.11).

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 IDE for DSP56800.
• 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.

• Default Settings

By default, the command converter server listens on port 41475. You can specify a different port number in the M56800 Target panel for the debugger to connect to if needed. 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.
• No user 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 IDE for DSP56800, you can change the port.

Change the parallel port:

1. 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 8.9):

![Figure 8.9 Command Converter Server Icon](image)

A menu appears (Figure 8.10):
2. Select **Show console** from the menu.

3. The **Metrowerks Command Converter Server** window appears (Figure 8.11):

4. On the console command line, type the following command:
   
   ```
   delete all
   ```

5. Press **Enter**.

6. Type the following command, substituting the number of the parallel port to use (for example, 1 for LPT1):
   ```
   config cc parallel:1
   ```

7. 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 8.9.

2. From the menu shown Figure 8.10, select Show Console.

3. At the console command line in the Metrowerks Command Converter Server window shown in Figure 8.11, type the following command:
   
   delete all

4. Press Enter.

5. Type the following command:

   config cc pci

6. Press Enter.

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 proper protocol configuration using instructions described in the section Essential Target Settings for Command Converter Server.

3. In the M56800 Target Settings panel for the debugger, do the following:

   a. Check the Connect to Remote CCS checkbox.

   The text boxes for IP Address and Port are now enabled.

   b. In the IP Address text box, type the IP address or machine name if on the network. If you leave the text box for the IP address blank, it will default to the local host machine.

   c. In the Port text box, type the port address. If you leave the text box blank, the CCS uses the default value.

NOTE: The default port number is 41475.
4. Ensure that the debugger is enabled on your computer.
5. Launch the debugger.

**DSP56800 Menu**

The DSP56800 menu offers these selections:

**Load Default Target**

From the menu bar of the Metrowerks CodeWarrior window, select **DSP56800 > Load default target** to load the target program without launching the debugger.

This feature performs the following:

- Retrieves the settings for the default target (as set in the IDE in the Project menu)
- Connects to the target machine
- Loads the target program from file to the target machine
- Disconnects

This feature is a subset of the debugger launch process. Use **Load Default Target** primarily for loading programs into flash ROM to allow a speedier turn-around time for stand-alone testing.

If the default target is not a DSP56800 target, Code Warrior generates an error message and aborts the process. While loading, the software displays a progress bar.

You can abort the load by clicking the **Cancel** button on the progress bar. **Load Default Target** is not available while you are debugging a DSP56800 target.

**Load/Save Memory**

From the menu bar of the Metrowerks CodeWarrior window, select **DSP56800 > Load/Save memory** to display the **Load/Save Memory** dialog ([Figure 8.12](#)).
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 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.

**Radio Buttons**

The Load/Save Memory dialog box has two radio buttons for you to use:
• Load Memory
• Save Memory

The default is load.

**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**

Use this field to specify the address you want to write the memory to. If you want your entry to be interpreted as hex, prefix it with 0x; otherwise, it is interpreted as decimal.

**Size Text Field**

Use this field to 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 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. 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** reads 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 an OPENFILENAME or a SAVEFILENAME dialog, depending on whether you selected the **Load Memory** or **Save Memory** radio button.

**Fill Memory**

From the menu bar of the Metrowerks CodeWarrior window, select **DSP56800 > Fill memory** to display the **Fill Memory** dialog box (Figure 8.13).

![Fill Memory Dialog Box](image)

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 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.

**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**

Use this field to specify the address you want to write the memory to. If you want it to be interpreted as hex, prefix it with 0x; otherwise, it is interpreted as decimal.

**Size Text Field**

Use this field to specify the number of bytes 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 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 0x0BEEF on the target. Optionally, the string could have been set to 0xBE00EF 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 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 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. Clicking the Cancel button again closes the dialog. Pressing the Esc key is the same as pressing the Cancel button.

Default Information

The dialog is empty when you first display it. However, if default information is to appear, ship the text file containing default information. The dialog loads default information when it first appears.

Using DSP56800 Simulator

The CodeWarrior IDE for DSP56800E includes the Motorola DSP56800 Simulator. This software lets you run and debug code on a simulated DSP56800 architecture without installing any additional hardware.

The simulator does not simulate interrupts, it only simulates core instructions. In order to use the simulator, you must select it as your debugging protocol from the M56800 Target Settings panel (Target Settings Window).

NOTE: The simulator also enables the DSP56800 menu to retrieve the machine cycle count and machine instruction count when debugging.
Cycle/Instruction Count

From the menu bar of the Metrowerks CodeWarrior window, select DSP56800 > Cycle/Instruction count. The following window appears (Figure 8.14):

![Figure 8.14 Simulator Cycle/Instruction Count](image)

**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 8.15  Simulator Memory Map

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. OnCETM 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 Metrowerks CodeWarrior window, select **DSP56800 > Watchpoint status**.
   
   The **Watchpoint Status** window appears (Figure 8.16).

**NOTE** The **Watchpoint Status** menu item is disabled when you use the Simulator or during a system-level connect.
Figure 8.16 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 8.17 and listed in Table 8.3.
### Figure 8.17 Breakpoint Unit 1 Options

![Breakpoint Unit 1 Options](image)

### Table 8.3 Options for Breakpoint Unit 1

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus</td>
<td>Execute program fetch</td>
<td>When a P memory instruction is executed. <strong>Mode</strong> defaults to <strong>Read</strong>. Useful when only interest is opcode instructions.</td>
</tr>
<tr>
<td></td>
<td>Any P memory access</td>
<td>Any time a P memory address is accessed, depending on the value of <strong>Mode</strong>. Useful when writing or reading data from P memory.</td>
</tr>
<tr>
<td></td>
<td>X Address Bus 1</td>
<td>Access for all X address values through XAB1 (internal or external memory) depending on the <strong>Mode</strong> you select.</td>
</tr>
<tr>
<td>Value</td>
<td>C hexadecimal or decimal notation</td>
<td>Range: 0x0 to 0xFFFF</td>
</tr>
<tr>
<td>Mode</td>
<td>Read</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read and Write</td>
<td></td>
</tr>
</tbody>
</table>

**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 8.18 and listed in Table 8.4.

**Figure 8.18  Breakpoint Unit 2 Options**

![Breakpoint Unit 2 Options](image)

**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 8.4 Options for Breakpoint Unit 2

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserve Breakpoint Unit 2 for Debugger</td>
<td>Enabled</td>
<td>Breakpoint unit 2 cannot be user defined and the occurrence counter defaults to 1 for BPU1.</td>
</tr>
<tr>
<td></td>
<td>Disabled</td>
<td>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.</td>
</tr>
<tr>
<td>Bus</td>
<td>Core Global Data Bus (CGDB)</td>
<td>Data transfer between the data ALU and X data memory for one memory access.</td>
</tr>
<tr>
<td></td>
<td>Program Address Bus (PAB)</td>
<td>19-bit program memory address bus.</td>
</tr>
<tr>
<td>Value</td>
<td>The hexadecimal value read from the specified Bus.</td>
<td>To read full value, set <strong>Mask</strong> to 0xFFFF.</td>
</tr>
<tr>
<td>Mask</td>
<td>Mask value in C hex notation from 0x0 to 0xFFFF.</td>
<td>Specify a value of 0xFFFF for full value specified by <strong>Value</strong>. 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 <strong>Mask</strong> and <strong>Value</strong> fields.</td>
</tr>
</tbody>
</table>
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 nth iteration of a program loop or when the nth 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 8.19). The value you set in the Occurrence Counter text box determines the value of COUNTER.

**Figure 8.19** Sequence Counter Options in the Watchpoint Status Window

<table>
<thead>
<tr>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>☐ Breakpoint 1 occurs COUNTER times</td>
</tr>
<tr>
<td>☐ Breakpoint 1 or Breakpoint 2 occurs COUNTER times</td>
</tr>
<tr>
<td>☐ Breakpoint 1 and Breakpoint 2 simultaneously occur COUNTER times</td>
</tr>
<tr>
<td>☐ Breakpoint 2 occurs once, then Breakpoint 1 occurs COUNTER times</td>
</tr>
<tr>
<td>☐ Breakpoint 2 occurs COUNTER times, then Breakpoint 1 occurs once</td>
</tr>
</tbody>
</table>

Table 8.5 explains the options available in the Sequence group box.
### Table 8.5 Options for the Occurrence Counter

<table>
<thead>
<tr>
<th>Option</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakpoint 1 occurs COUNTER times</td>
<td>If <strong>Reserve Breakpoint Unit 2 for Debugger</strong> is enabled, this is the default options and COUNTER is 1.</td>
</tr>
<tr>
<td>Breakpoint 1 or Breakpoint 2 occurs</td>
<td>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.</td>
</tr>
<tr>
<td>COUNTER times</td>
<td>BPU1 and BPU2 work together. This is useful for monitoring bit status with a defined mask.</td>
</tr>
<tr>
<td>Breakpoint 2 occurs once, then Breakpoint</td>
<td>Useful for monitoring the status of recursive or nested algorithms.</td>
</tr>
<tr>
<td>1 occurs COUNTER times</td>
<td></td>
</tr>
<tr>
<td>Breakpoint 2 occurs COUNTER times, then</td>
<td>Useful for monitoring the status of recursive or nested algorithms.</td>
</tr>
<tr>
<td>Breakpoint 1 occurs once</td>
<td></td>
</tr>
</tbody>
</table>

### Setting and Clearing Watchpoint Status

You can set and clear 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.

### Change of Flow FIFO Dump

From the menu bar of the Metrowerks CodeWarrior window, select **DSP56800 > Change-of-flow FIFO Dump** to see the most recent changes
in the program flow and a reconstructed program trace (Figure 8.20).

Use this feature to query the FIFO History 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 Change-of-flow FIFO Dump menu item is enabled when the IDE is debugging a hardware target and debugging has stopped.

Figure 8.20 Change-of-Flow History Window

Status Line

The status line at the bottom of the window explains the state of the target as either running or stopped. This alerts the user to the validity of the data. The data is updated each time the target stops. The Target running status indicates that the current data was valid at the most recent debugger halt, but is not currently valid.

Data Display

The data display uses a tree control object. Initially, the tree is collapsed. The top most tree items are the addresses at which a flow change occurred, followed by the current Program Counter address. By double-clicking each top item, the item expands, showing the program flow between the selected address and the next change-of-
flow address. Each line displays the address and, if available, the disassembly and function name for that instruction. Double-clicking an expanded line collapses it.

**Trace Reconstruction**

There are situations when the trace cannot be reconstructed due to inconclusive evidence. These situations include occasional returns from subroutines and returns from interrupts. When an interrupt occurs immediately after a flow change, the trace cannot always be reconstructed, due to the fact that the interrupt is not always apparent. However, most of these problems are solved when symbolic information is available to the debugger.

The window is modeless and resides in the CodeWarrior IDE workspace. It can be resized, minimized, and maximized. It can be closed directly, by standard Windows methods, or indirectly, by killing the debugger.

**Global Variable Watchpoints**

You can monitor a single global variable as a watchpoint during a debugging session. This section explains how to set and clear a watchpoint.

**Setting a watchpoint**

1. Set a watchpoint:
   a. In the program window, right-click on a variable name.
      A menu appears in the program window.
   b. Select **Set/Clear Watchpoint** from the menu (Figure 8.21).
2. Set watchpoint mode:
   a. From the menu bar of the Metrowerks CodeWarrior window, select **Debug > Set/Clear Watchpoint**. The **DSP568 Watchpoint Mode Dialog** box appears (Figure 8.22):
b. Select Read, Write, or Read + Write from the DSP568 Watchpoint Mode dialog box (Figure 8.22).

c. Click OK.

You have now set a global watchpoint.

Clearing a Watchpoint

Use either of the following methods for clearing a watchpoint:

- Right-click the variable name in the program window during debugging. Select Set/Clear Watchpoint. The watchpoint is cleared.

- Select View > Watchpoints. Right-click the variable name set as a global watchpoint. The Watchpoints window appears (Figure 8.23). Select the variable in the Description column of the Watchpoints window. A menu appears. Select Clear Watchpoint. The watchpoint is cleared.

NOTE

Conditions are not enabled for watchpoints.
Register Details Window

From the menu bar of Metrowerks CodeWarrior window, select View > Register Details. The Register Details window (Figure 8.24) appears.

![Figure 8.24 Register Details Window](image)

You can use the Register Details window to view different DSP568xx registers. The most accurate and up-to-date listing of XML register descriptions files is in the following path:

\bin\Plugins\support\Registers\Dsp568\Generic

**NOTE**

The Register Details window can show both all-purpose DSP56800 registers as well as memory-mapped registers. You can create your own register views with XML files.

In the Register Details window, type the name of the register you want to view in the Description File field to display the applicable register and its values.

By default, the IDE looks in the following path when searching for a register description file:
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.

### Viewing Memory

#### Viewing X: Memory

You can view X memory space values as hexadecimal values with ASCII equivalents. You can edit these values at debug time.

On targets that have Flash ROM, you cannot edit those values in the memory window that reside in Flash memory.

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

   The **Memory** window appears ([Figure 8.25](#)).
2. Locate the **Page** list box at the bottom of the **View Memory** window. Select **X Memory**.

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

   To enter a hexadecimal address use standard C hex notation, for example, 0x100.

**NOTE** 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.

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 DATA section in the EVM board’s memory. The memory address location for DATA (and CODE) are set in the **Memory Segment** and **Sections Segment** 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.
**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. To view the memory address range of a variable, select `Data > View Memory` from the menu bar of the Metrowerks CodeWarrior window.

   The **Memory** window appears (Figure 8.25).

2. In the `Page` list box located at the bottom of the **View Memory** window, select `P Memory`.

3. Enter the memory address in the `Display` field.

   To enter a hexadecimal address use standard C hex notation, for example: `0x1000`.

4. Using the View list box, you have the option to view four types of P:Memory:
   - **Raw Data** (Figure 8.26).

![Figure 8.26 View P:Memory (Raw Data) Window](image-url)
- Disassembly (Figure 8.27).

Figure 8.27 View P:Memory (Disassembly) Window

- Source (Figure 8.28).

Figure 8.28 View P:Memory (Source) Window
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, select the Custom JTAG checkbox in the M56800 Target Settings panel.
5. Specify the name of the JTAG initialization file in the JTAG Init. File text field.
6. Specify the index of the core to debug in the JTAG Target Core Index text field (the index numbering sequence starts with 0).

7. 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 the chip you are debugging. For example, *Listing 8.1* shows a JTAG initialization file for three 56800 chips in a JTAG chain.

**Listing 8.1 Example JTAG Initialization File for DSP56800 Boards**

```plaintext
# 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
```

**NOTE** See the sample configuration file in the DSP EABI Support\JTAG folder.

**Debugging a Loaded Target**

If you have already downloaded an application file to hardware, you can kill the debug process and start over without having to reset the hardware and download your application again.
To debug a loaded target:

1. Disable the **Always Reset on Download** and **Always Load Program at Debugger Launch** options in the **M56800 Target Settings** panel.

2. Form the menu bar of the Metrowerks CodeWarrior window, select **Project > Debug**.

Your application begins at the entry point specified in the entry point in the **M56800 Linker Settings** panel without resetting the hardware.

---

**System-Level Connect**

The CodeWarrior IDE 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.
ELF Linker and Command Language

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 File Syntax
- Linker Command File Keyword Listing
- Sample M56800 Linker Command File

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 9.1 shows a sample memory-segment format.

```markdown
# Listing 9.1  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, read, write, and execute 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 in P memory while RW attributes are placed into X 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 \texttt{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 9.2.

Listing 9.2  Sample Symbol-level Closure Block

```text
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 9.3 is an example.

Listing 9.3  Sample Section-level Closure Block

```text
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 9.4 is an example.

Listing 9.4  Sample Section-level Closure Block With File Dependency

```text
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 10.2.

Listing 9.5 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:
  {               #end of .next_section_name definition
    more content descriptions
  } > segment_x   # 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 byte-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 boundary. A sample is in Listing 9.6.
You can also align data on a specific byte-boundary with \texttt{ALIGNALL} as shown in (Listing 10.7).

### Listing 9.7 Sample ALIGNALL Command Usage

```
file.c (.text)
ALIGNALL (0x10);  #everything past this point aligned on 16 bytes
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 9.1 shows the order of precedence for each operator. All operators are left-associative. To learn more about C operators, refer to the C Compiler Reference.

### Table 9.1 Arithmetic Operators

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (highest)</td>
<td>-  ~  !</td>
</tr>
<tr>
<td>2</td>
<td>*  /  %</td>
</tr>
<tr>
<td>3</td>
<td>+  -</td>
</tr>
<tr>
<td>4</td>
<td>&gt;&gt;  &lt;&lt;</td>
</tr>
<tr>
<td>5</td>
<td>==  !=  &gt;  &lt;  &lt;=  &gt;=</td>
</tr>
<tr>
<td>6</td>
<td>&amp;</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
Comments

Add comments by using the pound character (#) or C++ style double-slashes (//). C-style comments are not accepted by the LCF parser. Listing 9.8 shows examples of valid comments.

Listing 9.8 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 9.9 shows examples of valid lines for a command file:

Listing 9.9 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 9.10 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 9.10 Using a Global Variable in the LCF

F_foot = .;

If you use a global symbol in an LCF, as in Listing 9.10, it can be accessed from C program sources as shown in Listing 9.11.

Listing 9.11 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 9.12.

Listing 9.12 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 9.13 shows valid octal patterns you could put into your linker command file.

**Listing 9.13 Sample Octal Patterns**

```c
_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 10.14 shows examples of valid decimal integers that you could write into your linker command file.

**Listing 9.14 Sample Decimal Integers**

```c
_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 9.15.
Listing 9.15  Example Hexadecimal Integers

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>_somenumber</td>
<td>0x0F21</td>
</tr>
<tr>
<td>_fudgefactorspace</td>
<td>0XF00D</td>
</tr>
<tr>
<td>_hexonyou</td>
<td>0xcafe</td>
</tr>
</tbody>
</table>

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, as shown in Listing 10.16.

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 addition of the .text sections from those files again.

Sometimes you may only want to include the files from a particular file group. The GROUP keyword allows you to specify all the files of a named file group.

Listing 9.16  Sample file listing

```plaintext
SECTIONS {
    .example_section :
    {
        main.c  (.text)
        file2.c (.text)
        file3.c (.text)
        *   (.text)
        GROUP(fileGroup1) (.text)
        GROUP(fileGroup1) (.data)
    } > MYSEGMENT
}
```

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 10.17.

**Listing 9.17**  
Sample Function Selection Using the Object Keyword

```c
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 10.18 shows 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 9.18 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)
    *(fp_state.data);
    *(rtlib.data);
    F_End_Data = .;             # Get end location for RAM

    # .bss sections
    *(rtlib.bss.lo)
  }

TARGETING DSP56800
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 9.19, all variables in the data section from ROM to RAM in C code are copied.

**Listing 9.19**  
**ROM to RAM Copy From C After Data-Flash Write**

```c
#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. Listing 9.20 shows a code fragment from a section definition that illustrates this arithmetic.

You do not need to set up your heap and stack in the linker, as this is done by the run-time initialization of the MSL.

Listing 9.20 Setting Up Some Heap

```c
HEAP_ADDR = .;
HEAP_SIZE = 0x2000;  // this is the size of the heap
HEAP_END = HEAP_ADDR + heap_size;
. = HEAP_END         // reserve the space
```

The same thing is done for the stack using the ending address of the heap as the start of the stack (Listing 9.21).

Listing 9.21 Setting Up the Stack

```c
_stack_size = 0x2000; // this is the size of the stack
_stack_addr = heap_end + _stack_size;
. = _stack_addr;
```

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 WRITEx command writes four bytes. You insert the data at the section’s current address.
Listing 9.22  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 IDE for DSP56800E applications with the linker command file. Valid linker command file functions, keywords, directives, and commands are described:

- . (location counter)
- ADDR
- ALIGN
- ALIGNALL
- FORCE_ACTIVE
- GROUP
- INCLUDE
- KEEP SECTION
- MEMORY
- OBJECT
- REFINCLUDE
- SECTIONS
- SIZEOF
- SIZEOFW
- WRITEB
- WRITEH
- WRITES
- WRITEW
. (location counter)

Definition: 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 cannot 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 bytes past the symbol FSTART_.

Example:

```plaintext
=data :
{
   *(.data)
   *(.bss)
   FSTART_ = .;
   . = FSTART_ + 0x1000;
   __end = .;
}
> DATA
```

ADDR

Definition: 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:

```plaintext
MEMORY{
   ROOT (RWX) : ORIGIN = 0x8000, LENGTH = 0
}
```

SECTIONS{
.code :
{
  _rootbasecode = ADDR(ROOT);
  *(.text);
} > ROOT

ALIGN

Definition  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

Definition  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

Definition  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] }
KEEP_SECTION

Definition  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

Definition  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

Definition  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 either the GROUP keyword or the '*' wildcard selector.

REF_INCLUDE

Definition  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

Definition  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
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:

- `> 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

**Definition**  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

**Definition**  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

**Definition**  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

**Definition**  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

Definition 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 C string (must be within parentheses).

TIME returns the current time as 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

Definition 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 9.23. This is the typical linker command file.

Listing 9.23 Sample Linker Command File (DSP56824 EVM)

# ----------------------------------------------------
# Metrowerks, a company of Motorola
# sample code

# linker command file for DSP56824EVM
# using
#   external pRAM
#   external xRAM
#   internal xRAM (for compiler regs)
#     mode 3
#     EXT 0

# revision history
# 011020 R4.1 A.H. first version

# ----------------------------------------------------

# see end of file for additional notes
# additional reference: Motorola docs
#   DSP56F801-7UM.pdf
#   DSP56824EVMUM.pdf

# 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
# this is required for hostIO

# DSP56824EVM eval board settings:
# ON --> jumper JG1 pins 1-2 & 3-4 (enable mode 3 upon exit
#                                      from reset)
#
# due to above jumpers, we stay in mode 3 all time for 824

# CodeWarrior debugger Target option settings
# OFF --> "Use Hardware Breakpoints"

# if using DSP56824EVM, JG1 overrides this next option
# 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

# about the reserved sections
# for this external RAM only LCF:

# p_isr -- 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 -- reserved in internal xRAM 1
# The compiler uses page 0 address locations 0x30-0x40
# as register variables. See the Target manual for more info.

# 56824
# mode 3 (development)
# EX = 0

MEMORY {
  .p_isr_ext_RAM    (RWX) : ORIGIN = 0x0000, LENGTH = 0x0080
  .p_external_RAM   (RWX) : ORIGIN = 0x0080, LENGTH = 0x0000
  .x_comp_regs_iRAM (RW)  : ORIGIN = 0x0030, LENGTH = 0x0010
  .x_internal_RAM_1 (RW)  : ORIGIN = 0x0040, LENGTH = 0x07C0
  .x_internal_ROM   (R)   : ORIGIN = 0x0800, LENGTH = 0x0010
  .x_internal_RAM_2 (RW)  : ORIGIN = 0x0040, LENGTH = 0x0010
  .x_reserved       (R)   : ORIGIN = 0x1600, LENGTH = 0x0A00
}
ELF Linker and Command Language
Sample M56800 Linker Command File

.x_external_RAM   (RW)  : ORIGIN = 0x2000, LENGTH = 0xDF80
.x_on_chip_peri_1 (RW)  : ORIGIN = 0xFF80, LENGTH = 0x0040
.x_on_chip_peri_2 (RW)  : ORIGIN = 0xFFC0, LENGTH = 0x0040

} # we ensure the interrupt vector section is not deadstripped here

KEEP_SECTION{ isrVector.text }

# place all executing code & data in external memory

SECTIONS {
.interrupt_vectors_for_p_ram :
{
  * (isrVector.text)  # from 56824_vector.asm
}
> .p_isr_ext_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)
# setup the heap address
. = ALIGN(4);
_HEAP_ADDR = .;
.HEAP_SIZE = 0x00FF;
.HEAP_END = _HEAP_ADDR + _HEAP_SIZE;
. = _HEAP_END;

# setup the stack address
_min_stack_size = 0x1000;
_stack_addr = _HEAP_END;
_stack_end = _stack_addr + _min_stack_size;
. = _stack_end;

# export heap and stack runtime to libraries
F_heap_addr = _HEAP_ADDR;
F_heap_end = _HEAP_END;
F_stack_addr = _HEAP_END;

} > .x_external_RAM

# additional notes:
# about the reserved sections
# for this external RAM only LCF:

# p_isr -- 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 -- 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
This chapter covers features of the CodeWarrior™ debugger and ELF linker command file that allow you to program Flash ROM on DSP56800 series devices that have this capability. The CodeWarrior debugger for DSP56800 has Flash drivers for both program (P) and data (X) memory. When you use the debugger with a flash configuration file and linker command file, the debugger writes your code and/or data to Flash ROM.

This chapter contains the following sections:

- Setting up the Debugger for Flash Programming
- Setting up the Linker Command File for Flash Programming
- Preparing the Hardware for Flash Programming
- Flash Programming Tips
- Flash Programming the Reset and Interrupt Vectors

**NOTE** This chapter assumes that you are familiar with the CodeWarrior debugger and the linker command file format for DSP56800. Familiarize yourself with these topics before programming Flash ROM on DSP56800 devices.

### Setting up the Debugger for Flash Programming

In order for the debugger to download into Flash, the following options are required in the M56800 Target Settings panel:

- **Use Flash Config File**
  This option must be enabled.

- **Always load program at** debugger launch
  This option must be enabled.
• Debugger sets OMR on launch

This option must be disabled. The debugger does not set OMR at all when this option is disabled.

Figure 10.1 shows the M56800 Target Settings panel when you use minimum requirements for Flash programming.

**Figure 10.1** M56800 Target Settings Panel for Programming Flash

---

**Setting up the Linker Command File for Flash Programming**

To write to Flash you must properly set up your linker command file. The CodeWarrior debugger takes the link information from the MEMORY segments to know whether the intended memory segment in intended for ROM or RAM. If a MEMORY segment resides in ROM and the appropriate debugger settings are used, the debugger
performs a mass erase of the data and/or code ROM region before writing to ROM.

**NOTE**

Mass erases of Flash ROM regions for X: memory and P: memory are performed individually. For example, if you only want to flash P: memory and not X: memory, the debugger does not perform a mass erase of X: memory. This is determined by the setup in the linker command file.

---

**Specifying P: Memory**

To specify P: (program) memory you must specify your MEMORY segment attributes as (RWX). For example:

```
.text (RWX) : ORIGIN = 0x1000, LENGTH = 0x0
```

specifies that the `.text` memory segment is intended for P: memory and starts at 0x1000 in memory. See the section “Memory Segment” for more information.

**Specifying X: Memory**

To specify X: (data) memory you must specify your MEMORY segment attributes as (RW) or (W). For example:

```
.data (RW) : ORIGIN = 0x1000, LENGTH = 0x0
```

specifies that the `.data` memory segment is intended for X: memory and starts at 0x1000 in memory. See the section “Memory Segment” for more information.

---

**Preparing the Hardware for Flash Programming**

For the CodeWarrior debugger to write to Flash, you must have your hardware jumpered to mode 0. All EVM provide jumpers to do this. Refer to hardware manuals for EVM boards.
Flash Programming Tips

If you are programming Flash:

- 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.

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.
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
- Runtime Initialization

**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.

**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 for when you need minimal `printf` support for debugging and to save 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 11.1.
Libraries and Runtime Code

MSL for DSP56800

Table 11.1  Host File Creation Location

<table>
<thead>
<tr>
<th>fopen filename parameter</th>
<th>host creation location</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename with no path</td>
<td>target project file folder</td>
</tr>
<tr>
<td>full path</td>
<td>location of full path</td>
</tr>
</tbody>
</table>

**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 11.2.

Table 11.2  Host File Creation Location

<table>
<thead>
<tr>
<th>file opened as</th>
<th>host elements</th>
<th>target elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>8-bit</td>
<td>16-bit</td>
</tr>
<tr>
<td>binary</td>
<td>16-bit</td>
<td>16-bit</td>
</tr>
</tbody>
</table>

**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 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 and Command Language,” 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 M56824 init (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 11.3 for the variables defined in each Stationery LCF.

Table 11.3 LCF Variables and Address

<table>
<thead>
<tr>
<th>Variables</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>_stack_addr</td>
<td>The start address of the stack</td>
</tr>
<tr>
<td>_heap_size</td>
<td>The size of the heap</td>
</tr>
<tr>
<td>_heap_addr</td>
<td>The start address of the heap</td>
</tr>
<tr>
<td>_heap_end</td>
<td>The end address of the heap</td>
</tr>
<tr>
<td>_bss_start</td>
<td>Start address of memory reserved for</td>
</tr>
<tr>
<td></td>
<td>uninitialized variables</td>
</tr>
<tr>
<td>_bss_end</td>
<td>End address of BSS</td>
</tr>
</tbody>
</table>

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 Settings panel.

When creating a project from R5.0 stationery, the init code is specific to the DSP56800 board. See the startup folder in the new project folder for the init code.

Listing 11.1  Sample Initialization File (DSP56824EVM)

```c
/*
56824_init.c
sample code
Metrowerks, Inc., a company of Motorola
*/

#include "56824_init.h"

asm void init_M56824_()
{
    bfset __32bit_comparcs,omr // debugger will override this
        // if debugger option is on
    move #-1,x0
    move x0,m01          // set the m register 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:TCR01
move r1,x:TCR2
move r1,x:SCR2
move r1,x:SPCR0
move r1,x:SPCR1
move r1,x:COPCTL

// copy interrupt table to address 0
move #$80,r2 // internal interrupt size 0x80
move #M56824_intVec,r3 // address originally loaded
move #0,r1 // destination address
do r2,enddoA
move p:(r3)+,x0
move x0,p:(r1)+
enddoA:

// 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)

// PLL (phase-locked loop) init for 824
move #$0180,X:PCR1 // configure
move #$0260,X:PCR0 // set Feedback Divider to 1/20

// wait for PLL lock
move #$FFFF,y0 // an amount in keeping with data sheet
move y0,lc
Libraries and Runtime Code
Runtime Initialization

```assembly
  do   lc,delay_for_pll
  nop
delay_for_pll:
       // that should be enough time
       // for PLL stabilization
  bfset #$4000,X:PCR1    // now enable PLL for Phi Clock

// setup exception handler and interrupt levels

  move M56824_int_Addr,r1  // exception handler address
  pushr 1                  // establish exception handler
  bfset   #$0100, sr        // enable all levels of interrupts
  bfclr  #$0200, sr         // allow IPL 0 interrupts

// call main()

  move #M56824_argc,y0     // pass parameters to main()
  move #M56824_argv,r2
  move #M56824_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.
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:

- **The Debugger Crashes or Freezes When Stepping Through a REP Statement**
- **“Can’t Locate Program Entry On Start” or “Fstart.c Undefined”**
- **When Opening a Recent Project, the CodeWarrior IDE Asks If My Target Needs To Be Rebuilt**
- **“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”**
- **IDE Closes Immediately After Opening**
- **Errors When Assigning Physical Addresses With The `org` Directive**
- **The Debugger Reports a Plug-in Error**
- **Windows Reports a Failed Service Startup**
- **No Communication With The Target Board**
- **Downloading Code to DSP Hardware Fails**
- **The CodeWarrior IDE Crashes When Running My Code**
- **The Debugger Acts Strangely**
- **Problems With Notebook Computers**

If you are having trouble with CodeWarrior IDE for DSP56800E and this section does not help you, e-mail technical support at: support@metrowerks.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 Metrowerks 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 Metrowerks CodeWarrior window, the dialog box shown in Figure 12.1 appears:
Troubleshooting
Troubleshooting Tips

Figure 12.1 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 appropriate stationery.

The stationery is located in the following directory:

CodeWarrior\Stationery\DSP56800 EABI

Locate the directory for the DSP568xxEVM processor you are using. The flash configuration file is located in the config directory.

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
Troubleshooting
Troubleshooting Tips

Metrowerks/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 12.2** appears:

![Figure 12.2 Service Control Manager Message Box](image)

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.
Troubleshooting Tips

Targeting DSP56800

- Check input/output addresses according to the operating system you are using:
  - Windows 95 and 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".

- If you are using the Motorola 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.
Troubleshooting Tips

**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.
Porting Issues

This chapter explains issues relating to successfully porting code to the most current version of the CodeWarrior™ IDE for Motorola DSP56800. This chapter lists issues related to successfully porting sources from the Suite56™ toolset and differences that occur between the CodeWarrior IDE and the Suite56 tools.

This chapter contains the following sections:

- Converting the DSP56800 3.x and 4.x Projects to 5.x Projects
- Porting DSP56811 to DSP56824 Projects
- Using XDEF and XREF Directives
- Using the ORG Directive

Converting the DSP56800 3.x and 4.x Projects to 5.x Projects

When you open older projects in the CodeWarrior IDE, the IDE automatically prompts you to convert your existing project (Figure 12.3). 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.
Porting Motorola 56824EVM projects to the CodeWarrior IDE

Porting projects for DSP56800 processors created with the Motorola DSP56800 Suite56 toolset to the CodeWarrior assembler is based on the Motorola syntax. Virtually all your code is usable with little modification. However, porting Motorola applications to the CodeWarrior assembler require some manual intervention. To port a CodeWarrior IDE project from the Motorola Suite56:

1. Create a new project from the project stationary in the CodeWarrior IDE for your target board.
2. Convert Motorola bootstrap assembly source and interrupt tables to the CodeWarrior IDE.
3. Add your new files to the new CodeWarrior project.
4. Call your assembly startup function from `main()`.
5. Modify your linker command file.

A sample assembly source program is given below that was ported from the DSP56824 Suite56 tools to the CodeWarrior IDE that has detail information.

A simple one file assembly program written with the Motorola DSP56824EVM toolset is shown in listing Listing 12.1 This simple program blinks the PB8 LED on the 56824EVM board at a rate of 1 Hz. The same program, shown in Listing 12.2, was ported to the CodeWarrior IDE.
Listing 12.1  Motorola Suite56 Assembly Sources

; Program Name: SimpleLed.asm

; Equates for DSP56824 core
;=================================================================================
ipr equ $fffb ; Interrupt priority register
bcr equ $fff9 ; Bus control register
pcr1 equ $fff3 ; PLL control register 1
pcr0 equ $fff2 ; PLL control register 0
pbd equ $ffec ; Port B data register
pbddr equ $ffeb ; Port B data direction register
pbint equ $ffea ; Port B Interrupt register
;=================================================================================
; PARAMETERS
;=================================================================================
PLL_MUL equ 19 ; PLL Feedback Multiplier
Red equ $0100 ; Port B bit 8 is Red Led
;=================================================================================
; Memory Space Setup
;=================================================================================
org p:$e000 ; warm boot
jmp Start

org p:$0000 ; Location p:$0 to p:$ff are set to nop
dup 256
nop
endm

org p:$0000 ; start of program
jmp Start

;=================================================================================
; MAIN
;=================================================================================
org p:$0100 ; Starting location of this program
Start

move #$40,sp ; Set stack pointer to first location after page 0
Porting Issues
Porting Motorola 56824EVM projects to the CodeWarrior IDE

move   #$0000,x:bcr ;Initialize BCR for zero wait states
       ; Configure PLL feedback divider
move   #(PLL_MUL-1)<<5,x:pcr0 ;3.6864 MHz * 19 = 70.042MHz
       ; Enable PLL using oscillator clock 4ac8
move   #$4208,x:pcrl  ; Enable P
move   #$1fff,lc
move   #$0000,x:pbd   ; Led off
bfset  #$0700,x:pbddr ; Port b 8,9,10 output for Red,
       ; Yellow and Green Leds
move   #$0800,x:ipr   ; Enable Timer interrupts
bfset  #$0100,sr      ; Enable all level of interrupts
bfclr  #$0200,sr

;=============================
; this will make the Red Led blink using normal delays
Blink
   ;
   jsr   Wait
   jsr   Wait
   bfchg  #Red,x:pbd  ; blink Red Led
   jmp   Blink

;================================
; Wait
Wait
   move   #$1fff,a0
   move   #$03ff,y0
Up1  move   #$ffff,x0
Up2  nop
Up2  rep   a0
Up2  nop
Up2  decw  x0
Up2  bgt   Up2
Up2  nop
Up2  decw  y0
Up2  bgt   Up1
Up2  nop
   rts
The following assembly source (Listing 12.2) was ported to the CodeWarrior IDE to be called from C. Note that the only difference between these files is in the memory space setup. The CodeWarrior IDE does not allow you to specify memory space and location counter addresses with the ORG directive. Memory space setup must be done within a linker command file.

### Listing 12.2 Converted Assembly Sources for the CodeWarrior IDE

```assembly
; Program Name: SimpleLed.asm
SECTION user
; Equates for DSP56824 core
;==============================================================================
ipr equ $fffb           ; Interrupt priority register
bcr equ $fff9           ; Bus control register
pcr1 equ $fff3           ; PLL control register 1
pcr0 equ $fff2           ; PLL control register 0
pbd equ $ffec           ; Port B data register
pbddr equ $ffe6b         ; Port B data direction register
pbint equ $ffea          ; Port B Interrupt register
;==============================================================================
;==============================================================================
;                       PARAMETERS
;==============================================================================
PLL_MUL equ 19            ; PLL Feedback Multiplier
Red equ $0100            ; Port B bit 8 is Red Led
;==============================================================================
;==============================================================================
;                       Memory Space Setup
;==============================================================================
org p:
GLOBAL FStart
NewInstx0
jmp FStart
;==============================================================================
FStart
move #$40,sp           ; Set stack pointer to first location after page 0
move #$0000,x:bcr     ; Initialize BCR for zero wait states
```
Porting Issues
Porting Motorola 56824EVM projects to the CodeWarrior IDE

; Configure PLL feedback divider
move  #(PLL_MUL-1)<<5,x:pcr0 ; 3.6864 MHz * 19 = 70.042MHz
; Enable PLL using oscillator clock 4ac8
move  #$4208,x:pcr1       ; Enable PLL
move  #$1fff,lc

move  #$0000,x:pbd        ; Led off
bfset  #$0700,x:pbddr    ; Port b 8,9,10 output for Red,
                        ; Yellow and Green Leds
move  #$0800,x:ipr       ; Enable Timer interrupts
bfset  #$0100,sr         ; Enable all level of interrupts
bfclr  #$0200,sr

;==============================================================
; this will make the Red Led blink using normal delays
Blink
    jsr     Wait
    jsr     Wait
    bfchg     #Red,x:pbd    ; blink Red Led
    jmp     Blink

;==============================================================
; Wait
Wait
    move  #$1fff,a0
    move  #$03ff,y0
Up1
    move  #$ffff,x0
Up2
    nop
    rep    a0
    nop
    decw  x0
    bgt    Up2

    nop
    decw  y0
    bgt    Up1
    nop
    rts
ENDSEC
END
Now, create a new target project, add the sources, and modify your main function. Your main function looks like the one in Listing 12.3 and retrieves your assembly program’s F_START function (Listing 12.2). The default linker command file in the project stationery is used for this example.

**Listing 12.3 Calling an Assembly Program from main**

```c
int main( void ){
    Start();
    return 0;
}
```

**Porting DSP56811 to DSP56824 Projects**

Refer to the following document for issues relating to porting DSP56811 designs to DSP56824 designs:

*Converting DSP56L811-Based Designs to the DSP56824. October 1, 1998. Motorola, Inc.*

**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:

```asm
ORG P:$0020
SECTION myISR_20
rti
ENDSEC
SECTION myISR_30
```
Porting Issues
Using the ORG Directive

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
```
Index

A
about, CodeWarrior IDE 9
__abs 157
Access Paths panel 78
access permission flags 224, 240
__add 159
Add Files command 59
adding assembly language 149
addr 237
ADS Base Address pop-up menu 183
ADS Command Converter protocol 182
after 241
align 238
align all 238
alignment 226
Allocating Memory and Heaps for DSP56800 258
Allow DO Instructions option 108
Allow Rep Instructions checkbox 108
Always load program at debugger launch checkbox 127
Application option, of Project Type pop-up menu 92
asm keyword 148
assembler, stand-alone 42
assembly language 147
create output option 108
statements, adding 149
AT keyword for ROM location 232
Auto-clear previous breakpoint on new breakpoint release 128

B
back-end compiler See compiler
bool size 130
bootstrap code 260
breakpoints 64
Bring Up To Date command 44
Build Extras panel 82
build system, depicted 45
build targets
setting in project 54
build targets, defined 10

C
C/C++ Language panel 93
C/C++ Warnings panel 98
calling assembly functions from C code 151
calling conventions for DSP 133
Case Insensitive Identifiers checkbox 102
changing 145
changing PCI connection, command converter server 195
char size 130
code
compiling 58
deadstripping unused 145
editing 59
navigation 63
code and data storage for DSP 134
CodeWarrior
collection to command line 43
compiler architecture 45–46
compiler, described 42
components 41
debugger, described 42
debugging for DSP 181
development process 43–46
getting started 13
IDE, about 9
IDE, described 41
installing 16
introduction 9
linker, described 42
stand-alone assembler described 42
target settings 71
tools, listed 41
troubleshooting 263
tutorial 47, 47–69
using the debugger 59
using the IDE 47
CodeWarrior IDE
about 9
available tools 10
documentation, described 9
introduction 9
Command Converter Server 195
changing PCI connection 195
Index

changing the parallel port 193
target settings 192
command converter server
icon 193
command line and CodeWarrior compared 43
commands
Add Files 59
Bring Up To Date 44
Compile 44
Enable Debugger 46
M56800 Settings 60
Make 46
Preprocess 46
Run 68
commands for linker command file 228
communications with target board, problems 267
Compile command 44
compiler
architecture 45–46
back-end for DSP 129
build system, depicted 45
described 42
intermediate representation (IR) 45
plug-in modules, explained 46
support for inline assembly 147
See also C Compilers Reference
compiling 44
code 58
See also IDE User Guide
compress 241
Connect menu item 222
Constant Data Section 143
converting CodeWarrior projects 271
converting Suite56 to CodeWarrior
M56824EVM projects 272–277
core tools, tutorial 47–69
Create Assembly Output checkbox 108
creating labels for DSP56800E Assembly 150
Custom Keywords settings panel 116

data, deadstripping unused 145
deadstripping
prevention 224, 228
deadstripping unused code and data 145
debug information, generating 57
debugger
described 42
DSP56800 menu 196
Kill command 69
M56800 Target Settings 182
preferences 181
problems with behavior 268
setting preferences 60
setting up for Flash programming 251
toolbar 63
using 59
See IDE User Guide
Debugging
Register Disassembly Window 215–216
Remote Target Board 195
debugging 46
connecting to a loaded target 220, 221
per file 57
projects 62
target settings 181
watchpoint status 203–214
without symbolics 222
See also IDE User Guide
Debugging a loaded target 220, 221
defining an inline assembly function 149
definition
BSS 258
heap 258
stack 258
development tools 41
Directive
XDEF 277
Disable Deadstripping checkbox 114
Disassemble Exception Tables checkbox 106
__div 166
__div_ls 167
DO instructions, allowing 108
Domain Technologies SB-56K
installing 22, 26
double size 131
downloading code, problems 268
DSP
code and data storage 134
installing hardware 19
linker 145
DSP hardware
system requirements 14
DSP56800
calling conventions 132
fixed-point formats 131
floating-point formats 131
integer formats 130
stack frame 133
DSP56800 menu 196
DSP56800 Simulator, using 201
DWARF Info checkbox 112

E
editing
  code 44
  project contents 59
  source files 59
  See also IDE User Guide
deritor, of IDE 59
Ethernet Command Converter protocol 183
ELF Disassembler settings panel 103–105
  Disassemble Exception Tables checkbox 106
  Show Addresses and Object Code checkbox 105
  Show Code Modules checkbox 105
  Show Comments checkbox 106
  Show Data Modules checkbox 106
  Show Debug Info checkbox 106
  Show Headers checkbox 104
  Show Relocations checkbox 105
  Show Source Code checkbox 105
  Show Symbol Table checkbox 104
  Use Extended Mnemonics checkbox 105
  Verbose Info checkbox 104
Enable Debugger command 46
  enabling the debugger 57
Entry Point field 116
  FSTART 116
expressions, in LCF 230
  _extract_h 165
  _extract_l 165

F
F 229
failed service startup in Windows 266
Faster Execution Speed option, Global Optimizations panel 110
File Mappings panel 86
fixed type 131
  fixed__ 131
  __fixed2long 161
  __fixed2short 162
fixed-point formats, for DSP 56800 131
  fixed 131
  long fixed 131
  short fixed 131
fixed-point formats, for DSP 56800 short fixed 131
flash configuration file format 127
Flash ROM
  debugger configuration 251
  initializing variables in P or X memory 232
  programming tips 251–254
  ROM to RAM copy 232–235
float size 131
floating-point formats, for DSP 56800 131
Force Active Symbols edit box 116
force_active 225, 228, 239
format, flash configuration file 127
fractional arithmetic 155
equation for converting 155
FSTART 116
troubleshooting entry point 264
fstart 260

G
Generate ELF Symbol Table checkbox 114
Generate Link Map checkbox 112
Generate Listing File checkbox 102
Generate Symbolic Info checkbox 112
generating debug info 57
GLOBAL directive 277
GLOBAL directive, assembly function definitions 152
Global Optimizations 109–110
  faster execution speed option 110
  optimizing for space or speed 110
  smaller code size option 111
global variable watchpoints 212–214
global variables
  linker command file 229
group 231, 239

H
hardware breakpoints
  enabling in debugger settings 128
  watchpoints 203–214
index

heap size 235
host, defined 9

I
icon, command converter server 193
IDE
described 41
using 47
implied fractional value 155
include 239
inline assembler
for DSP 147–164
inline assembly
defining functions 149
function-level 148
instructions 148
statement-level 148
syntax 147
installing
CodeWarrior 16
DSP hardware 16
SB-56K Emulator 22, 26
Instruction Scheduling checkbox 107
int size 130
__int2fixed 162
integer formats, for DSP56800 130
integral types, in LCF 228
intrinsic functions
absolute/negate 157
__abs 157
__L_negate 158
__negate 158
addition/subtraction 158
__add 159
__L_add 160
__L_sub 160
__sub 159
control 160
__stop 161
conversion 161
__fixed2int 161
__fixed2long 161
__fixed2short 162
__int2fixed 162
__labs 162
__long2fixed 163
__short2fixed 163
__memcpy 163
__strcpy 164
deposit/extract 164
__extract_h 165
__extract_l 165
__L_deposit_h 165
__L_deposit_l 166
division 166
__div 166
__div_ls 167
multiplication/MAC 167
__L_mac 170
__L_msu 170
__L_mult 171
__mac_r 167
__msu_r 168
__mult 169
__mult_r 169
normalization 171
__norm_l 172
__norm_s 172
rounding 173
__round 173
shifting 173
__L_shl 175
__L_shr 176
__L_shr_r 177
__shl 174
__shr 174
__shr_r 175
multiplication/MAC
__L_mult_ls 171
introduction
to CodeWarrior 9
introduction to the CodeWarrior IDE 9
K
keep_section 225, 228, 240
Kill command 69
L
__L_add 160
__L_deposit_h 165
__L_deposit_l 166
__L_mac 170
__L_msu 170
__L_mult 171
Index

_L_mult_ls 171
_L_negate 158
_L_shl 175
_L_shr 176
_L_shr_r 177
labels, M56800 assembly 150
__labs 162
LCF. See linker command files
libraries
  MSL for DSP 255
  support for DSP 255
  using MSL 255
Library option, of Project Type pop-up menu 92
license key 17
linear addressing 260
link order 145
linker
described 42
for DSP 145
link order 145
  See also linker command files
settings 111–115
linker command files 223
access permission flags 224, 240
addr 237
after 241
align 238
alignall 238
alignment 226
arithmetic operations 227
comments 228
compress 241
deadstripping prevention 228
expressions 230
file selection 231
force_active 239
function selection 231
group 231, 239
heap size 235
include 239
integral types 228
keep_section 240
memory 224, 240–241
memory attributes for X and P
  memory 224
object 231, 242
ref_include 242
sections 226, 242
sizeof 244
stack size 235
symbols 228
variables 228
writeb 244
writen 244, 245
writew 245
writing data 235
Linker pop-up menu 76
linking 46
  See also IDE User Guide
List Unused Objects checkbox 113
local variables, displaying 68
long double size 131
long fixed type 132
long size 130
__long2fixed 163
longfixed__ 132
_L_sub 160
M
M01 260
M56800 Assembler settings panel 101–103
  Case Insensitive Identifiers checkbox 102
  Generate Listing File checkbox 102
  Prefix File 103
M56800 Linker
  Disable Deadstripping checkbox 114
  Force Active Symbols edit box 116
  Generate ELF Symbol Table checkbox 114
  Generate Symbolic Info checkbox 112
  List Unused Objects checkbox 113
  Show Transitive Closure checkbox 113–114
  Store Full Path Names checkbox 112
M56800 Linker option, in Linker pop-up menu 76
M56800 Linker settings panel 111–115
  and debugging settings panels 182
  Entry Point field 116
  Generate DWARF Info 112
  Generate Link Map checkbox 112
  Max Record Length field 115
  S-Record EOL Character pop-up menu 115
  Suppress Warning Messages checkbox 114
M56800 Processor settings panel 106–108
  Allow DO Instructions 108
  Allow Rep Instructions checkbox 108
and debugging settings panels 182
Create Assembly Output checkbox 108
Instruction Scheduling checkbox 107
Make Strings Read-Only checkbox 108
M56800 Settings command 60
M56800 Target Settings 56, 60, 182
Always load program at debugger launch option 127
Use Flash Config File option 125
Use hardware breakpoints option 128
M56800 Target Settings panel 123, 182
M56800 Target settings panel
Output File Name 92
Project Type 92
M56800 Target settings panels 91
__mac_r 167
Make command 46
Make Strings Read-Only checkbox 108
makefiles 43
__memcp y 163
memory 240–241
memory space setup
OR G directive 275
memory window
X memory 216
Metrowerks Standard Library (MSL)
for DSP 255
using 255
modifier register 260
modulo addressing 260
Motorola Documentation 11
__msu_r 168
__mult 169
__mult_r 169

O
number formats, for DSP 129, 132

O B J E C T 231
object 231, 242
optimizing
for DSP 135
page 0 register assignment 134
register coloring 134
space or speed 110
ORG 275
ORG directive 152
memory space location 152
Output Directory field 76

P
page 0 register assignment 134
non-volatile registers 134
volatile registers 134
Parallel Port - ADS or EVM protocol 182
Parallel Port pop-up menu 187, 188, 190, 191
PCI Command Converter protocol 182, 183
platform target, defined 9
plug-in error 266
pointer size 131
porting issues 271
Post-Linker option 76
Pragma Directives
pragma interrupt 139
pragma optimization 142
pragma optimization 142
Prefix File 103
Prefix File field 103
Pre-Linker pop-up menu 76
Preprocess command 46
preprocessing 46
See also IDE User Guide
project stationery 48, 51
Project Type pop-up menu 92
Project window 52
projects
build targets 10
debugging 62
defined 9
editing contents of 59
platform target 9
navigating code 63
__n e g a t e 158
New Project window 51
New window 49
None option
in Post-Linker pop-up menu 76
in Pre-Linker pop-up menu 76
non-volatile registers 134
__norm_1 172
__norm_s 172

N

DSP–284 Targeting DSP56800
Index

stationery 48, 51
targets, defined 9
Protocol pop-up menu 182
protocol, selecting
  CCS 183
  Ethernet 183
  Parallel Port - ADS or EVM 182
  PCI 182
  Serial-EVM 182
  Serial-SB-56K 182
  Simulator 182
  UCC 182
protocols, setting 61

R
rebuild alert 264
REF_INCLUDE 228
ref_include 225, 228, 242
references
  Motorola Documentation 11
register coloring optimization 134
register description files and XML 215
Register Disassembly Window 215–216
  Format drop down 216
  locating register description files 216
  Text View drop down 216
registers
  display contents 65, 66, 67, 216, 218
  function parameters 132
  special-purpose 65, 66, 67, 216, 218
  stack pointer 133
regservers.bat 265
rep instruction
  problems in debugger 264
REP instructions, allowing 108
requirements See system requirements 13
requirements, system 13
rodata section 143
ROM to RAM copy 232–235
  _round 173
Run command, debugger 68
runtime
  ROM to RAM copy 234
runtime initialization 260
Runtime Settings panel 84

S
Sample Initialization File 260
Save Project Entries Using Relative Paths checkbox 78
SB-56K Emulator, installing 22, 26
SECTION mapping, in assembly language 152
sections 226, 242
segment location specifier 243
Serial - EVM protocol 182
Serial - SB56K protocol 182
setting
  a build target 76
  breakpoints 64
  debugger preferences 60
settings panels
  Access Paths 78
  Build Extras 82
  C/C++ Language 93
  C/C++ Warnings 98
  Custom Keywords 116
  ELF Disassembler 103–105
  File Mappings 86
  Global Optimizations 109–110
  M56800 Assembler 101–103
  M56800 Linker 111–115, 182
  M56800 Processor 106–108, 182
  M56800 Target 91
  M56800 Target Settings 123, 182
  Runtime Settings 84
Settings window 54
__shl 174
short double size 131
short fixed type 132
short size 130
__short2fixed 163
Show Addresses and Object Code checkbox 105
Show Code Modules checkbox 105
Show Comments checkbox 106
Show Data Modules checkbox 105
Show Debug Info checkbox 106
Show Headers checkbox 104
Show Relocations checkbox 105
Show Source Code checkbox 105
Show Symbol Table checkbox 104
Show Transitive Table checkbox 113–114
__shr 174
Index

__shr_r 175
signed char size 130
simulator
  using 201
Simulator protocol 182
sizeof 244
Smaller Code Size option, Global Optimizations panel 111
source files
  editing 59
special-purpose registers 65, 66, 67, 216, 218
S-record 115
S-Record EOL Character pop-up menu 115
S-Record, Max Record Length field 115
stack frame, for DSP56800 133
stack pointer register 133
stack size 235
stand-alone assembler See Assembler Guide
stand-alone assembler, described 42
statement-level inline assembly 148
__stop 161
storage of code and data for DSP 134
Store Full Path Names checkbox 112
__strcpy 164
__sub 159
Suite56 toolset 271
support
  web page 46
Suppress Warning Messages checkbox 114
symbols, in LCF 228
syntax, inline assembly language 147
system requirements 13
  for DSP hardware 14
  Windows 13
system-level connect 222

T
Target Name field 75
Target Settings panel 75–78
  Linker 76
  Output Directory field 76
  Post-Linker 76
  Pre-Linker 76
  Save Project Entries Using Relative Paths checkbox 78
  Target Name 75

Target Settings panels
  Access Paths 78
  Build Extras 82
  C/C++ Language 93
  C/C++ Warnings 98
  Custom Keywords 116
  File Mappings 86
  Global Optimizations 109–110
  M56800 Linker 111–115
  M56800 Processor 106–108
  M56800 Target Settings 123
  M56800 Target settings 91
  Runtime Settings 84
Target Settings window 54
target settings, and debugging panels 181
target, defined 9
Troubleshooting
  Parallel Port Converter on Windows 2000 269
troubleshooting 263–268
  communications with target board 267
downloading code 268
  entry point errors 264
  FSTART 264
  ORG and memory addresses 266
  plug-in error 266
  rebuild alert 264
  rep instruction and breakpoints 264
tutorial, core tools 47–69

U
unsigned char size 130
unsigned int size 130
unsigned long size 130
unsigned short size 130
unused code and data, deadstripping 145
Use Extended Mnemonics checkbox 105
Use Flash Config File checkbox 125
Use hardware breakpoints checkbox 128
using
  DSP56800 Simulator 201
  the CodeWarrior debugger 59
  the CodeWarrior IDE 47
using comments in M56800E assembly 150

V
variables, displaying locals 68
variables, in LCF 228
Index

Variables, Stationery Linker Command Files  259
Verbose Info checkbox  104
viewing memory
  Register Disassembly Window  215-216
  X memory  216
volatile registers  134
  page 0 register assignment  134

W
watchpoint status  203-214
watchpoints
  clearing  214
  global variable watchpoints  212-214
web site  11
Windows
  failed service startup error  266
  system requirements  13
writeb  235, 244
writeh  235, 244, 245
writew  235, 245

X
X memory
  displaying  216
XDEF directive  277
xMAP  112
XML and register description files  215
XREF directive  277