CodeWarrior™ Development Studio for StarCore™ and SDMA Targeting Manual

Revised: 22 May 2007

How to Contact Us

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

Table of Contents

1

Introduction

This manual explains how to use the CodeWarrior Development Studio for software development on the StarCore and Smart DMA (SDMA) cores of the MXC05, i.300-30, MXC275-30, MXC300-10, and MXC300-30 chipsets. These system-on-a-chip (SoC) architectures have:

- An ARM® 11 processor
- A Smart Direct Memory Access (SDMA) controller
- A StarCore SC140V3 digital signal processor (DSP)
- Associated memory subsytems and peripherals

This chapter has these topics:

- Read the Release Notes
- Related Documentation
- Development Tools Overview

Read the Release Notes

Please read the release notes. They contain important information about new features, bug fixes, and incompatibilities that might not be in the documentation due to release deadlines.

The release notes are in

`CodeWarrior\Release_Notes`

where *`CodeWarrior`* is the directory in which you installed the CodeWarrior product.

Related Documentation

This manual explains high-level software development specific to the MXC05, i.300-30, MXC275-30, MXC300-10, and MXC300-30 chipsets. Table 1.1 lists additional CodeWarrior documentation.StarCore® and SDMA Targeting Manual

Table 1.1 Related Documentation

Document	Description
CodeWarrior™ Development Studio IDE User's Guide	Explains CodeWarrior software features that apply to all host platforms and platform targets. Documents all standard features of the IDE.
CodeWarrior™ Development Studio for StarCore™ DSP Architectures Targeting Manual	Explains StarCore-specific CodeWarrior software features, including target settings panels that are specific to the StarCore platform target.
SC140 DSP Core Reference Manual	Explains the instruction-set architecture and programming model for the SC140 core as well as corresponding register details and programming modes.
Targeting the Interprocessor Communication Module Core for the Rainbow Platform	Explains the Interprocessor Communication Module (IPCM), including target settings that are specific to the IPCM platform target.
Enterprise C Compiler User Guide	Explains the Enterprise C compiler. (This compiler produces object code that is compatible with the StarCore processor.)
StarCore and SDMA Build Tools Reference	Explains the assembler, linker, and Executable and Linking Format (ELF) utilities for SDMA.

Development Tools Overview

This section has an overview of the development tools specific to your architecture. These tools are part of the CodeWarrior software:

- Enterprise C Compiler
- Assembler
- Linker
- CodeWarrior Debugger
- StarCore Utilities

Enterprise C Compiler

The Enterprise C Compiler has these features:

- Conforms to version 1 of the StarCore™ Application Binary Interface (ABI) standard

- Conforms to ISO/IEC 9899:1999 (C99), except for complex-type support
- Supports a set of digital signal processor (DSP) extensions
- Supports International Telecommunications Union (ITU)/European Telecommunications Standards Institute (ETSI) primitives for saturating arithmetic. Additional parameters are available for non-saturating arithmetic and double-precision arithmetic.
- Allows standard C constructs for representing special addressing modes
- Supports a wide range of runtime libraries and runtime environments
- Optimizes for size (smaller object code), speed (faster object code), or a combination of both, depending on options that you select

The compiler can link all application modules before optimizing. By examining the entire linked application before optimizing, the compiler produces highly optimized object code. The compiler performs many optimizations, such as:

- Software pipelining
- Instruction paralleling and scheduling
- Data and address register allocation
- Aggressive loop transformations, including automatic unrolling

For more information about this tool, refer to the *Enterprise C Compiler User Guide*.

Assembler

The assembler translates assembly-language source code into machine-language object files or executable programs. You can provide the assembly-language source code, or the compiler can generate it.

For each assembly-language module in a build target, the SC140 assembler can generate a list file that shows the generated object code side-by-side with the assembly-language source code.

The CodeWarrior tools support the Target Assembler for the Smart DMA controller. You must purchase the Target Assembler separately from this CodeWarrior product.

Linker

The linker combines object files into a single executable file. You specify the link mappings of your program in a linker command file (LCF).

You can create an LCF by typing commands in a text file. Alternatively, you can use the the Link Commander utility. The Link Commander presents graphical representations of your memory segments and program sections that you can manipulate with the mouse to create an LCF.

The CodeWarrior tools support the Target Linker for the Smart DMA controller. You must purchase the Target Linker separately from this CodeWarrior product.

CodeWarrior Debugger

The CodeWarrior debugger lets you diagnose your software on both simulator and hardware targets.

Also, the CodeWarrior tools include a multi-core debugger. You use this debugger to run, stop, and kill more than one core simultaneously. Also, you can set linked breakpoints across multiple cores.

StarCore Utilities

The CodeWarrior software has these additional utilities:

- SC100 Archiver
- disasmsc100 Disassembler
- ELF File Dump Utility
- Name Utility
- Size Utility

SC100 Archiver

The SC100 archiver groups separate object files into a single file for linking or archival storage. You can add, extract, delete, and replace files in an existing archive. See SC100 Archiver Utility for more information.

disasmsc100 Disassembler

The `disasmsc100` utility disassembles both SC140 and SC140E DSP binaries. It features:

- Interpretation of relocation information
- Data disassembling
- Label (symbol) address output
- Padding awareness (alignment)
- Statistics display

See disasmsc100 Disassembler for more information.

ELF File Dump Utility

The Executable and Linkable Format (ELF) file dump utility outputs the headers of each ELF object file specified on the command line in a human-readable form. The ELF object-file type determines the output that the ELF dump utility generates. See SC100 ELF Dump Utility for more information.

Name Utility

The name utility displays the symbolic information for each object file and library specified on the command line. Also, the utility reports that a file lacks symbolics. See SC100 Name Utility for more information.

Size Utility

The size utility outputs the size (in bytes) of each section of each ELF object file specified on the command line. The default output provides sizes for all `.text`, `.rodata`, `.data`, and `.bss` sections. See SC100 Size Utility for more information.

Tutorial

This chapter shows you how to use the CodeWarrior™ development tools to create, build, and debug a sample project. This chapter has these topics:

- Using Stationery
- Working with the Project
- Debugging the Project

Using Stationery

You use *project stationery* to create most new projects. The project stationery has placeholder files and specifies default settings for specific targets. After you create a new project, you can customize it to suit your needs. The IDE has project stationery that you can use to create projects for MXC 05, i.300-30, MXC275-30, MXC300-10, and MXC300-30 targets.

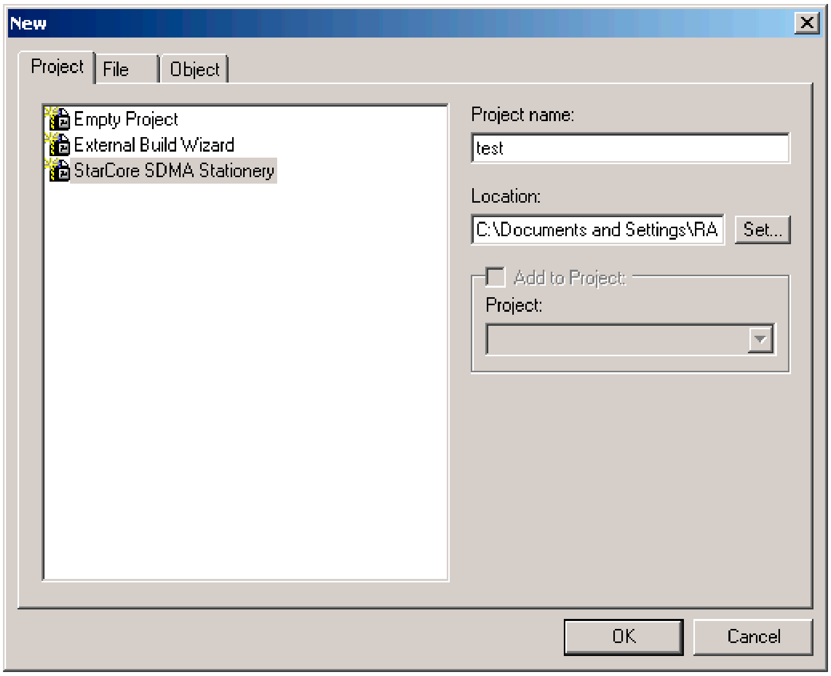
To use stationery to create a new project, follow these steps:

1. Select **File** > **New**

   The **New** window (Figure 2.1) appears. The **Project** page of this window shows a list of available project stationery.

2. From the stationery list, select **StarCore**™ **SDMA Stationery** to create a project that targets the StarCore and SDMA architecture.
3. In the **Project name** text box, enter the name of the new project.
4. In the **Location** text box, enter the path in which to create the project.

Figure 2.1 The New Window



Empty Project
External Build Wizard
StarCore SDMA Stationery

Project name: test

Location: C:\Documents and Settings\RA

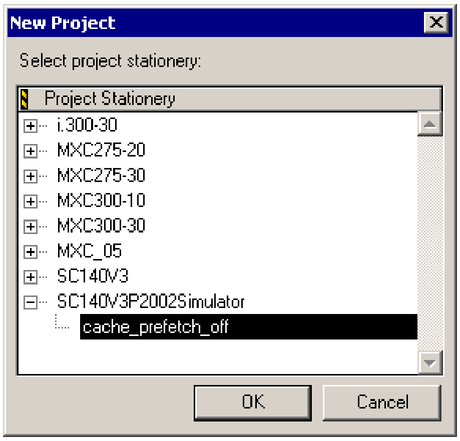
☐ Add to Project:

Project:

5. Click the **OK** button

   The **New Project** window (Figure 2.2) appears.

Figure 2.2 New Project Window



Select project stationery:

Project Stationery
- i.300-30
- MXC275-20
- MXC275-30
- MXC300-10
- MXC300-30
- MXC_05
- SC140V3
- SC140V3P2002Simulator
  - cache_prefetch_off

NOTE The New Project window shows project stationery for various targets. The MXC 05, i.300-30, MXC275-30, MXC300-10, and MXC300-30 targets provide stationery for a particular core (**SDMA** or **StarCore**) or for all cores (**MultiCore**). The **SC140V3** target has stationery for C programming, assembly programming, or a mixture of both. Targets with the **Palladium** suffix identify project stationery for the Cadence® Palladium™ emulator. The **SC140V3P2002Simulator** target has stationery for the StarCore Platform 2002 Simulator.

6. From the list in the New Project window, select the stationery for the project you want to create. For example, the IDE uses the stationery selected in Figure 2.2 to create a project that targets the SC140 v3 Platform 2002 Simulator.
7. Click the **OK** button.

   The IDE uses the selected stationery to create a new CodeWarrior project. The project window for this new project appears in the IDE.

NOTE The new project includes the placeholder file `SC140_main.c`.

Working with the Project

After you complete the steps in "Using Stationery" on page 17, you have a new project. This section explains the steps for working with the project in these ways:

- View Target Settings
- Build the Project

View Target Settings

Follow these steps to view target settings:

1. Select **Edit** > ***TargetName*** **Settings**, where ***TargetName*** is the name of the current build target.

NOTE For this tutorial, select **Edit** > **C for SC Simulator Settings**.

The Target Settings window (Figure 2.3) appears.

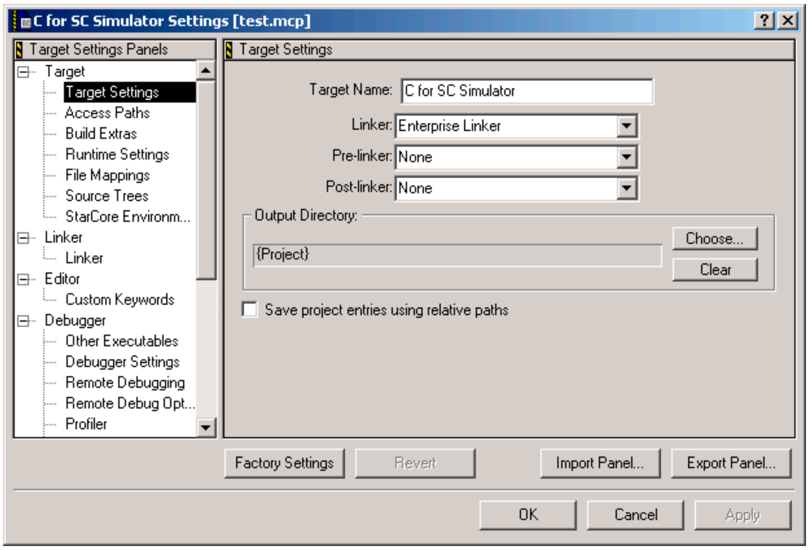
TIP To quickly display a build target's settings, click the **Targets** tab of the project window, then double-click the build-target's name in the list that appears. You can use this method to change settings for two or more build targets simultaneously.

The Target Settings window groups all options into a series of panels. The panel names appear on the left side of the window, in the **Target Settings Panels** list. When you click a panel's name in the list, the panel itself appears on the right side of the window.

Different panels affect:

- settings for all build targets
- settings for a specific build target (including settings that affect object-code generation and linker output)
- settings for a specific programming language

Figure 2.3 The Target Settings Window

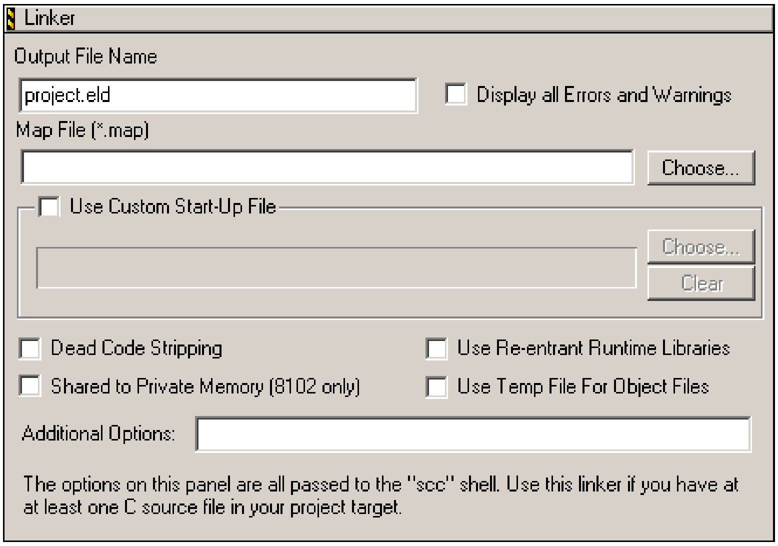


Target Name: C for SC Simulator
Linker: Enterprise Linker
Pre-linker: None
Post-linker: None
Output Directory: {Project}
☐ Save project entries using relative paths

2. Select **Linker** from the Target Settings Panels list.

   The Linker panel (Figure 2.4) appears.

Figure 2.4 Linker Panel



Linker

Output File Name: project.eld

☐ Display all Errors and Warnings

Map File (*.map):

☐ Use Custom Start-Up File

☐ Dead Code Stripping

☐ Use Re-entrant Runtime Libraries

☐ Shared to Private Memory (8102 only)

☐ Use Temp File For Object Files

Additional Options:

The options on this panel are all passed to the "scc" shell. Use this linker if you have at at least one C source file in your project target.

The **Output File Name** text box shows the name of the output file. This file has the extension `.eld`.

3. Browse other settings panels in the Target Settings window.
4. Click the Close box of the Target Settings window.

Build the Project

Select **Project** > **Make** to build the project.

After you select the **Make** command, the IDE compiles and links all the object code in the current build target, then generates an executable file.

NOTE The IDE updates all changed files so that it compiles the latest version of each file. The IDE tracks these dependencies automatically.

Debugging the Project

This section explains the steps for debugging the project:

- Start Debugging
- Set a Breakpoint

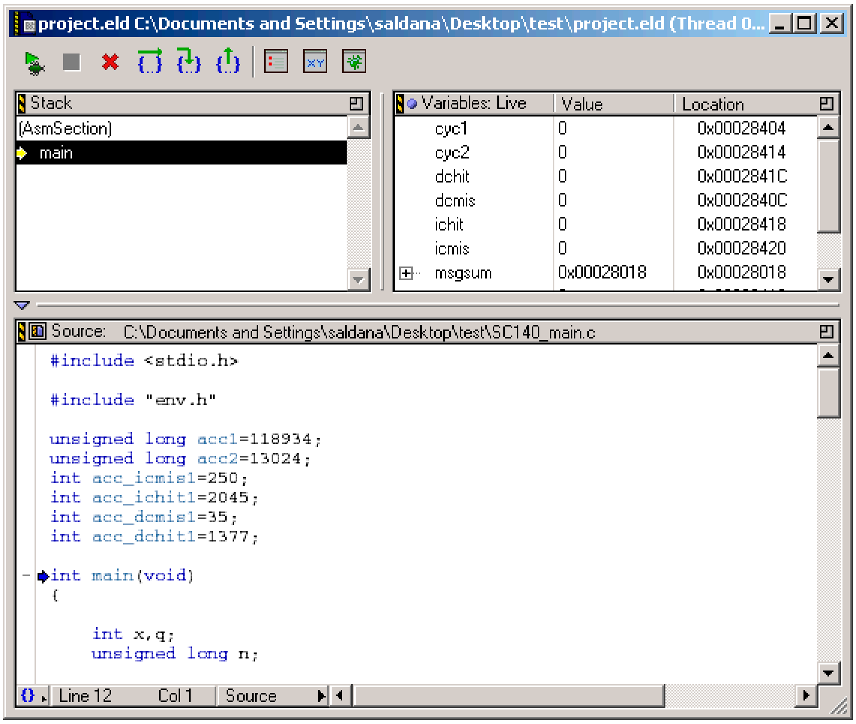
- Show Registers
- Show Instruction and Data Caches
- Finish Debugging

Start Debugging

Select **Project** > **Debug** to have the debugger start controlling your program's execution.

The debugger displays a message box while its downloads your program to the target board, then the debugger window (Figure 2.5) appears.

Figure 2.5 Debugger window



project.eld C:\Documents and Settings\saldana\Desktop\test\project.eld (Thread 0...

Stack
(AsmSection)
main

Variables: Live	Value	Location
cyc1	0	0x00028404
cyc2	0	0x00028414
dchit	0	0x0002841C
dcmis	0	0x0002840C
ichit	0	0x00028418
icmis	0	0x00028420
msgsum	0x00028018	0x00028018

Source: C:\Documents and Settings\saldana\Desktop\test\SC140_main.c

```
#include <stdio.h>

#include "env.h"

unsigned long acc1=118934;
unsigned long acc2=13024;
int acc_icmis1=250;
int acc_ichit1=2045;
int acc_dcmis1=35;
int acc_dchit1=1377;

int main(void)
{

    int x,q;
    unsigned long n;
```

Line 12 Col 1 Source

Set a Breakpoint

Follow these steps to set a breakpoint:

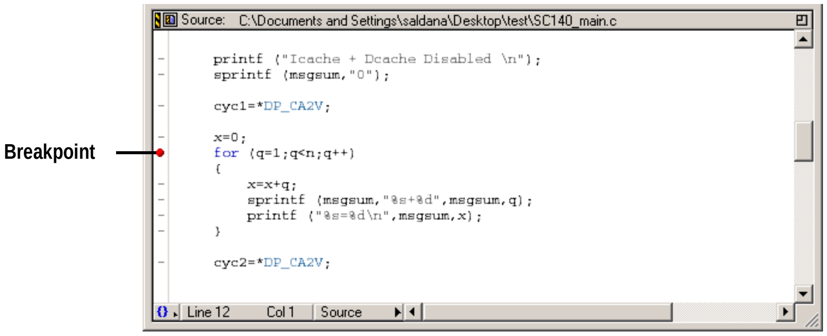
1. In the debugger window, click the gray dash in the Breakpoint column next to this statement:

   ```
   for (q=1;q<n;q++)
   ```

   A red dot appears next to the statement (Figure 2.6).

NOTE Also, you can set a breakpoint by clicking in the Breakpoint column next to any executable statement that appears in an editor window.

Figure 2.6 Debugger Window after Setting a Breakpoint

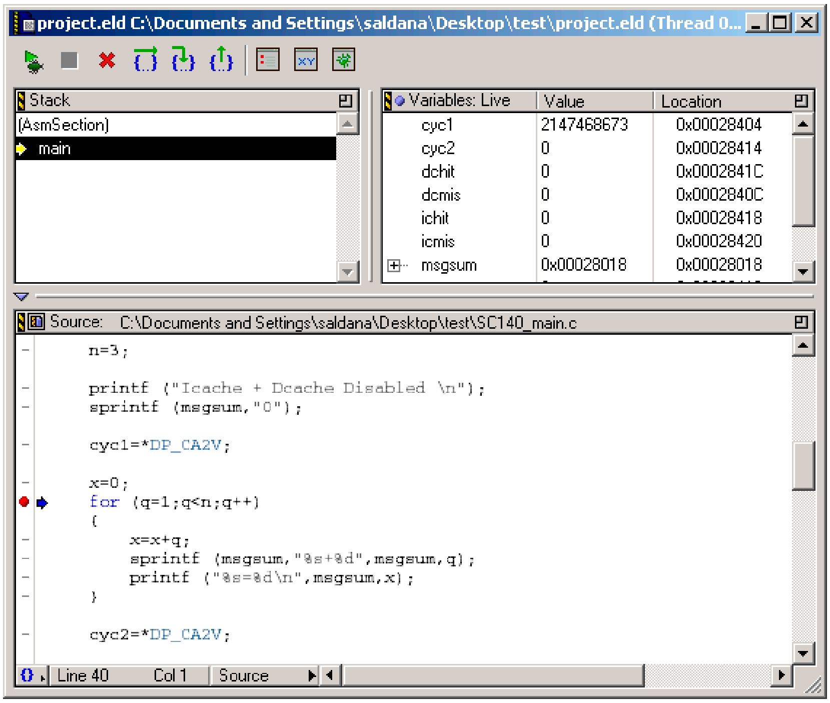

Breakpoint
Source: C:\Documents and Settings\saldana\Desktop\test\SC140_main.c
Line 12
Col 1
Source

```
printf ("Icache + Dcache Disabled \n");
sprintf (msgsum,"0");

cyc1=*DP_CA2V;

x=0;
for (q=1;q<n;q++)
{
    x=x+q;
    sprintf (msgsum,"%s+%d",msgsum,q);
    printf ("%s=%d\n",msgsum,x);
}

cyc2=*DP_CA2V;
```

2. Select **Project** > **Run** to have the debugger continue program execution up to the statement where you set the breakpoint.

   Figure 2.7 shows the debugger window after program execution reaches the breakpoint.

Figure 2.7 Debugger Window after Execution Reaches the Breakpoint


project.eld C:\Documents and Settings\saldana\Desktop\test\project.eld (Thread 0...
Stack
Source:

```
(AsmSection)
main
```

Variables: Live	Value	Location
cyc1	2147468673	0x00028404
cyc2	0	0x00028414
dchit	0	0x0002841C
dcmis	0	0x0002840C
ichit	0	0x00028418
icmis	0	0x00028420
msgsum	0x00028018	0x00028018

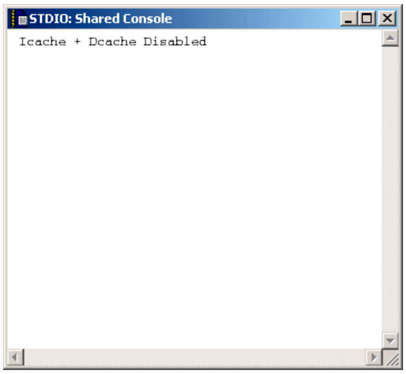
Source: C:\Documents and Settings\saldana\Desktop\test\SC140_main.c

```
    n=3;

    printf ("Icache + Dcache Disabled \n");
    sprintf (msgsum,"0");

    cyc1=*DP_CA2V;

    x=0;
    for (q=1;q<n;q++)
    {
        x=x+q;
        sprintf (msgsum,"%s+%d",msgsum,q);
        printf ("%s=%d\n",msgsum,x);
    }

    cyc2=*DP_CA2V;
```

Line 40 Col 1 Source

Also, the IDE displays an output window (Figure 2.8).

Figure 2.8 Example Program Output Window



```
Icache + Dcache Disabled
```

You just finished running your program under debugger control, setting a breakpoint, and letting your program run to this breakpoint.

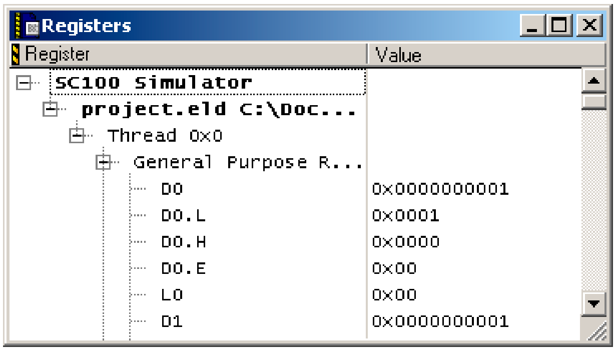
Show Registers

Follow these steps to show registers:

1. Select **View > Registers**.

   The **Registers** window (Figure 2.9) appears. This window has a tree control that lets you display the registers of the processor you are using.

Figure 2.9 Registers Window



```
SC100 Simulator
  project.eld C:\Doc...
    Thread 0x0
      General Purpose R...
        D0        0x0000000001
        D0.L      0x0001
        D0.H      0x0000
        D0.E      0x00
        L0        0x00
        D1        0x0000000001
```

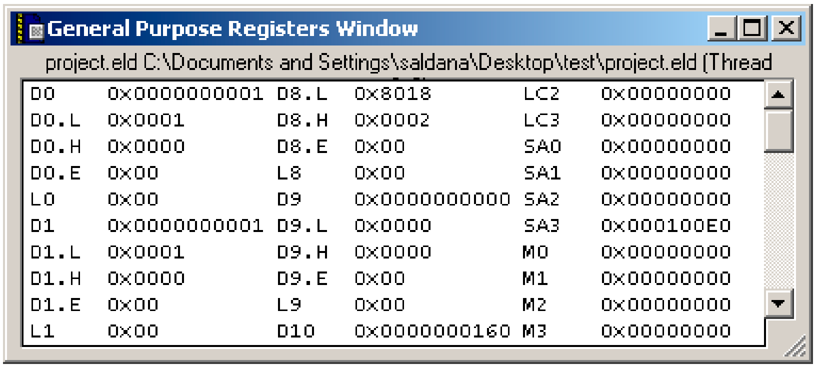
2. Choose registers from the Registers window.

   For this example, double-click:

   `General Purpose Registers`

   A new window appears for the selected registers (Figure 2.10).

Figure 2.10 General Purpose Registers Window



```
project.eld C:\Documents and Settings\saldana\Desktop\test\project.eld [Thread
D0     0x0000000001 D8.L  0x8018       LC2   0x00000000
D0.L   0x0001       D8.H  0x0002       LC3   0x00000000
D0.H   0x0000       D8.E  0x00         SA0   0x00000000
D0.E   0x00         L8    0x00         SA1   0x00000000
L0     0x00         D9    0x0000000000 SA2   0x00000000
D1     0x0000000001 D9.L  0x0000       SA3   0x000100E0
D1.L   0x0001       D9.H  0x0000       M0    0x00000000
D1.H   0x0000       D9.E  0x00         M1    0x00000000
D1.E   0x00         L9    0x00         M2    0x00000000
L1     0x00         D10   0x0000000160 M3    0x00000000
```

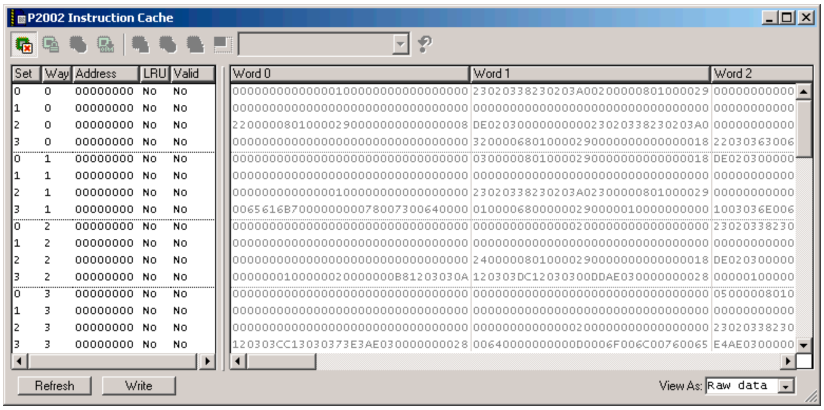
Show Instruction and Data Caches

Follow these steps to show the instruction and data caches:

1. Select **Data** > **View Cache** > **P2002 Simulator Instruction Cache**.

   The **P2002 Instruction Cache** window (Figure 2.11) appears. This window lets you display and modify the contents of the instruction cache.

Figure 2.11 P2002 Instruction Cache Window



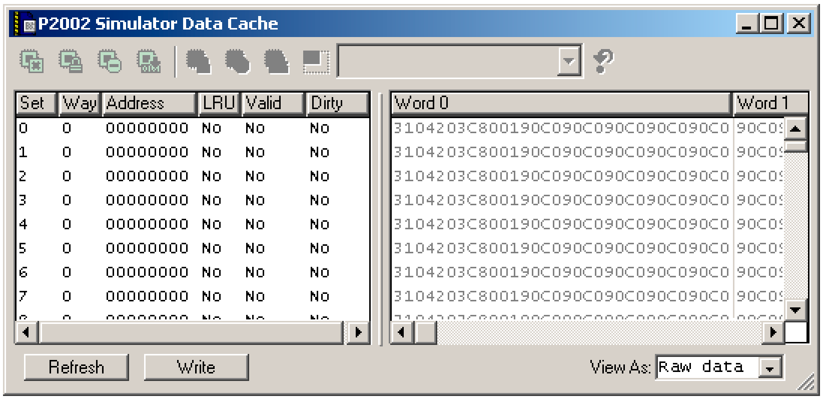
P2002 Instruction Cache

Set	Way	Address	LRU	Valid	Word 0	Word 1	Word 2
0	0	00000000	No	No	00000000000000010000000000000000	23020338230203A00200000801000029	00000000000
1	0	00000000	No	No	00000000000000000000000000000000	00000000000000000000000000000000	00000000000
2	0	00000000	No	No	22000008010000290000000000000008	DE020300000000002302033823020 3A0	00000000000
3	0	00000000	No	No	00000000000000000000000000000000	32000068010000290000000000000018	22030363006
0	1	00000000	No	No	00000000000000000000000000000000	03000008010000290000000000000018	DE020300000
1	1	00000000	No	No	00000000000000000000000000000000	00000000000000000000000000000000	00000000000
2	1	00000000	No	No	00000000000000010000000000000000	23020338230203A02300000801000029	00000000000
3	1	00000000	No	No	0065616B700000000078007300640000	01000068000000290000010000000000	1003036E006
0	2	00000000	No	No	00000000000000000000000000000000	00000000000000020000000000000000	23020338230
1	2	00000000	No	No	00000000000000000000000000000000	00000000000000000000000000000000	00000000000
2	2	00000000	No	No	00000000000000000000000000000000	24000008010000290000000000000018	DE020300000
3	2	00000000	No	No	0000000100000020000000B81203030A	120303DC12030300DDAE030000000028	00000100000
0	3	00000000	No	No	00000000000000000000000000000000	00000000000000000000000000000000	05000008010
1	3	00000000	No	No	00000000000000000000000000000000	00000000000000000000000000000000	00000000000
2	3	00000000	No	No	00000000000000000000000000000000	00000000000000020000000000000000	23020338230
3	3	00000000	No	No	120303CC13030373E3AE030000000028	00640000000000D0006F006C00760065	E4AE0300000

2. Select **Data** > **View Cache** > **P2002 Simulator Data Cache**.

   The **P2002 Data Cache** window (Figure 2.12) appears. This window lets you display and modify the contents of the data cache.

Figure 2.12 P2002 Data Cache Window



P2002 Simulator Data Cache

Set	Way	Address	LRU	Valid	Dirty	Word 0	Word 1
0	0	00000000	No	No	No	3104203C800190C090C090C090C090C0	90C0
1	0	00000000	No	No	No	3104203C800190C090C090C090C090C0	90C0
2	0	00000000	No	No	No	3104203C800190C090C090C090C090C0	90C0
3	0	00000000	No	No	No	3104203C800190C090C090C090C090C0	90C0
4	0	00000000	No	No	No	3104203C800190C090C090C090C090C0	90C0
5	0	00000000	No	No	No	3104203C800190C090C090C090C090C0	90C0
6	0	00000000	No	No	No	3104203C800190C090C090C090C090C0	90C0
7	0	00000000	No	No	No	3104203C800190C090C090C090C090C0	90C0

Finish Debugging

Select **Debug** > **Kill** to finish debugging.

You have completed this tutorial. You now know how to create a project, build it, and debug it. You worked with the major components of the CodeWarrior software: the project manager, source-code editor, and target settings panels.

3

Target Settings

This chapter explains target settings for MXC 05, i.300-30, MXC275-30, MXC300-10, and MXC300-30 development. Use these settings to control the behavior of the compiler, linker, debugger, and other CodeWarrior™ software development tools.

NOTE For documentation of generic target settings included in all CodeWarrior products, refer to the *IDE User's Guide*.

This chapter has these topics:

- Overview
- General Purpose Target Settings Panels
- Target Settings Panels for StarCore and SDMA Development
- Memory-Configuration Files

Overview

A CodeWarrior project can contain multiple *build targets*. A build target is a named collection of files and settings that the CodeWarrior IDE uses to generate an output file. For example, the project you worked with in the Tutorial chapter had a build target named **C for SC Simulator**.

A build target contains all build-specific *target settings*. Target settings define:

- the files that belong to a build target
- the behavior of the compiler, assembler, linker, and other build tools

Build targets and their associated target settings let you create different versions of your program for different purposes. For example, you might have these build targets:

- a *debug* build target that is not optimized, making it easier to debug
- a *release* build target that it heavily optimized for increased speed or reduced memory requirements

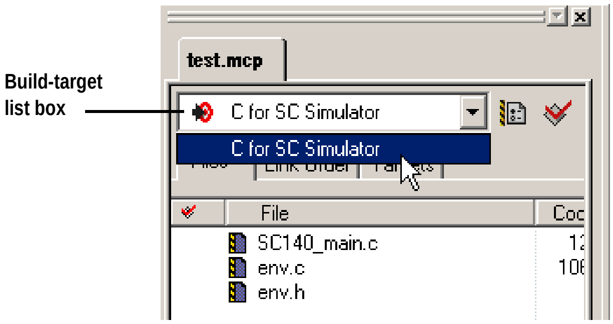
Changing Target Settings

If you use project stationery to create a new project, the IDE automatically sets the target settings of each build target to reasonable default values. You can change these default values to better suit your needs.

Follow these steps to change target settings:

1. Start the CodeWarrior IDE.
2. Open or bring forward the project that has the build target you want to modify.
3. Use the build-target list box (Figure 3.1) in the project-window toolbar to select the build target that you want to modify.

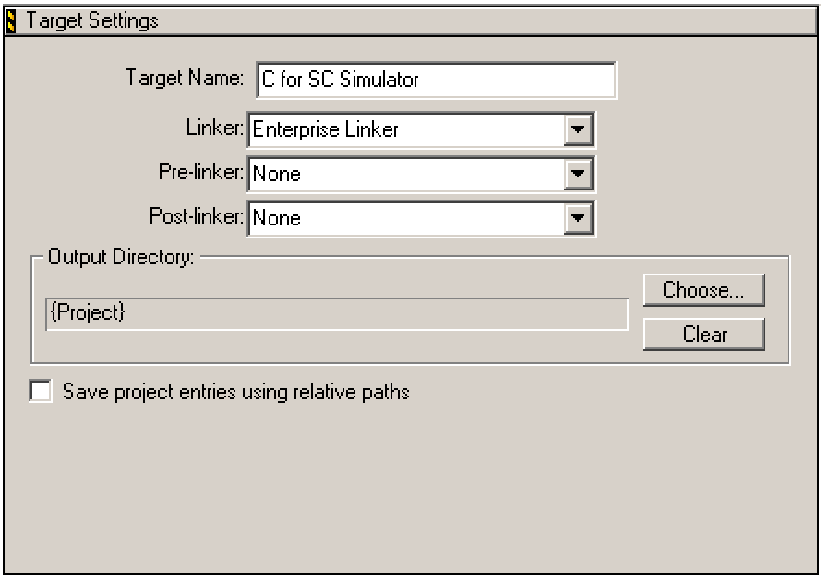
Figure 3.1 Project Window Showing the Selection of a Build Target



4. Select **Edit > *TargetName* Settings**, where ***TargetName*** is the name of the current build target in the CodeWarrior project.

   The Target Settings window (Figure 3.2) appears. The left side of this window has the **Target Settings Panels** list, which shows the settings panels that apply to the build target you selected in the build-target list box. Also, your **Linker** and **Post-linker** selections in the **Target Settings** panel determine the settings panels shown in the list.

Figure 3.2 Target Settings Window


Target Settings

Target Name: C for SC Simulator

Linker: Enterprise Linker

Pre-linker: None

Post-linker: None

Output Directory: {Project}

☐ Save project entries using relative paths

5. In the Target Settings Panels list, select a panel name.

   The Target Settings window shows the panel that you selected.

6. Change settings in the panel.

7. Click the **Apply** button.

   The IDE saves your new settings.

You can select other panels and modify their settings. When you finish, click the **OK** button in the Target Settings window. The IDE saves your changes and closes the window.

Creating New Stationery

You might find it useful to configure a project's build targets such that you can use the project as a template for creating other projects. The IDE lets you create project stationery for this purpose.

To create project stationery based on an existing project, follow these steps:

1. Create a project.

2. For each build target in the project, change the target settings as needed.

3. Select **File** > **Save a Copy As**

   The **Save a copy of project as** dialog box appears.

4. Use the dialog box to save the project in the CodeWarrior `Stationery` directory, where *`CodeWarrior`* is the directory in which you installed the CodeWarrior product:

   `CodeWarrior``\Stationery`

The next time you use project stationery to create a new project, the stationery you just created appears in the **New Project** dialog box.

Restoring Target Settings

After you change target settings in a project, you can restore their previous values or their default values. The process you follow depends on which values you want to restore:

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

General Purpose Target Settings Panels

Some target settings panels apply to all development done with the CodeWarrior IDE. Table 3.1 lists each target settings panel that is *not* specific to development for the StarCore™ and SDMA cores. Refer to the *IDE User's Guide* for documentation of these panels.

Table 3.1 General Purpose Target Settings Panels

Target Settings Panel	Description
Access Paths	Defines the list of directories that the build tools search for `#include` files.
Build Extras	Specifies options that affect the performance of the software development tools. Also configures third-party debuggers.
Runtime Settings	Specifies information, such as command-line arguments, that your program needs when run under control of the CodeWarrior IDE.
File Mappings	Associates each filename extension with a specific tool.
Source Trees	Defines aliases for paths that change from one computer to the next. Using source trees makes it easier to share a project among multiple developers.

Table 3.1 General Purpose Target Settings Panels (*continued*)

Target Settings Panel	Description
Custom Keywords	Defines up to four sets of custom keywords, along with the color the editor uses for each.
Debugger Settings	Configures generic debugger behavior.

Target Settings Panels for StarCore and SDMA Development

Table 3.2 lists the target settings panels specific to developing software for devices with StarCore and SDMA cores.

Table 3.2 Panels for StarCore and SDMA Development

Category	Panel	Settings for
Target	Target Settings	Target operating system, microprocessor, and build-target name
	IPCM Target	Development-tool selection, output type, and command-line display for the Interprocessor Communication Module (IPCM)
	StarCore Environment	Byte order (endian), memory model, and whether to display generated command lines in a message window
	SDMA Target	Whether to display generated command lines in the message window
Code Generation	IPCM Assembler	Display options, assembly preprocessors, and additional command-line options for the IPCM assembler
	SDMA Assembler	Assembly preprocessors and additional command-line options for the Smart Direct Memory Access (SDMA) assembler

Table 3.2 Panels for StarCore and SDMA Development (*continued*)

Category	Panel	Settings for
Linker	IPCM Linker	Output filename, memory-map generation, linker optimizations, relocatable files, and additional command-line options for the IPCM linker.
	External Build	Build command, build directory, output filename, and debugging platform for the external-build linker
	DSP Linker	The DSP linker. To use this panel, you must select **DSP Linker** from the Linker list box of the Target Settings panel.
	Linker	The Linker. The IDE passes the `-Xlink` option to the linker for each option that you select. To use this panel, you must select **Enterprise Linker** from the Linker list box of the Target Settings panel.
	DSP Librarian	The name of the library the build target uses and the command-line options that pass to the archiver utility To use this panel, you must select **DSP Librarian** from the Linker list box of the Target Settings panel.
	SDMA Linker	The SDMA Linker To use this panel, you must select **SDMA Linker** from the Linker list box of the Target Settings panel.

Table 3.2 Panels for StarCore and SDMA Development (*continued*)

Category	Panel	Settings for
Debugger	Other Executables	Other projects and files that the debugger uses in addition to the executable file that the current build target generates To use the multi-core debugging feature, use this panel to specify the paths to the CodeWarrior projects for each core that you want to debug.
	Remote Debugging	The communication protocol for the target hardware
	Remote Debug Options	Which parts of your program to download to the target, when to download these parts, and whether to verify them
	SDMA Target Settings	The target board, reset option, symbolic-download setting, and initialization file for the SDMA controller
	Nexus Configuration	Nexus debugging and ARM® RealView® trace options
	Nexus Data Trace Configuration	The name, addressability, and memory offset of the S-Record file that the `elfsrec` utility generates
	Nexus General Trace Configuration	Watchpoint-messaging and generic trace
	Nexus Program Trace Configuration	Program-trace triggering and time stamps
	SC100 Debugger Target	The simulator or device on which you debug a build target's binary file, how the CodeWarrior debugger interacts with the selected device, and how the debugger behaves at startup and during a debugging session To use this panel, you must select **SC100 ELF Dump**, **SC100 ELF to LOD**, or **SC100 ELF to S-Record** from the Post-linker list box of the Target Settings panel.

Table 3.2 Panels for StarCore and SDMA Development (*continued*)

Category	Panel	Settings for
	Source Folder Mapping	Path information for debugging an executable file built in one place, but debugged from another place
	VTB Configuration	The Debugging and Profiling Unit (DPU) trace registers and viewing trace-buffer information
StarCore Assembler	Assembler Preprocessors	The assembler, such as search paths for files and file-handling settings
	Listing File Options	The format and contents of the source listing file. Also, you can specify other assembler options in the **Additional Options** text box.
	Code & Language Options	Symbols and the StarCore Assembler
StarCore Compiler	C Language	The version of C that you use. If you use the default version, you do not need to specify any settings in this panel.
	Compiler	The behavior of the compiler, such as where the IDE stops processing files and whether the compiler includes debugging information in the output file
	I/O & Preprocessors	Additional directories for the IDE to search, as well as define and undefine preprocessor macros
	Optimizations	Several types of optimizations, such as for space and time, and whether the IDE applies optimizations globally
	Passthrough, Hardware	Options and arguments to pass to specified tool components

Table 3.2 Panels for StarCore and SDMA Development (*continued*)

Category	Panel	Settings for
Post-Linker	SC100 ELF Dump	The behavior of the ELF file dump utility To use this panel, you must select **SC100 ELF Dump** from the Post-linker list box of the Target Settings panel.
	SC100 ELF to LOD	The name of the LOD file that the elflod utility generates To use this panel, you must select **SC100 ELF to LOD** from the Post-linker list box of the Target Settings panel.
	SC100 ELF to S-Record	The name, addressability, and memory offset of the S-Record file that the elfsrec utility generates. To use this panel, you must select **SC100 ELF to S-Record** from the Post-linker list box of the Target Settings panel.
Other	Profiler	The CodeWarrior profiler, so that it can interact with your target and collect information about the program running on that target

Target Settings

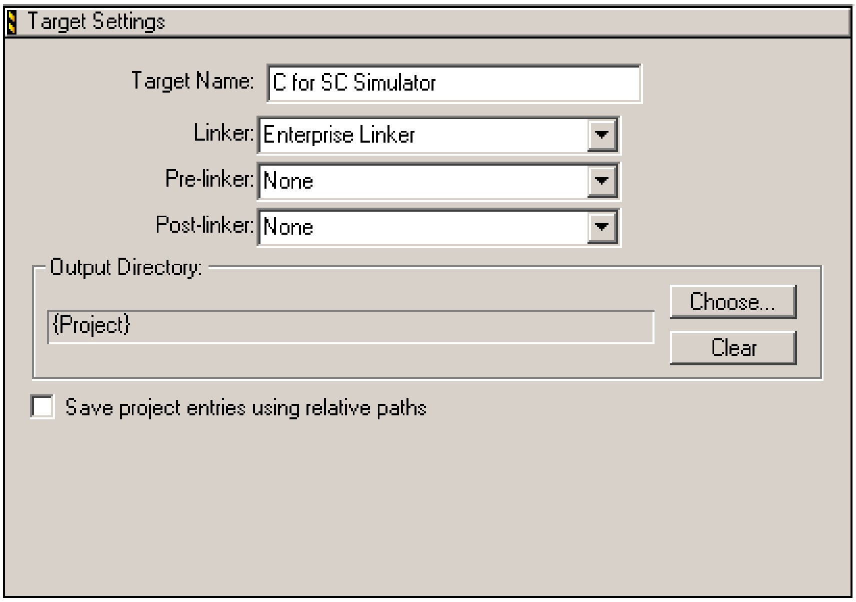
Use this panel to specify the name, linker, pre-linker, post-linker, and output directory of the current build target.

NOTE The ***Target* Settings** *window* contains a **Target Settings** *panel*. The window and the panel are not the same.

The **Target Settings** window displays the **Target Settings** panel if you select Target Settings from the list on the left side of the **Target Settings** window.

Figure 3.3 shows the **Target Settings** panel. Table 3.3 explains each panel option.

Figure 3.3 Target Settings Panel



Target Name: C for SC Simulator

Linker: Enterprise Linker

Pre-linker: None

Post-linker: None

Output Directory: {Project}

[ ] Save project entries using relative paths

Table 3.3 Target Settings Panel Options

Option	Explanation
Target Name	Enter the name to the current build target. The name you specify appears in the build target list box and in the Targets view of a project window. Note that this name is the name of the current build target, not the name of the file that this build target generates. You specify a build target’s output filename in the **Output File Name** text box of the Linker, DSP Linker, or DSP Librarian panel.
Linker	Use this list box to specify the linker the current build target uses: • **DSP Linker**—The IDE directly invokes the linker (sc100-ld). This linker is appropriate for build targets that consist entirely of assembly-language source files because the linker does not link with the C runtime library. • **Enterprise Linker**—The IDE invokes the compiler shell (scc) which, in turn, invokes the linker. Use either of these linkers with build targets that include even one C-language source file because the linker links with the C runtime library. • **DSP Librarian**—The IDE puts the build target’s output in a library (`.elb`) file. Use the DSP Librarian panel to specify the library a build target uses and whether the build target replaces or adds its output to the library. The linker that you specify determines which target settings panels appear in the **Target Settings Panels** list of the Target Settings window.
Pre-linker	This setting does not apply to StarCore or SDMA software development.

Table 3.3 Target Settings Panel Options (*continued*)

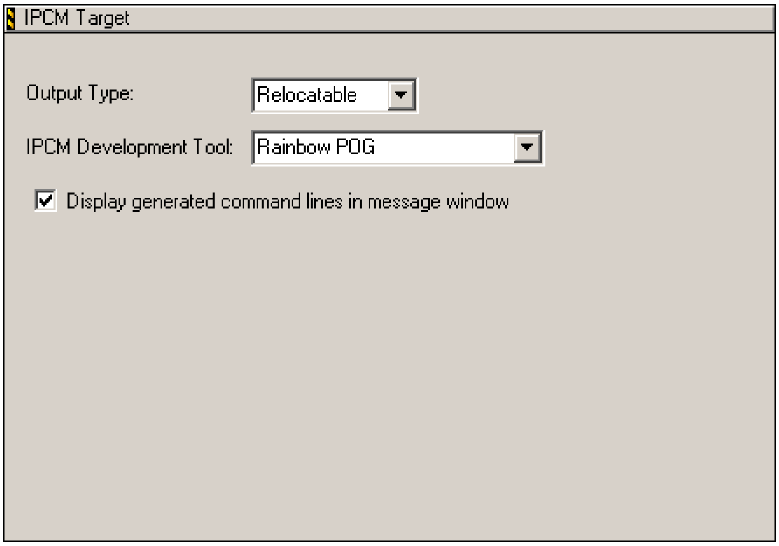
Option	Explanation
Post-linker	Use this list box to select the post-linker the current build target uses: • **None**—Do not use a post-linker. • **SC100 ELF Dump**—The IDE runs the sc100-elfdump utility on the output that the build target generates. If you select this post-linker, the SC100 ELF Dump panel appears in the Target Settings Panels list. Use this panel to define the utility’s behavior. See SC100 ELF Dump Utility for more information • **SC100 ELF to LOD**—The IDE runs the elflod utility on the output that the build target generates. If you select this post-linker, the SC100 ELF to LOD panel appears in the Target Settings Panels list. Use this panel to define the utility’s behavior. See ELF to LOD Utility for more information • **SC100 ELF to S-Record**—The IDE runs the elfsrec utility on the output that the build target generates. If you select this post-linker, the SC100 ELF to S-Record panel appears in the Target Settings Panels list. Use this panel to define the utility’s behavior. See ELF to S-Record Utility for more information
Output Directory	This read-only text box shows the path to which the build target writes its output. To specify an output directory, click the **Choose** button, then use the resulting dialog box. To restore the default output directory (the project directory), click the **Clear** button.
Save project entries using relative paths	Checked—The IDE saves the relative path of each file in a build target, along with the root file name of the file. After checking the checkbox, you can add two or more files that have the same name to a project. When searching for files, the IDE prepends the directory names in the **Access Paths** target settings panel to the relative path of each project file. The IDE uses these unique paths to distinguish files with the same name. Cleared—The IDE requires a unique name for each file in the project. After clearing the checkbox, you cannot add two or more files that have the same name to a project. When searching for files, the IDE combines the directory names in the **Access Paths** panel with just the root filename of each project file. As a result, the IDE cannot distinguish two files that have the same name but different relative paths.

IPCM Target

Use this panel to select options that control the behavior of the Interprocessor Communication Module (IPCM). To view this panel in the Target Settings window, you must go to the **Target Settings** panel and use the **Linker** list box to specify **IPCM Linker**.

Figure 3.4 shows the **IPCM Target** settings panel. Table 3.4 explains each panel option.

Figure 3.4 IPCM Target Settings Panel


IPCM Target

Output Type: Relocatable

IPCM Development Tool: Rainbow POG

☑ Display generated command lines in message window

Table 3.4 IPCM Target Settings Panel Options

Option	Explanation
Output Type	Use this list box to specify the type of executable code that the IDE produces for the IPCM: • **Absolute**—Specify this type when you compile object code that does not invoke the linker. • **Relocatable**—Specify this type when you compile object code that invokes the linker or relocator.

Table 3.4 IPCM Target Settings Panel Options (*continued*)

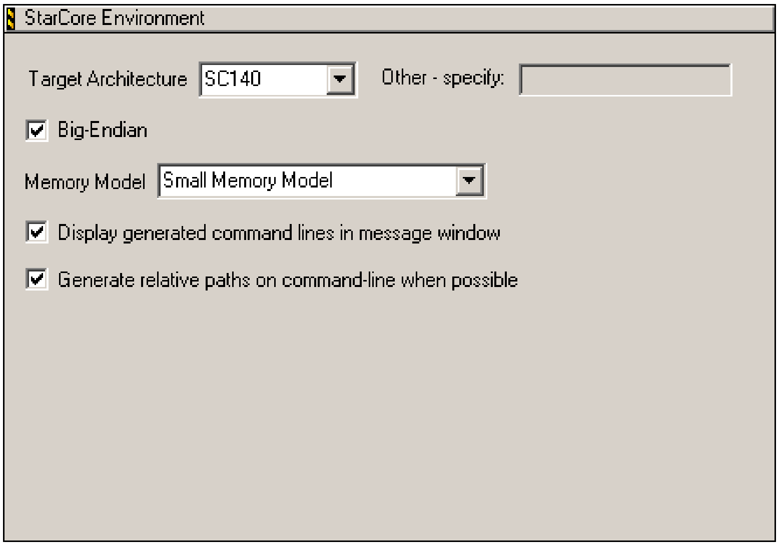
Option	Explanation
IPCM Development Tool	Use this list box to specify the development tool that the IDE uses to produce output for the IPCM. The tool that you select depends on the target hardware. The IDE invokes the specified tool to build an `.elf` file.
Display generated command lines in message window	Checked—The IDE shows the command lines it passes to the build tools. The command lines appear in the **Errors and Warnings** window. Cleared—The IDE hides the command lines it passes to the build tools.

StarCore Environment

Use this panel to specify the StarCore architecture the build target uses, the byte order (endian) and memory mode this architecture uses, and how the IDE handles the command lines it passes to the shell program (scc).

Figure 3.5 shows the **StarCore Environment** target settings panel. Table 3.5 explains each panel option.

Figure 3.5 StarCore Environment Target Settings Panel



StarCore Environment

Target Architecture: SC140
Other - specify:

[✓] Big-Endian

Memory Model: Small Memory Model

[✓] Display generated command lines in message window

[✓] Generate relative paths on command-line when possible

Table 3.5 StarCore Environment Panel Options

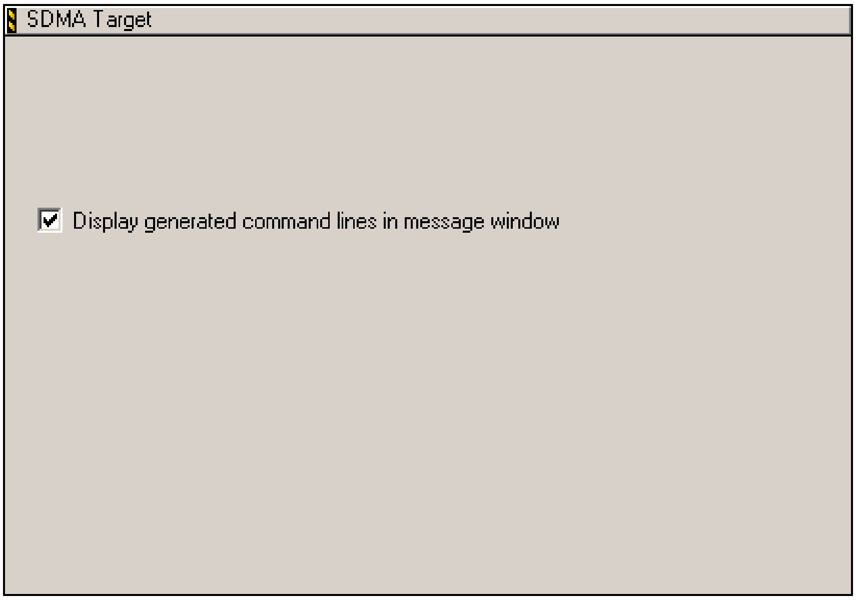
Option	Explanation
Target Architecture	Use this list box to specify the StarCore architecture that you are targeting.
Other - specify	Use this text box to pass an architecture identifier other than the ones available in the Target Architecture list box to the StarCore development tools. The IDE passes `-arch` followed by the string you enter. Note that you must select **OtherArch** from the Target Architecture list box to enable the Other - specify text box.
Big-Endian	Checked—The compiler, assembler, and linker use big-endian byte ordering. Cleared—The compiler, assembler, and linker use little-endian byte ordering.
Memory Model	Use this list box to specify the memory model for the build tools: • **Small Memory Model—**Absolute addresses fit in 64KB. • **Big Memory Model—**Absolute addresses do not fit in 64KB. • **Big Memory Model w/ Far RT Lib Calls—**Absolute addresses do not fit in 64KB. The build tools make runtime-library calls in the same manner they do for the huge model. • **Huge Memory Model—**Absolute addresses do not fit in 1MB.
Display generated command lines in message window	Checked—The IDE shows the command lines it passes to the build tools. The command lines appear in the **Errors and Warnings** window. Cleared—The IDE hides the command lines it passes to the build tools.
Generate relative paths on command-line when possible	Checked—The IDE puts project-relative paths in the command lines it passes to the tools. Enabling this option eliminates OS error 87 because it shortens the command lines passed to the tools. Cleared—The IDE puts absolute paths in the command lines it passes to the tools.

SDMA Target

Use this panel to select options that control the behavior of the SDMA. To view this panel in the Target Settings window, you must go to the **Target Settings** panel and use the **Linker** list box to specify **SDMA Linker**. Figure 3.6 shows the **SDMA Target** settings panel.

This panel has the **Display generated command lines in message window** checkbox. Check this checkbox to have the IDE show the command lines it passes to the SDMA target. The command lines appear in the **Errors and Warnings** window. Clear this checkbox to have the IDE hide the command lines it passes to the SDMA target.

Figure 3.6 SDMA Target Settings Panel


SDMA Target

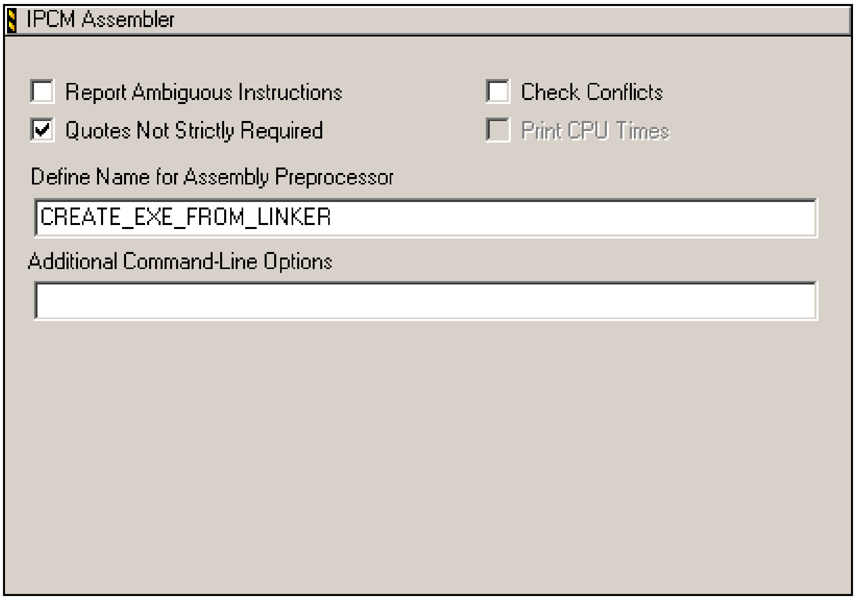
[✓] Display generated command lines in message window

IPCM Assembler

Use this panel to select options that control the behavior of the IPCM assembler. To view this panel in the Target Settings window, you must go to the **Target Settings** panel and use the **Linker** list box to specify **IPCM Linker**.

Figure 3.7 shows the **IPCM Assembler** target settings panel. Table 3.6 explains each panel option.

Figure 3.7 IPCM Assembler Target Settings Panel



- [ ] Report Ambiguous Instructions
- [✓] Quotes Not Strictly Required
- [ ] Check Conflicts
- [ ] Print CPU Times

Define Name for Assembly Preprocessor: CREATE_EXE_FROM_LINKER

Additional Command-Line Options:

Table 3.6 IPCM Assembler Target Settings Panel Options

Option	Explanation
Report Ambiguous Instructions	Checked—The IPCM assembler reports instructions with an unclear context. Cleared—The IPCM assembler does not report instructions with an unclear context.
Check Conflicts	Checked—The IPCM assembler checks your source code for conflicting statements. Cleared—The IPCM assembler does not check your source code for conflicting statements.
Quotes Not Strictly Required	Checked—The IPCM assembler strictly enforces required quotation marks in your source code. Cleared—The IPCM assembler lets you omit quotation marks from your source code.

Table 3.6 IPCM Assembler Target Settings Panel Options (*continued*)

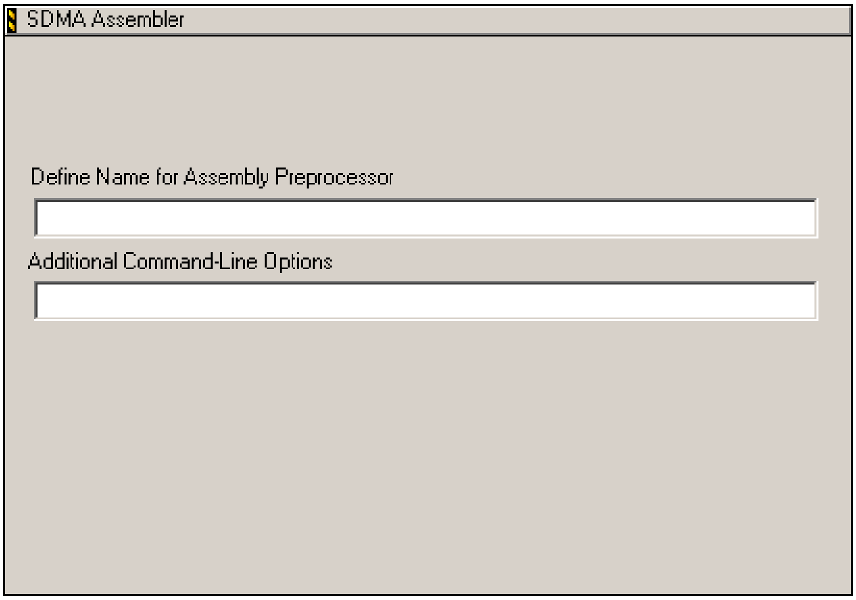
Option	Explanation
Print CPU Times	This checkbox applies to the Solaris-hosted IDE only. Checked—The IPCM assembler shows CPU times. These times appear in the **Errors and Warnings** window. Cleared—The IPCM assembler hides CPU times.
Define Name for Assembly Preprocessor	Enter the name of the assembly preprocessor that you want to use. The default entry, CREATE_EXE_FROM_LINKER, specifies that the IDE uses the IPCM linker to create the executable file.
Additional Command-Line Options	Enter additional IPCM assembler command-line options. The IDE passes these options to the IPCM assembler during the assembly phase. Note that the IDE passes command-line options to the IPCM assembler exactly as you enter them in the text box.

SDMA Assembler

Use this panel to select options that control the behavior of the SDMA assembler. To view this panel in the Target Settings window, you must go to the **Target Settings** panel and use the **Linker** list box to specify **SDMA Linker**.

Figure 3.8 shows the **SDMA Assembler** target settings panel. Table 3.7 explains each panel option.

Figure 3.8 SDMA Assembler Target Settings Panel



Define Name for Assembly Preprocessor

Additional Command-Line Options

Table 3.7 SDMA Assembler Target Settings Panel Options

Option	Explanation
Define Name for Assembly Preprocessor	Enter the name of the assembly preprocessor that you want to use.
Additional Command-Line Options	Enter additional SDMA assembler command-line options. The IDE passes these options to the SDMA assembler during the assembly phase. Note that the IDE passes command-line options to the SDMA assembler exactly as you enter them in the text box.

IPCM Linker

Use this panel to select options that control the behavior of the IPCM linker. To view this panel in the Target Settings window, you must go to the **Target Settings** panel and use the **Linker** list box to specify **IPCM Linker**.

Figure 3.9 shows the **IPCM Linker** settings panel. Table 3.8 explains each panel option.

Figure 3.9 IPCM Linker Target Settings Panel

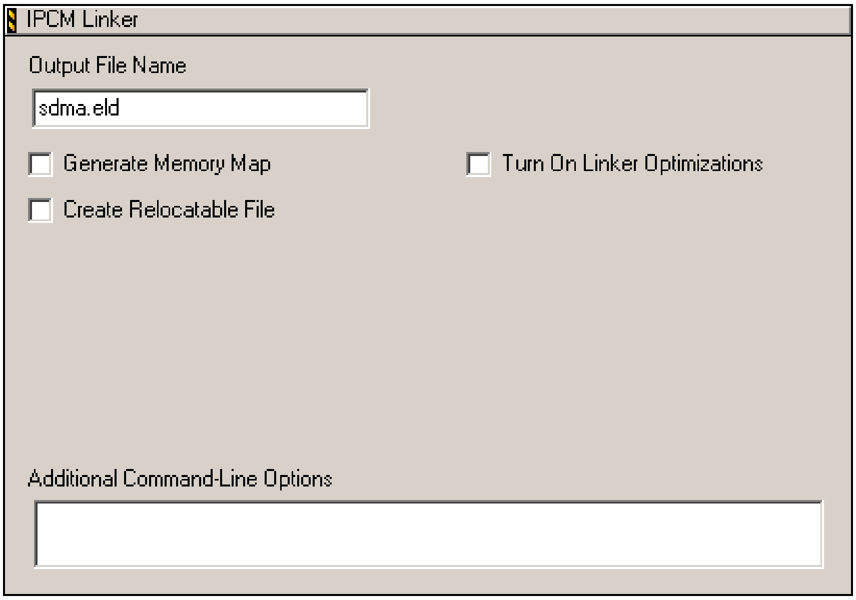


Table 3.8 IPCM Linker Target Settings Panel Options

Option	Explanation
Output File Name	Enter the output filename for your IPCM ELF application. This filename must have the `.eld` extension. If you used the Output Type list box of the IPCM Target settings panel to specify Absolute, the linker passes the output filename to the assembler.
Generate Memory Map	Checked—The IPCM linker generates a memory-map file named `bridge.map`. Checking this checkbox is equivalent to using the linker command-line option `-m`. Cleared—The IPCM linker does not generate a memory-map file.
Turn On Linker Optimizations	Checked—The IPCM linker uses optimizations as it processes object. Checking this checkbox is equivalent to using the linker command-line option `-O`. Cleared—The IPCM linker does not use optimizations as it processes object code.

Table 3.8 IPCM Linker Target Settings Panel Options (*continued*)

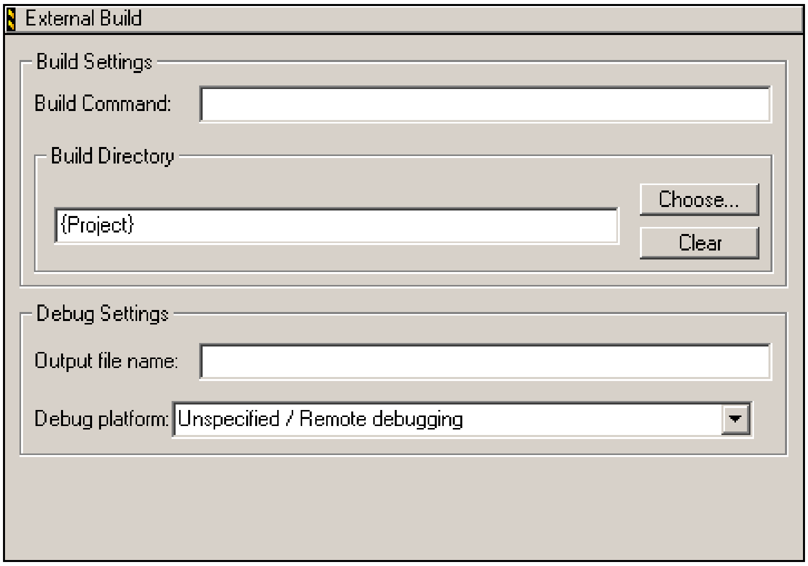
Option	Explanation
Create Relocatable File	Checked—The IPCM linker generates a relocatable object file. This file has all the information from the object files that the assembler creates. Checking this checkbox is equivalent to using the command-line option `-i`. Cleared—The IPCM linker does not generate a relocatable file.
Additional Command-Line Options	Enter additional linker command-line options that the IDE passes to the IPCM linker during the link phase. Note that the IDE passes command-line options to the IPCM linker exactly as you enter them in the text box.

External Build

Use this panel to select options that control the behavior of the external-build linker. To view this panel in the Target Settings window, you must go to the **Target Settings** panel and use the **Linker** list box to specify **External Build Linker**.

Figure 3.10 shows the **External Build** settings panel. Table 3.9 explains each panel option.

Figure 3.10 External Build Target Settings Panel



Build Directory: {Project}

Debug platform: Unspecified / Remote debugging

Table 3.9 External Build Target Settings Panel Options

Option	Explanation
Build Command	Enter a command that the IDE passes to the external-build linker. The external-build linker executes this command on the directory shown in the Build Directory text box.
Build Directory	This text box shows the path to the target directory of the command that you enter in the Build Command text box. The IDE executes the specified build command on this directory. To specify a build directory, enter its path in the text box. Alternatively, click the **Choose** button, then use the resulting dialog box. To restore the default build directory (the relative path to the project directory), click the **Clear** button.
Output File Name	Enter the name the linker gives to its generated file. This filename must have the `.eld` extension.
Debug Platform	Use this list box to specify the platform on which the executable file runs and gets debugged.

DSP Linker

Use this panel to select options that control the behavior of the DSP Linker. To view this panel in the Target Settings window, you must go to the **Target Settings** panel and use the **Linker** list box to specify **DSP Linker**.

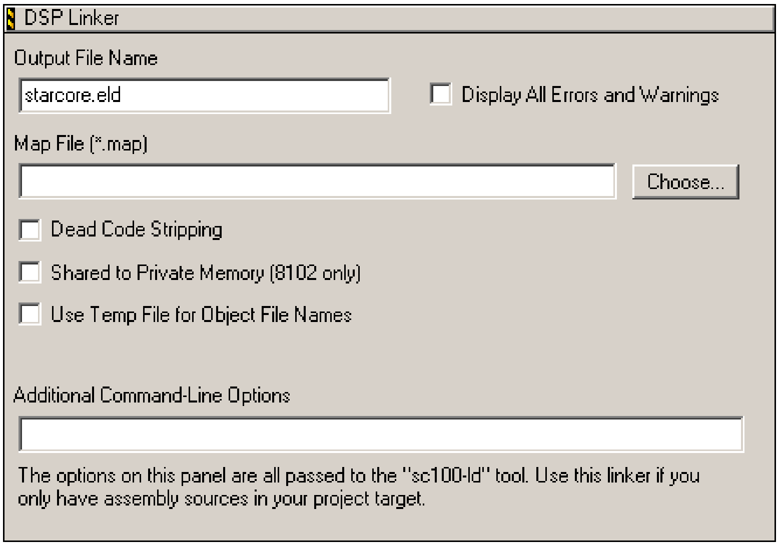
NOTE Select the DSP Linker for build targets that have just assembly-language source files. This option does not optimize the build target's output. Hand-coded assembly language does not require optimization. For build targets that include C-language source files, specify the Enterprise Linker instead. Specifying the Enterprise Linker lets the build tools optimize the build target's output.

The options of the **DSP Linker** panel are identical to those of the **Linker** panel, except that the **DSP Linker** panel does not include these options:

- Use Custom Start-Up File
- Use Re-entrant Runtime Libraries

Therefore, see Table 3.10 for an explanation of each **DSP Linker** panel option. Figure 3.11 shows the **DSP Linker** target settings panel.

Figure 3.11 DSP Linker Target Settings Panel


DSP Linker
Output File Name
starcore.eld
Display All Errors and Warnings
Map File (*.map)
Choose...
Dead Code Stripping
Shared to Private Memory (8102 only)
Use Temp File for Object File Names
Additional Command-Line Options

The options on this panel are all passed to the "sc100-ld" tool. Use this linker if you only have assembly sources in your project target.

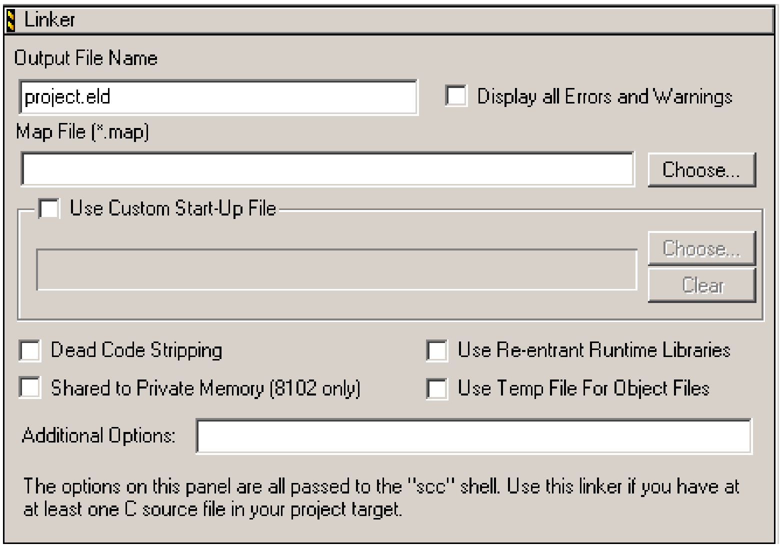
Linker

Use this panel to select options that control the behavior of the Linker. To view this panel in the Target Settings window, you must go to the **Target Settings** panel and use the **Linker** list box to specify **Enterprise Linker**.

NOTE Select the Linker for build targets that include C-language source files. This option lets the build tools optimize the build target's output. For build targets that have just assembly-language source files, specify the DSP Linker instead. Hand-coded assembly language does not require optimization.

Figure 3.12 shows the **Linker** target settings panel. Table 3.10 explains each panel option.

Figure 3.12 Linker Target Settings Panel



The options on this panel are all passed to the "scc" shell. Use this linker if you have at at least one C source file in your project target.

Table 3.10 Linker Panel Options

Option	Explanation
Output File Name	Enter the name the linker gives to its generated file. This filename must have the `.eld` extension.
Display all Errors and Warnings	Checked—The IDE shows all error and warnings messages that the linker emits. Cleared—The IDE hides all error and warning messages that the linker emits.
Map File	Enter the path of the file to which the linker writes memory-map information. This filename must have the `.map` extension.
Use Custom Start-Up File	Checked—The linker uses a custom start-up file instead of the default start-up file. Checking this checkbox enables a related text box that shows the path to the current custom start-up file. Click the **Choose** button, then use the resulting dialog box to specify a custom start-up file. Click the **Clear** button to delete the current path that appears in the text box. Cleared—The linker uses the default start-up file.

Table 3.10 Linker Panel Options (*continued*)

Option	Explanation
Dead Code Stripping	Checked—The linker strips both unreferenced source code and unreferenced data from your program. Enabling this option reduces the memory footprint of a program. Cleared—The linker preserves both unreferenced source code and unreferenced data in your program.
Use Re-entrant Runtime Libraries	Checked—The linker uses the correct thread-safe libraries and start-up code for your target architecture. If checked, the IDE passes `-reentrant` to the scc shell. Cleared—The linker uses default libraries and start-up code.
Use Temp File For Object Files	Checked—The IDE passes object filenames to the linker in a temporary file. Use this option on Windows®-hosted computers to avoid exceeding the maximum command-line length imposed by the operating system. Cleared—The IDE passes object filenames directly to the linker, instead of through a temporary file.
Additional Options	Enter additional linker command-line options. The IDE passes these options to the scc shell during the link phase. Note that the IDE passes command-line options to the scc shell exactly as you enter them in the Additional Options text box.

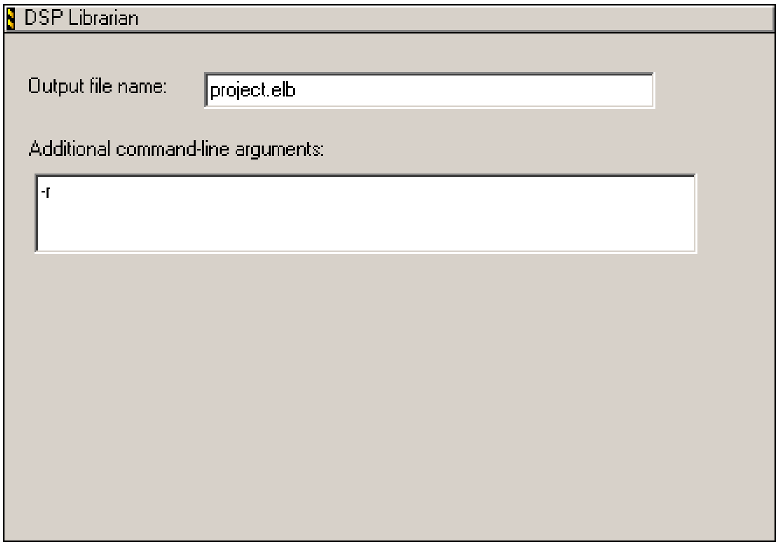
DSP Librarian

Use this panel to specify the name of the library the build target uses and to pass command-line options to the archiver utility. To view this panel in the Target Settings window, you must go to the **Target Settings** panel and use the **Linker** list box to specify **DSP Librarian**.

NOTE Select the DSP Librarian to have the tools invoke the archiver utility each time you make the build target.

Figure 3.13 shows the **DSP Librarian** target settings panel. Table 3.11 explains each panel option.

Figure 3.13 DSP Librarian Target Settings Panel



Output file name: project.elb

Additional command-line arguments: -r

Table 3.11 DSP Librarian Panel Options

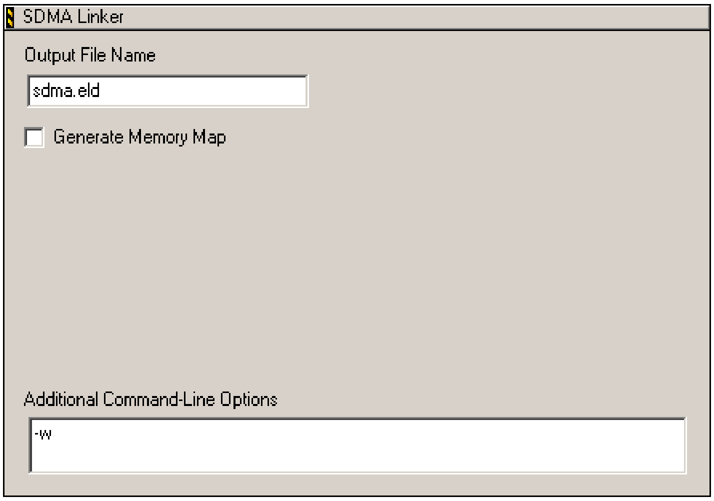
Option	Explanation
Output file name	Enter the filename of the library that the build target uses. A library filename has the `.elb` extension.
Additional command-line arguments	Enter additional command-line arguments that the IDE passes to the archiver utility. See SC100 Archiver Utility for a list of archiver command-line options.

SDMA Linker

Use this panel to select options that control the behavior of the SDMA Linker. To view this panel in the Target Settings window, you must go to the **Target Settings** panel and use the **Linker** list box to specify **SDMA Linker**.

Figure 3.12 shows the **SDMA Linker** target settings panel. Table 3.10 explains each panel option.

Figure 3.14 SDMA Linker Target Settings Panel


SDMA Linker

Output File Name: sdma.eld

☐ Generate Memory Map

Additional Command-Line Options: -w

Table 3.12 SDMA Linker Panel Options

Option	Explanation
Output File Name	Enter the name the linker gives its generated file. This filename must have the `.eld` extension.
Generate Memory Map	Checked—The SDMA linker generate a memory-map file named `bridge.map`. Checking this checkbox is equivalent to using the linker command-line option `-m`. Cleared—disable generating a memory-map file.
Additional Command-Line Options	Enter additional linker command-line options that the IDE passes to the SDMA linker during the link phase. Note that the IDE passes command-line options to the SDMA linker exactly as you enter them in the text box.

Other Executables

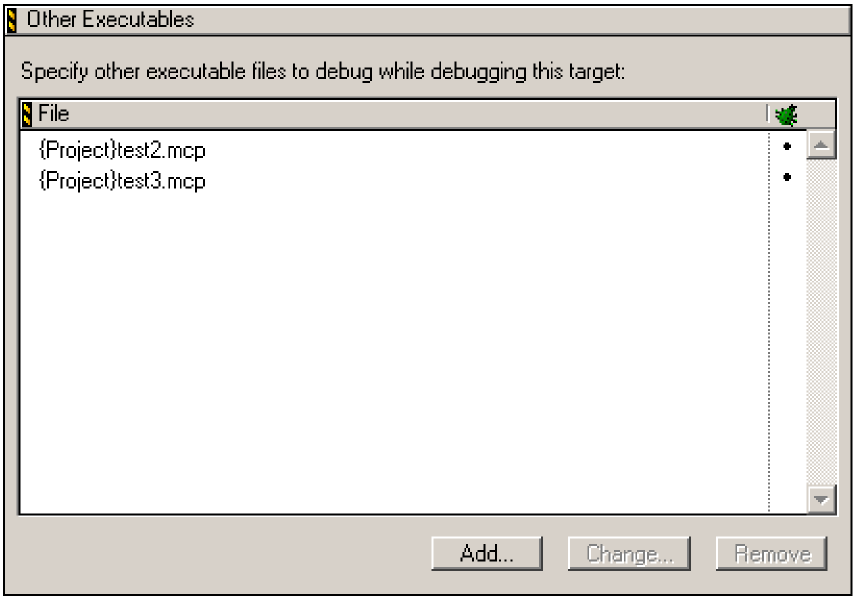
Use this panel to list other projects and files for the debugger to use in addition to the executable that the current build target generates. Also, use this panel to specify additional ELF files to debug together with the current build target’s ELF file.

NOTE To use the multi-core debugging feature, in the Other Executables panel of the primary CodeWarrior project, add the path to the CodeWarrior *project* for each core that you want to debug.

Figure 3.15 shows the **Other Executables** target settings panel. Table 3.13 explains each panel option.

For more information about using this panel to debug multiple ELF files, see "Debugging Multiple ELF Files" on page 142.

Figure 3.15 Other Executables Target Settings Panel



Specify other executable files to debug while debugging this target:

File	
{Project}test2.mcp	•
{Project}test3.mcp	•

Table 3.13 Other Executables Panel Options

Option	Explanation
File list	Shows files and projects that the debugger uses during each debug session. Click in the Debug column (icon shown at right) for a particular file or project to toggle debugging on or off for that file or project. A dot indicates the debugger will debug the file or project.
Add	Click to open the **Debug Additional Executable** dialog box. Use this dialog box to specify the path to a file or project for the debugger to use in addition to the executable file that the primary build target generates. Click the **OK** button in the dialog box to add the specified file or project to the File list.
Change	Click to open the **Debug Additional Executable** dialog box. The fields of the dialog box show the current settings for the entry currently selected in the File list. Change this information as needed, then click the **OK** button to update the information for the entry.
Remove	Click to remove the entry currently selected in the File list.

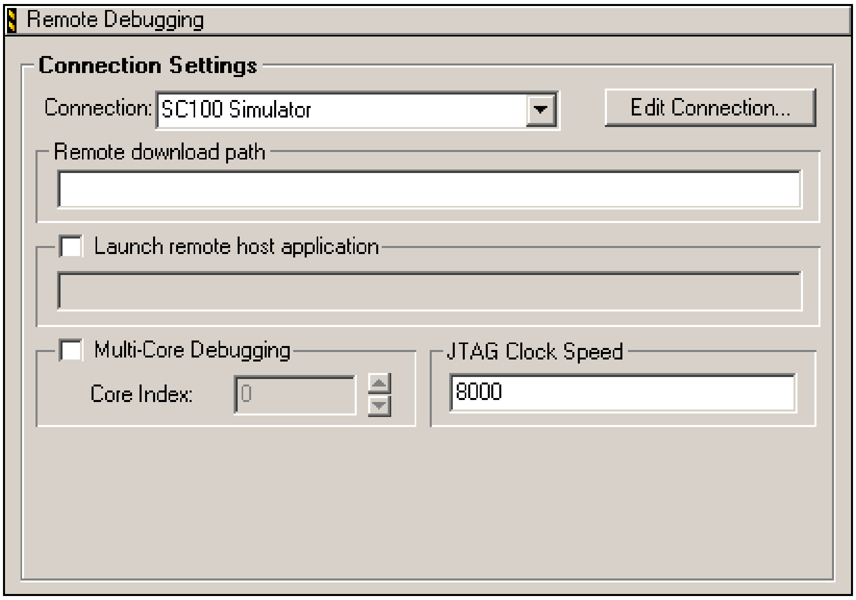
Remote Debugging

Use this panel to select and configure the connection the CodeWarrior debugger uses to communicate with your target device or simulator.

NOTE You define a remote connection in the **Remote Connections** preference panel. You use the **Remote Debugging** target settings panel to assign a remote connection to a build target and to configure this connection. See the *IDE User's Guide* for documentation of the **Remote Connections** preference panel.

Figure 3.16 shows the **Remote Debugging** target settings panel. Table 3.14 explains each panel option.

Figure 3.16 Remote Debugging Target Settings Panel



Connection: SC100 Simulator

Core Index: 0

8000

Table 3.14 Remote Debugging Panel Options

Option	Explanation
Connection	Use this list box to specify the remote connection for the current build target.
Edit Connection	Click, then use the resulting dialog box to change the configuration of the selected remote connection. Each remote connection has just one definition; as a result, if you change a remote connection's definition in this panel, you change it *everywhere* the connection gets used.
Remote download path	Enter the path of a host-computer directory to which a program running on the target hardware can read and write files.
Launch remote host application	Checked—The IDE configures the current build target to launch a host application on the target hardware at the start of each debug session. Checking this checkbox enables a related text box. Enter the path to the host application that the build target uses. Cleared—The IDE configures the current build target to not launch a host application on the target hardware.

Table 3.14 Remote Debugging Panel Options (*continued*)

Option	Explanation
Multi-Core Debugging	Checked—The IDE configures the current build target for multi-core debugging. Checking this checkbox enables the Core Index spin box. Cleared—The IDE configures the current build target to not use multi-core debugging.
Core Index	Use this spin box to specify the index of the core on which the IDE loads the build target's binary file. The debugger interacts with the core whose index you specify. For most debuggers, 0 is the index of the first core, 1 is the index of the second core, and so on.
JTAG Clock Speed	Enter the clock speed (in kilohertz) of the connection between your computer and the JTAG header of the target hardware. The IDE uses the JTAG clock speed you specify to set the debug port clock frequency for the USBTAP or EthernetTAP.

Remote Debug Options

Use this panel to specify options related to downloading executable code to the target and options related to memory access.

NOTE Checking all checkboxes in the Program Download Options group significantly increases debugging time.

Figure 3.17 shows the **Remote Debug Options** target settings panel. Table 3.15 explains each panel option.

The **Program Download Options** specify which executable code *sections* the debugger downloads to the target, and wether the debugger should read back those sections and verify them.

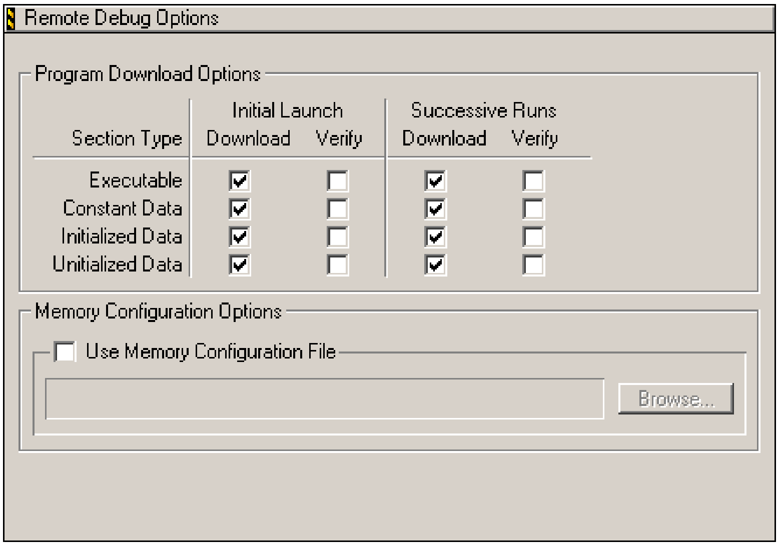
Initial Launch options apply to the first debugging session. **Successive Runs** options apply to subsequent debugging sessions (like those you initiate when you select **Debug > Restart**).

The **Download** options control whether the debugger downloads the specified **Section Type** to the target hardware. The **Verify** options control whether the debugger reads the specified **Section Type** from the target hardware and compares the read data against the generated executable code.

The **Section Type** corresponds to the section defined in the linker command file (`.lcf`), as shown in Table 3.16.

For more information about memory-configuration files, see "Memory-Configuration Files" on page 123.

Figure 3.17 Remote Debug Options Target Settings Panel



Remote Debug Options

Program Download Options

Section Type	Initial Launch Download	Initial Launch Verify	Successive Runs Download	Successive Runs Verify
Executable	✓		✓	
Constant Data	✓		✓	
Initialized Data	✓		✓	
Unitialized Data	✓		✓	

Memory Configuration Options

☐ Use Memory Configuration File

Table 3.15 Remote Debug Options Panel Options

Option	Explanation	Comments
Section Type Executable	Controls downloading and verification for executable sections. Check appropriate checkboxes to specify downloading and verifications, for initial launch and for successive runs.	Default: Both Download checkboxes checked.
Section Type Constant Data	Controls downloading and verification for constant-data sections. Check appropriate checkboxes to specify downloading and verifications, for initial launch and for successive runs.	Default: Both Download checkboxes checked.

Table 3.15 Remote Debug Options Panel Options (*continued*)

Option	Explanation	Comments
Section Type Initialized Data	Controls downloading and verification for initialized-data sections. Check appropriate checkboxes to specify downloading and verifications, for initial launch and for successive runs.	Default: Both Download checkboxes checked.
Section Type Uninitialized Data	Controls downloading and verification for uninitialized-data sections. Check appropriate checkboxes to specify downloading and verifications, for initial launch and for successive runs.	Default: Both Download checkboxes checked.
Use Memory Configuration File checkbox	Checked—The debugger uses the specified memory configuration file. Cleared—The debugger does not use a memory configuration file.	Default: Cleared. Checking this checkbox activates the **Browse** button. To specify a memory configuration file, click the **Browse** button, then use the resulting dialog box.

Table 3.16 Section Type Correspondence to Linker Command File

Section Type	Linker Command File Section Type	Comments
Executable	text	Program-code sections that have X flags in the linker-command file
Constant Data	data	Program-data sections that have neither X nor W flags in the linker-command file

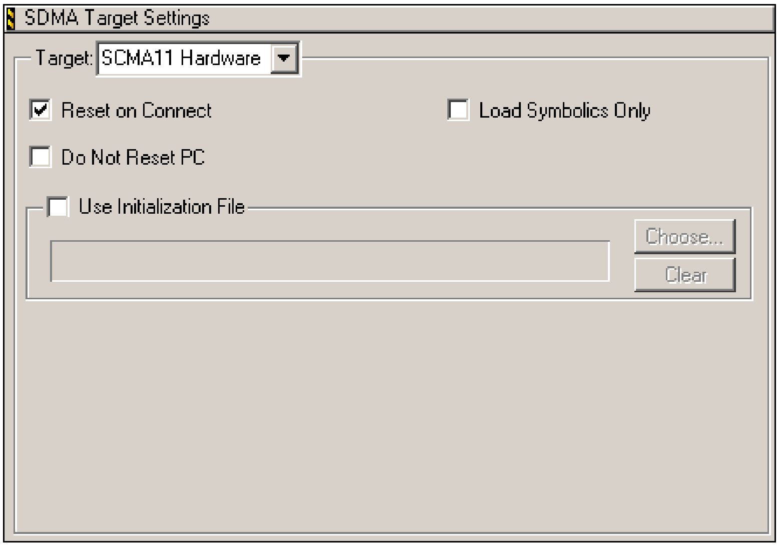
Table 3.16 Section Type Correspondence to Linker Command File (*continued*)

Section Type	Linker Command File Section Type	Comments
Initialized Data	data	Program-data sections with initial values. These sections have W flags, but not X flags, in the linker-command file.
Uninitialized Data	bss	Program-data sections without initial values. These sections have W flags, but not X flags, in the linker-command file.

SDMA Target Settings

Use this panel to select options that control the behavior of the SDMA controller. Figure 3.18 shows the **SDMA Target Settings** panel. Table 3.17 explains each panel option.

Figure 3.18 SDMA Target Settings Panel


SDMA Target Settings

Target: SCMA11 Hardware

[✓] Reset on Connect

[ ] Load Symbolics Only

[ ] Do Not Reset PC

[ ] Use Initialization File

Table 3.17 SDMA Target Settings Panel Options

Option	Explanation
Target	Use this list box to specify the target board that the debugger controls.
Reset on Connect	Checked—The debugger resets the target each time you download a program for debugging. If you use a JTAG chain, the debugger resets all boards. Cleared—The debugger downloads a program to the target without first resetting that target.
Load Symbolics Only	Checked—The debugger downloads just a program's symbolic debugging information to the target. Cleared—The debugger downloads all program code, including symbolic debugging information, to the target. Checking this checkbox is useful if: • You debug a program that is in ROM. • You repeatedly debug the same program. Under these circumstances, loading just symbolics saves time because the debugger skips the (sometimes lengthy) download of executable code that is already on the target.
Do Not Reset PC	Checked—The debugger preserves the program-counter value when you restart a debugging session. Cleared—The debugger resets the program-counter value when you restart a debugging session.
Use Initialization File	Checked—The debugger uses an initialization file at the start of each debugging session. To specify the initialization file, click the **Choose** button, then use the resulting dialog box. To remove the currently specified file, click the **Clear** button. Cleared—The debugger starts a debugging session without using an initialization file. An *initialization file* is a text file that tells the debugger how to initialize the target after reset, just before downloading your binary file. Use initialization-file commands to write values to various registers, core registers, and memory locations.

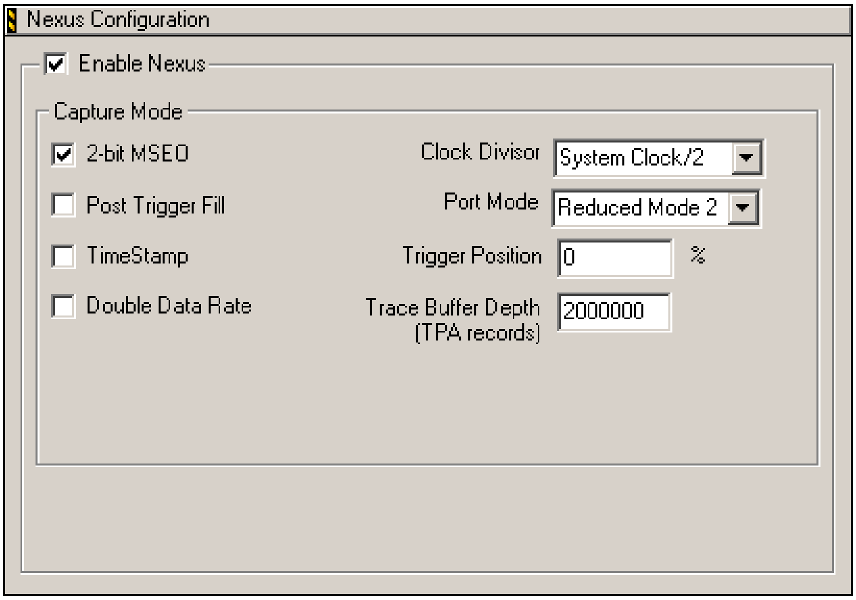
Nexus Configuration

Use this panel to configure Nexus registers and RealView trace options. To view this panel in the Target Settings window, you must go to the **Remote Debugging** panel and use the **Connection** list box to specify a RealView In-Circuit Emulator (RVI)-based remote connection.

NOTE To work with remote connections, use the Remote Connections preference panel of the IDE Preferences window. To view this panel, select **Edit > Preferences**, then select **Remote Connections** from the **IDE Preference Panels** list of the window that appears.

Figure 3.19 shows the **Nexus Configuration** panel. Table 3.17 explains each panel option.

Figure 3.19 Nexus Configuration



Nexus Configuration

[✓] Enable Nexus

Capture Mode

[✓] 2-bit MSEO
[ ] Post Trigger Fill
[ ] TimeStamp
[ ] Double Data Rate

Clock Divisor: System Clock/2
Port Mode: Reduced Mode 2
Trigger Position: 0 %
Trace Buffer Depth (TPA records): 2000000

Table 3.18 Nexus Configuration Panel Options

Option	Explanation
Enable Nexus	Checked—The debugger enables the Nexus hardware module. Cleared—The debugger disables the Nexus hardware module.
2-bit MSEO	Checked—The debugger configures the Message Start/End Control Indicator (MSEO) to use 2 pins (and act as a 2-bit state machine). Cleared—The debugger configures the MSEO to use 1 pin. You can improve bandwidth by having the MESO use 2 pins, which allows faster message writes to the Nexus trace buffer.
Clock Divisor	Use this list box to specify the frequency at which to send the Nexus hardware module's data output to trace hardware. For each of these settings, the **System Clock** is the SC140 processor clock: • **System Clock**—Send data at that same frequency as the system clock • **System Clock/2**—Send data at 50% the frequency of the system clock • **System Clock/4**—Send data at 25% the frequency of the system clock • **System Clock/8**—Send data at 12.5% the frequency of the system clock
Post Trigger Fill	Checked—The Nexus hardware module uses a specified trigger as a signal to collect trace information. Use the Trigger Position text box to specify the percentage of the trace hardware's buffer size that collected trace information must fill before collection activity stops. The trace buffer continues to fill past the Trigger Position until becoming full. Cleared—The Nexus hardware module does not use a specified trigger as a signal to collect trace information. Note that this type of trigger does not turn trace collection on or off. Instead, this trigger signals whether to collect trace information.

Table 3.18 Nexus Configuration Panel Options (*continued*)

Option	Explanation
Port Mode	Use this list box to specify the port mode of the Nexus interface: • **Reduced Mode 1**—Set the message-port width to 8 bits • **Reduced Mode 2**—Set the message-port width to 16 bits The ARM RealView Trace hardware supports up to a 16-bit trace interface.
TimeStamp	Checked—The debugger enables timestamp input to the Nexus hardware module. Cleared—The debugger disables timestamp input to the Nexus hardware module.
Trigger Position	Enter the percentage of the trace hardware’s buffer size that must fill before trace collection stops. For example, entering `50` means that collected trace information must fill half (50%) of the trace hardware’s buffer size before collection stops. If you check the Post Trigger Fill checkbox, the debugger captures trace before and after a trigger point.
Double Data Rate	Checked—The Message Clock Out (MCKO) of the Nexus hardware module runs at half the normal rate. The trace-capture tool samples the auxiliary outputs on the rising and falling edges of MCKO. Message Data Out (MDO) captures on both the rising and falling edges of MCKO. Cleared—The MCKO of the Nexus hardware module runs at the normal rate. MDO captures on just the rising edge of MCKO.
Trace Buffer Depth	Enter the maximum number of Nexus messages (TPA records) to collect in the trace buffer, up to 2 million (decimal notation). Note that Nexus trace-retrieval time increases proportionally to the depth that you specify.

Nexus Data Trace Configuration

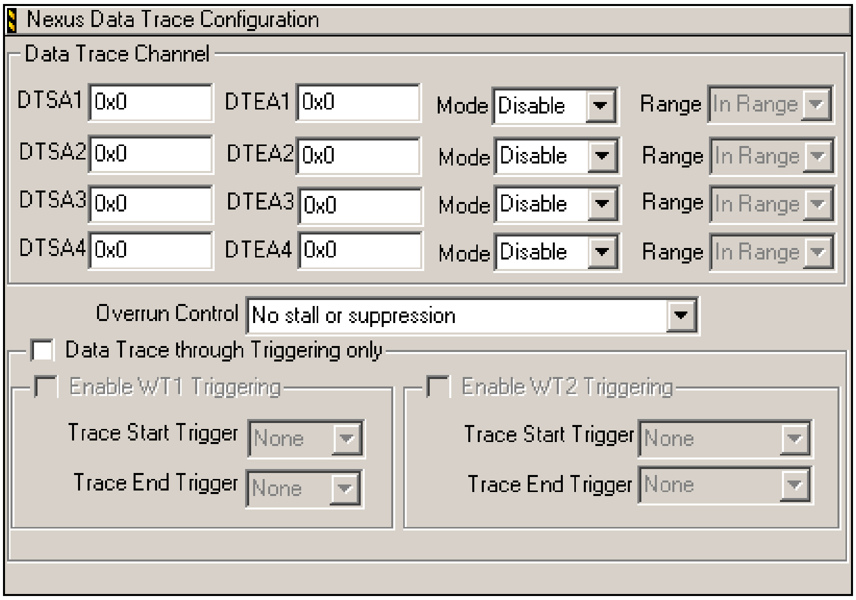
Use this panel to configure up to four data channels for collecting data trace with the Nexus interface.

For each data channel, you specify

- the starting address that the interface writes to the Data Trace Start Address (DTSA) register for that channel
- the ending address that the interface writes to the Data Trace End Address (DTEA) register for that channel
- the data-trace mode
- whether to have the debugger monitor outside or inside the specified address range

Figure 3.20 shows the **Nexus Data Trace Configuration** panel. Table 3.19 explains each panel option.

Figure 3.20 Nexus Data Trace Configuration



Nexus Data Trace Configuration

Data Trace Channel

DTSA1 0x0 DTEA1 0x0 Mode Disable Range In Range
DTSA2 0x0 DTEA2 0x0 Mode Disable Range In Range
DTSA3 0x0 DTEA3 0x0 Mode Disable Range In Range
DTSA4 0x0 DTEA4 0x0 Mode Disable Range In Range

Overrun Control No stall or suppression

☐ Data Trace through Triggering only

☐ Enable WT1 Triggering
Trace Start Trigger None
Trace End Trigger None

☐ Enable WT2 Triggering
Trace Start Trigger None
Trace End Trigger None

Table 3.19 Nexus Data Trace Configuration Panel Options

Option	Explanation
DTSA*n*	Enter the starting address, in hexadecimal notation, of the *n*th data channel. You can enter up to four starting addresses. For example, **DTSA3** is the starting address of the third data channel. Use the corresponding DTEA register to define the ending address. For example, use **DTEA3** to define the ending address for **DTSA3**.
DTEA*n*	Enter the ending address, in hexadecimal notation, of the *n*th data channel. You can enter up to four ending addresses. For example, **DTEA3** is the ending address of the third data channel. Use the corresponding DTSA register to define the starting address. For example, use **DTSA3** to define the starting address for **DTEA3**.
Mode	Use this list box to specify when the Nexus interface generates trace messages: • **Disable**—Do not generate trace messages. This is the default setting. • **Read**—Generate only data-read trace messages. • **Write**—Generate only data-write trace messages. • **Read & Write**—Generate both data-read and data-write trace messages. Each **Mode** setting applies to the corresponding data channel.
Range	To use this list box, you must use the corresponding **Mode** list box to specify a setting other than **Disable**. Use this list box to specify whether to have the debugger monitor inside or outside the specified address range: • **In Range**—Specifies that the debugger monitor the range of addresses from DTSAn to DTEAn. • **Outside Range**—Specifies that the debugger monitor all addresses *except* the range from DTSAn to DTEAn. Each **Range** setting applies to the corresponding data channel.

Table 3.19 Nexus Data Trace Configuration Panel Options (*continued*)

Option	Explanation
Overrun Control	Use this list box to specify how the target hardware handles possible message-queue overflows, depending on the capacity of internal memory arrays: • **No stall or suppression**—The hardware ignores message-queue overflows. • **Stall CPU when 1/4th full** —The hardware stalls or suppresses the CPU when any array becomes 25% full. • **Stall CPU when 1/2 full** —The hardware stalls or suppresses the CPU when any array becomes 50% full. • **Stall CPU when 3/4th full** —The hardware stalls or suppresses the CPU when any array becomes 75% full. • **Suppress Data Trace when 1/4th full** —The hardware stalls or suppresses collecting data trace when any array becomes 25% full. • **Suppress Data Trace when 1/2 full** —The hardware stalls or suppresses collecting data trace when any array becomes 50% full. • **Suppress Data Trace when 3/4th full** —The hardware stalls or suppresses collecting data trace when any array becomes 75% full.
Data Trace through Triggering Only	Checked—The debugger collects data trace only for triggering on Watchpoint Trigger Register 1 (WT1), Watchpoint Trigger Register 2 (WT2), or both. Checking this checkbox enables the **Enable WT1 Triggering** and **Enable WT2 Triggering** checkboxes. Cleared—The debugger collects data trace, regardless of triggering on WT1 or WT2.
Enable WT1 Triggering	To use this checkbox, you must check the **Data Trace through Triggering Only** checkbox. Checked—The interface collects data trace according to triggering on WT1. Checking this checkbox enables the corresponding **Trace Start Trigger** and **Trace End Trigger** list boxes. Cleared—The interface collects data trace, regardless of triggering on WT1. WT1 lets you specify internal SC140 Nexus module watchpoints that trigger data-trace start and stop events.

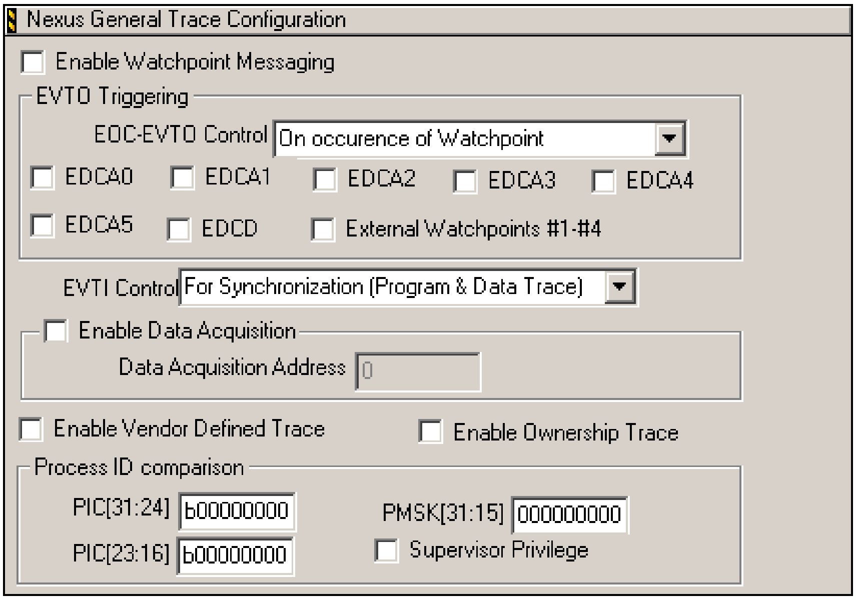
Table 3.19 Nexus Data Trace Configuration Panel Options (*continued*)

Option	Explanation
Enable WT2 Triggering	To use this checkbox, you must check the **Data Trace through Triggering Only** checkbox. Checked—The interface collects data trace according to triggering on WT2. Checking this checkbox enables the corresponding **Trace Start Trigger** and **Trace End Trigger** list boxes. Cleared—The interface collects data trace, regardless of triggering on WT2. WT2 lets you specify external-module watchpoints that trigger data-trace start and stop events.
Trace Start Trigger	To use this list box, you must check the corresponding **Enable WT1 Triggering** or **Enable WT2 Triggering** checkbox. Use the **Trace Start Trigger** list box to specify the watchpoint on which to start data trace.
Trace End Trigger	To use this list box, you must check the corresponding **Enable WT1 Triggering** or **Enable WT2 Triggering** checkbox. Use the **Trace End Trigger** list box to specify the watchpoint on which to stop data trace.

Nexus General Trace Configuration

Use this panel to configure watchpoint messaging and general trace options. Figure 3.21 shows the **Nexus General Trace Configuration** panel. Table 3.20 explains each panel option.

Figure 3.21 Nexus General Trace Configuration


Nexus General Trace Configuration

☐ Enable Watchpoint Messaging

EVTO Triggering

EOC-EVTO Control: On occurence of Watchpoint

☐ EDCA0 ☐ EDCA1 ☐ EDCA2 ☐ EDCA3 ☐ EDCA4

☐ EDCA5 ☐ EDCD ☐ External Watchpoints #1-#4

EVTI Control: For Synchronization (Program & Data Trace)

☐ Enable Data Acquisition

Data Acquisition Address: 0

☐ Enable Vendor Defined Trace ☐ Enable Ownership Trace

Process ID comparison

PIC[31:24]: b00000000 PMSK[31:15]: 000000000

PIC[23:16]: b00000000 ☐ Supervisor Privilege

Table 3.20 Nexus General Trace Configuration Panel Options

Option	Explanation
Enable Watchpoint Messaging	Checked—The Nexus interface generates watchpoint messages. Checking this checkbox sets the watchpoint-messaging enable bit in the DC1 register. Cleared—The Nexus interface does not generate watchpoint messages.
EOC-EVTO Control	Use this list box to specify the event on which the Event Out (EVTO) asserts: • **On occurrence of Watchpoint**—The EVTO asserts on the watchpoints you specify. Check the EDCAn, EDCD, and External Watchpoints #1-#4 checkboxes for the watchpoints on which you want the EVTO to assert. • **On entry into system-level Debug mode**—The EVTO asserts on entry into the system-level debugging mode.

Table 3.20 Nexus General Trace Configuration Panel Options (*continued*)

Option	Explanation
EDCA*n*	To use these checkboxes, you must use the **EOC-EVTO Control** list box to specify **On occurrence of Watchpoint**. Checked—Have the EVTO assert on the *n*th EDCA register. Cleared—Do not have the EVTO assert on the *n*th EDCA register.
EDCD	To use this checkbox, you must use the **EOC-EVTO Control** list box to specify **On occurrence of Watchpoint**. Checked—Have the EVTO assert on the EDCD register. Cleared—Do not have the EVTO assert on the EDCD register.
External Watchpoints #1-#4	To use this checkbox, you must use the **EOC-EVTO Control** list box to specify **On occurrence of Watchpoint**. Checked—Have the EVTO assert on the first through fourth external watchpoints. Cleared—Do not have the EVTO assert on the first through fourth external watchpoints.
EVTI Control	Use this list box to specify the actions that occur on asserting Event In (EVTI): • **For Synchronization (Program & Data Trace)**—Synchronization of program and data trace occurs on asserting EVTI. • **Assert debug request to processor**—The processor receives a debugging request on asserting EVTI. • **Disabled**—No events occur on asserting EVTI.
Enable Data Acquisition	Checked—The Nexus interface generates data-acquisition messages. Checking this checkbox enables the **Data Acquisition Address** text box. Each write access to the address you specify in this box causes the Nexus interface to generate a data-acquisition message. Cleared—The Nexus interface does not generate data-acquisition messages.

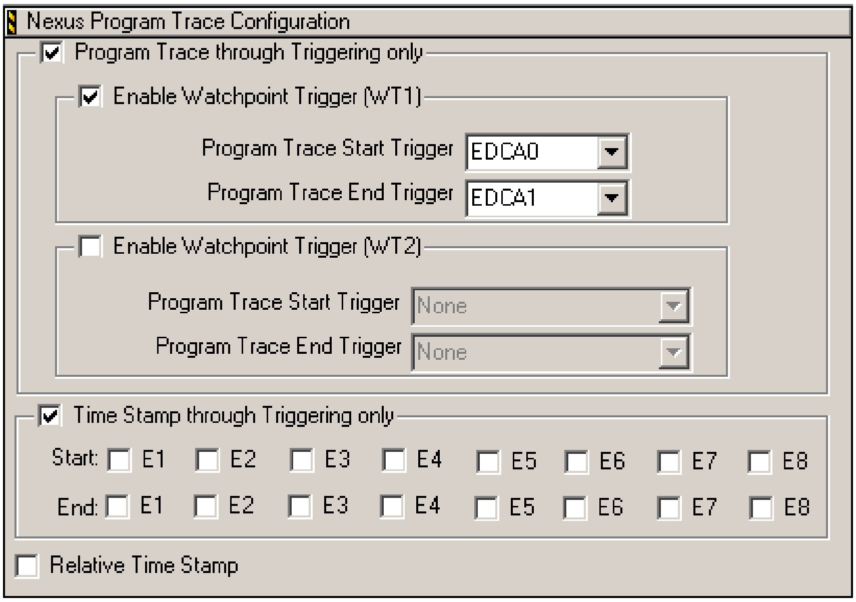
Table 3.20 Nexus General Trace Configuration Panel Options (*continued*)

Option	Explanation
Data Acquisition Address	To use this text box, you must check the **Enable Data Acquisition** checkbox. Enter the address to which a write access causes the Nexus interface to generate a data-acquisition message. The interface uses the address you specify to configure the User base address.
Enable Vendor Defined Trace	Checked—The Nexus interface uses vendor-defined messaging through DC1. Cleared—The Nexus interface dot not use vendor-defined messaging through DC1.
Enable Ownership Trace	Checked—The Nexus interface uses ownership trace. In this type of trace, the interface compares the PIC register value with the current Process ID value of Ownership messages to generate triggers on a Process ID match. Cleared—The Nexus interface does not use ownership trace. To specify triggers, you use the **Enable WT1 Triggering** and **Enable WT2 Triggering** options of the Nexus Data Trace Configuration settings panel, and the **Enable Watchpoint Trigger (WT1)** and **Enable Watchpoint Trigger (WT2)** options of the Nexus Program Trace Configuration settings panel.
PMSK[31:15]	Enter the mask that defines the bits in the PIC register that the Nexus interface uses for ownership trace.
Supervisor Privilege	Checked—The Nexus interface has supervisor-level access privileges for the specified register bits. Cleared—The Nexus interface has user-level access privileges for the specified register bits.

Nexus Program Trace Configuration

Use this panel to configure program-trace triggering and time stamps. Figure 3.22 shows the **Nexus Program Trace Configuration** panel. Table 3.21 explains each panel option.

Figure 3.22 Nexus Program Trace Configuration



Nexus Program Trace Configuration

[✓] Program Trace through Triggering only

[✓] Enable Watchpoint Trigger (WT1)

Program Trace Start Trigger: EDCA0

Program Trace End Trigger: EDCA1

[ ] Enable Watchpoint Trigger (WT2)

Program Trace Start Trigger: None

Program Trace End Trigger: None

[✓] Time Stamp through Triggering only

Start: [ ] E1 [ ] E2 [ ] E3 [ ] E4 [ ] E5 [ ] E6 [ ] E7 [ ] E8

End: [ ] E1 [ ] E2 [ ] E3 [ ] E4 [ ] E5 [ ] E6 [ ] E7 [ ] E8

[ ] Relative Time Stamp

Table 3.21 Nexus Program Trace Configuration Panel Options

Option	Explanation
Program Trace through Triggering only	Checked—The Nexus interface always generates program trace. Checking this checkbox sets the Trace Message (TM) field of the DC1 register. Enabling program trace this way causes the interface to override all trigger configurations. Cleared—The Nexus interface generates program trace only when triggered on. If you want to configure program-trace start and stop through watchpoint triggering, you must clear this checkbox.
Enable Watchpoint Trigger (WT1)	To use this checkbox, you must check the **Program Trace through Triggering only** checkbox. Checked—The interface collects program trace according to triggering on WT1. Checking this checkbox enables the corresponding **Program Trace Start Trigger** and **Program Trace End Trigger** list boxes. Cleared—The interface collects program trace, regardless of triggering on WT1. WT1 lets you specify internal SC140 Nexus module watchpoints that trigger program-trace start and stop events.
Enable Watchpoint Trigger (WT2)	To use this checkbox, you must check the **Program Trace through Triggering only** checkbox. Checked—The interface collects program trace according to triggering on WT2. Checking this checkbox enables the corresponding **Program Trace Start Trigger** and **Program Trace End Trigger** list boxes. Cleared—The interface collects program trace, regardless of triggering on WT2. WT2 lets you specify external-module watchpoints that trigger program-trace start and stop events.
Program Trace Start Trigger	To use this list box, you must check the corresponding **Enable Watchpoint Trigger (WT1)** or **Enable Watchpoint Trigger (WT2)** checkbox. Use the **Program Trace Start Trigger** list box to specify the watchpoint on which to start program trace.

Table 3.21 Nexus Program Trace Configuration Panel Options (*continued*)

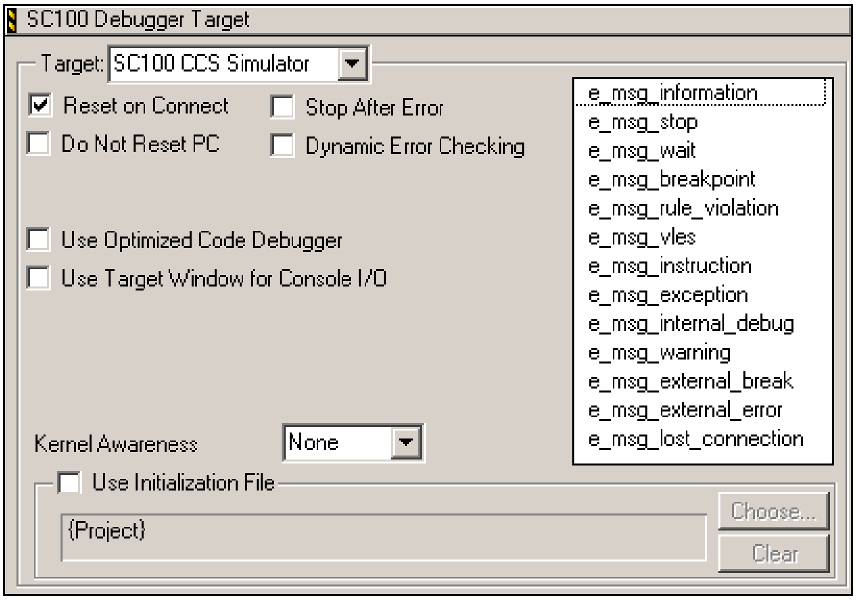
Option	Explanation
Program Trace End Trigger	To use this list box, you must check the corresponding **Enable Watchpoint Trigger (WT1)** or **Enable Watchpoint Trigger (WT2)** checkbox. Use the **Program Trace End Trigger** list box to specify the watchpoint on which to stop program trace.
Time Stamp through Triggering only	Checked—The interface generates timestamps for Embedded Cross Trigger (ECT) events. Check the **En** checkboxes for the ECT events on which you want to start and stop timestamp generation. Cleared—The interface always generates timestamps, regardless of ECT events. The Timestamp Control (TSC) register configures the timestamps according to the checkboxes you check.
E*n*	To use these checkboxes, you must check the **Time Stamp through Triggering only** checkbox. Checked—Use the *n*th ECT event to trigger a timestamp action. Cleared—Do not use the *n*th ECT event to trigger a timestamp action. Checkboxes in the **Start** row control start-timestamp actions. Checkboxes in the **End** row control end-timestamp actions. The PSC configures these actions.
Relative Time Stamp	Checked—The interface generates relative timestamps. Cleared—The interface generates absolute timestamps.

SC100 Debugger Target

Use this panel to select the simulator or device on which you will debug the binary file that the build target produces. Also, use the panel to control how the CodeWarrior debugger interacts with the selected device. Also, use the panel to control how the debugger behaves at startup and during a debug session.

Figure 3.23 shows the **SC100 Debugger Target** panel. Table 3.22 explains each panel option.

Figure 3.23 SC100 Debugger Target Settings Panel


SC100 Debugger Target

Target: SC100 CCS Simulator

- [✓] Reset on Connect
- [ ] Stop After Error
- [ ] Do Not Reset PC
- [ ] Dynamic Error Checking
- [ ] Use Optimized Code Debugger
- [ ] Use Target Window for Console I/O

e_msg_information
e_msg_stop
e_msg_wait
e_msg_breakpoint
e_msg_rule_violation
e_msg_vles
e_msg_instruction
e_msg_exception
e_msg_internal_debug
e_msg_warning
e_msg_external_break
e_msg_external_error
e_msg_lost_connection

Kernel Awareness: None

- [ ] Use Initialization File

{Project}

Table 3.22 SC100 Debugger Target Panel Options

Option	Explanation
Target	Use this list box to select the simulator or target hardware on which you will debug the binary file that the build target produces.
Simulator Options	This list appears on the right side of the panel after you use the Target list box to specify **SC100 CCS Simulator** or **SC1000 LLC Simulator**. Otherwise, the list does not appear. Click an item in the list to specify the message type that you want the specified Target to generate. To select more than one message type, Control-click each one. To deselect a message type, Control-click it. A plus sign (+) appears next to each message type that the specified Target will generate.

Table 3.22 SC100 Debugger Target Panel Options (*continued*)

Option	Explanation
Reset on Connect	Checked—The debugger resets the target each time you download a program for debugging. If you use a JTAG chain, the debugger resets all boards. Cleared—The debugger downloads a program to the target without first resetting that target.
Stop After Error	Checked—The debugger stops after generating the first error message. Cleared—The debugger continues after generating an error message.
Do Not Reset PC	Checked—The debugger preserves the program-counter value when you restart a debugging session. Cleared—The debugger resets the program-counter value when you restart a debugging session.
Dynamic Error Checking	Checked—The debugger checks for *dynamic errors*. These runtime errors are violations of the StarCore architecture programming rules. Cleared—The debugger does not check for dynamic errors.
Use Target Window for Console I/O	Checked—The debugger displays target output in a separate console window. Cleared—The debugger displays all output in the same console window.
Kernel Awareness	Use this list box to specify the real-time operating system (RTOS) you use. The debugger can display kernel objects for the selected RTOS. If you do not use an RTOS, use this list box to specify **None**.
Use Initialization File	Checked—The debugger uses an initialization file at the start of each debugging session. To specify the initialization file, click the **Choose** button, then use the resulting dialog box. To remove the currently specified file, click the **Clear** button. Cleared—The debugger starts a debugging session without using an initialization file. An *initialization file* is a text file that tells the debugger how to initialize the target after reset, just before downloading your binary file. Use initialization-file commands to write values to various registers, core registers, and memory locations.

Source Folder Mapping

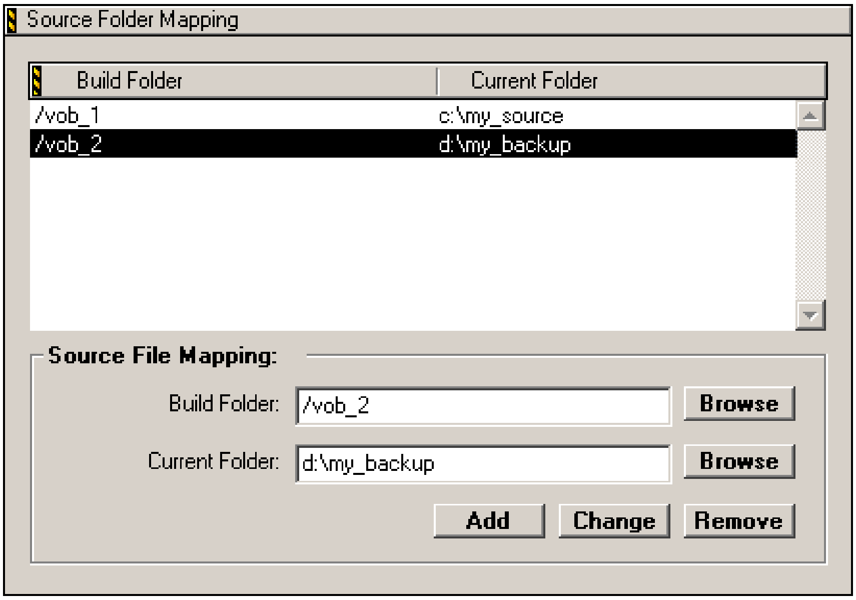
Use this panel when you debug an executable file that is built in one place, but debugged from another. The mapping information you specify lets the CodeWarrior debugger find and display your source-code files, even though they are not in the locations specified in the executable file's debug information.

NOTE If you create a CodeWarrior project by opening an ELF file in the IDE, the IDE uses the debug information in this file to add to the new project the source files used to build the file.

If the IDE cannot find a particular source file, a dialog box appears. Use this dialog box to select the missing file. The IDE uses the current-location information together with the debug information in the ELF file to create entries in the **Source Folder Mapping** panel.

Figure 3.24 shows the **Source Folder Mapping** target settings panel. Table 3.23 explains each panel option.

Figure 3.24 Source Folder Mapping Target Settings Panel


Source Folder Mapping

Build Folder	Current Folder
/vob_1	c:\my_source
/vob_2	d:\my_backup

Source File Mapping:

Build Folder: /vob_2

Current Folder: d:\my_backup

Table 3.23 Source Folder Mapping Panel Options

Option	Explanation

Build Folder	Enter the path that contained the executable file's source files at the time the executable file was built. Alternatively, click the **Browse** button, then use the resulting dialog box. The specified path can be the root of a source-code tree.
	
	For example, if your source-code files were in the directories
	
	`/vob/my_project/headers`
	`/vob/my_project/source`
	
	you can enter `/vob/my_project` in the **Build Folder** text box.
	
	If the debugger cannot find a file referenced in the executable file's debug information, the debugger replaces the string `/vob/my_project` in the missing file's name with the associated **Current Folder** string and tries again. The debugger repeats this process for each **Build Folder**/**Current Folder** pair until it finds the missing file, or no more folder pairs remain.

Current Folder	Enter the path that contains the executable file's source files now, that is, at the time of the debug session. Alternatively, click the **Browse** button, then use the resulting dialog box. The supplied path can be the root of a source-code tree.
	
	For example, if your source-code files are now in the directories
	
	`C:\my_project\headers`
	`C:\my_project\source`
	
	you can enter `C:\my_project` in the **Current Folder** text box.
	
	If the debugger cannot find a file referenced in the executable file's debug information, the debugger replaces the **Build Folder** string in the missing file's name with the string `C:\my_project` and tries again. The debugger repeats this process for each **Build Folder**/**Current Folder** pair until it finds the missing file, or no more folder pairs remain.

Add	Click to add the current **Build Folder**/**Current Folder** pair as a new entry to the Source Folder Mapping list.

Table 3.23 Source Folder Mapping Panel Options (*continued*)

Option	Explanation
Change	Click to change the **Build Folder**/**Current Folder** pair currently selected in the Source Folder Mapping list.
Remove	Click to delete the **Build Folder**/**Current Folder** pair currently selected in the Source Folder Mapping list.

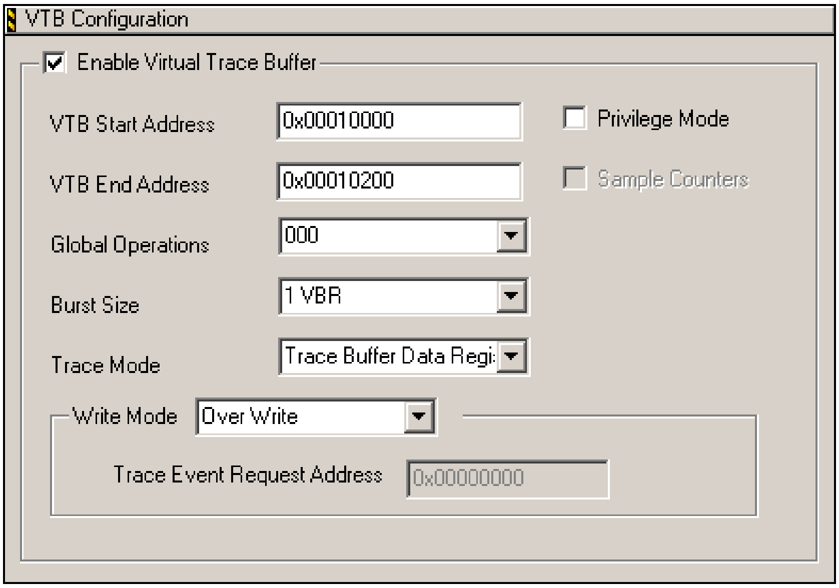
VTB Configuration

Use this panel to configure Debugging and Profiling Unit (DPU) trace registers for use with the CodeWarrior Virtual Trace Buffer (VTB). The StarCore SC140e Platform 2002 architecture supports real-time tracing to the main memory (either external memory or M2 memory) with the Trace Write Buffer. You can use the VTB to view the Trace Write Buffer's contents in the IDE.

NOTE See "Set Up the VTB to Use an On-host Profiler" on page 287 for more information about on-host profiling and the VTB.

Figure 3.25 shows the VTB Configuration panel. Table 3.24 explains each panel option.

Figure 3.25 VTB Configuration


VTB Configuration

[✓] Enable Virtual Trace Buffer

VTB Start Address: 0x00010000

VTB End Address: 0x00010200

Global Operations: 000

Burst Size: 1 VBR

Trace Mode: Trace Buffer Data Regi:

Write Mode: Over Write

Trace Event Request Address: 0x00000000

[ ] Privilege Mode

[ ] Sample Counters

Table 3.24 VTB Configuration Panel Options

Option	Explanation
Enable Virtual Trace Buffer	Checked—The IDE writes the value 1 in the EN bit (bit 0) of the DPU Trace Control register (DP_TC). Writing this value enables the VTB. Cleared—The IDE writes the value 0 in the EN bit (bit 0) of the DPU Trace Control register. Writing this value disables the VTB. Check this checkbox to enable the other options in the panel.
VTB Start Address	Enter the starting address of the VTB. The IDE writes this value in bits 5 to 31 of the DPU VTB Start Address register (DP_TSA).
VTB End Address	Enter the ending address of the VTB. The IDE writes this value in bits 5 to 31 of the DPU VTB End Address register (DP_TEA).
Global Operations	Use this list box to specify the global attribute for VTB write access. The IDE writes the appropriate value in bits 18 to 20 of the DPU Trace Control register (DP_TC).
Burst Size	Use this list box to specify the burst size for VTB write access. The IDE writes the appropriate value in bits 16 and 17 of the DPU Trace Control register (DP_TC).
Trace Mode	Use this list box to specify whether to sample the counter values to the trace buffer. The IDE writes the appropriate value in bits 1 to 4 of the DPU Trace Control register (DP_TC). Specifying **Trace Buffer Data Register** causes the IDE to store the content of the trace-buffer data register into VTB memory.
Write Mode	Use this list box to specify how the IDE writes data to the VTB. The IDE writes the appropriate value in bits 8 and 9 of the DPU Trace Control register (DP_TC).
Trace Event Request Address	The IDE enables this text box after you use the Write Mode list box to specify **Trace Event Request**. Enter the Trace Event address of the VTB. The IDE writes this value in bits 0 to 31 of the DPU Trace Event Request register (DP_TER).

Table 3.24 VTB Configuration Panel Options (*continued*)

Option	Explanation
Privilege Mode	Checked—The IDE writes the value 1 in bit 21 of the DPU Trace Control register (DP_TC). Writing this value sets the privilege attribute for VTB write access. Cleared—The IDE writes the value 0 in bit 21 of the DPU Trace Control register (DP_TC). Writing this value clears the privilege attribute for VTB write access.
Sample Counters	Checked—The IDE writes the value 1 in bit 6 of the DPU Trace Control register (DP_TC). With this option enabled, the IDE samples counter values to the trace buffer. Cleared—The IDE writes the value 0 in bit 6 of the DPU Trace Control register (DP_TC). With this option disabled, the IDE does not sample counter values to the trace buffer.

Viewing VTB Information

After you use the VTB Configuration panel to configure DPU trace registers, you can view collected trace information during a debugging session.

To view VTB information, follow these steps:

1. Select **Project** > **Debug**.

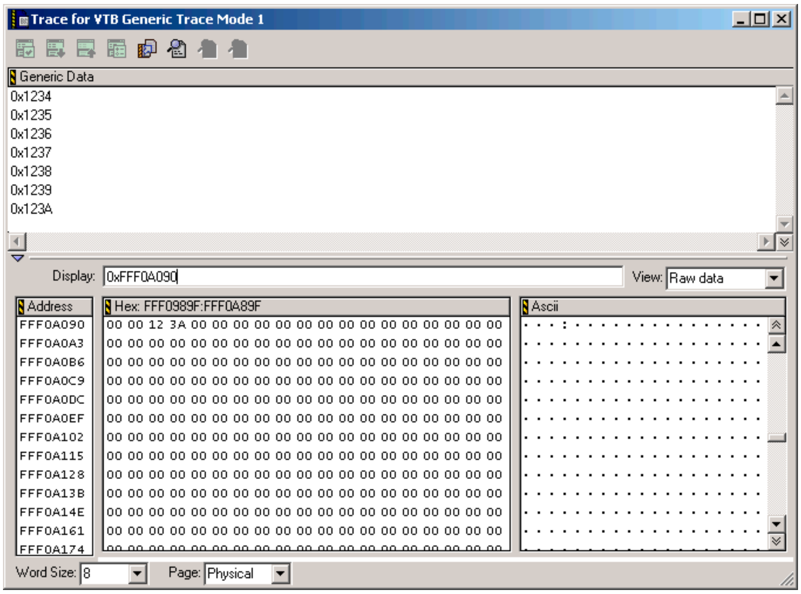
   A debugging session starts.

2. Select **Data** > **View Trace**.

The Trace Mode you specify determines the type of trace window that the IDE displays. Figure 3.26 shows the trace window for the **Trace Buffer Data Register** trace mode. Figure 3.27 shows the trace window for the **EOnCE with no compression** trace mode. Figure 3.28 shows the trace window for these trace modes:

- EOnCE with compression
- EOnCE counter
- Sample Bit Counter
- EDCA5 Event Counter

Figure 3.26 Trace Window for Trace Buffer Data Register



Trace for VTB Generic Trace Mode 1

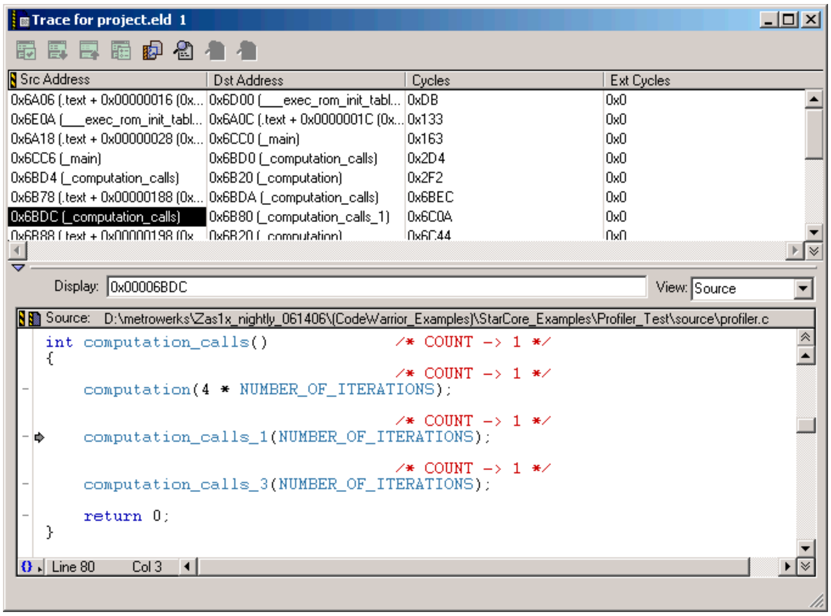
Generic Data
0x1234
0x1235
0x1236
0x1237
0x1238
0x1239
0x123A

Display: 0xFFF0A090 View: Raw data

```
Address   Hex: FFF0989F:FFF0A89F                                   Ascii
FFF0A090  00 00 12 3A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...:...............
FFF0A0A3  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
FFF0A0B6  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
FFF0A0C9  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
FFF0A0DC  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
FFF0A0EF  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
FFF0A102  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
FFF0A115  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
FFF0A128  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
FFF0A13B  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
FFF0A14E  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
FFF0A161  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
FFF0A174  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
```

Word Size: 8 Page: Physical

Figure 3.27 Trace Window for EOnCE with No Compression



Trace for project.eld 1

Src Address	Dst Address	Cycles	Ext Cycles
0x6A06 (.text + 0x00000016 (0x...	0x6D00 (___exec_rom_init_tabl...	0xDB	0x0
0x6E0A (___exec_rom_init_tabl...	0x6A0C (.text + 0x0000001C (0x...	0x133	0x0
0x6A18 (.text + 0x00000028 (0x...	0x6CC0 (_main)	0x163	0x0
0x6CC6 (_main)	0x6BD0 (_computation_calls)	0x2D4	0x0
0x6BD4 (_computation_calls)	0x6B20 (_computation)	0x2F2	0x0
0x6B78 (.text + 0x00000188 (0x...	0x6BDA (_computation_calls)	0x6BEC	0x0
0x6BDC (_computation_calls)	0x6B80 (_computation_calls_1)	0x6C0A	0x0
0x6B88 (.text + 0x00000198 (0x	0x6B20 (_computation)	0x6C44	0x0

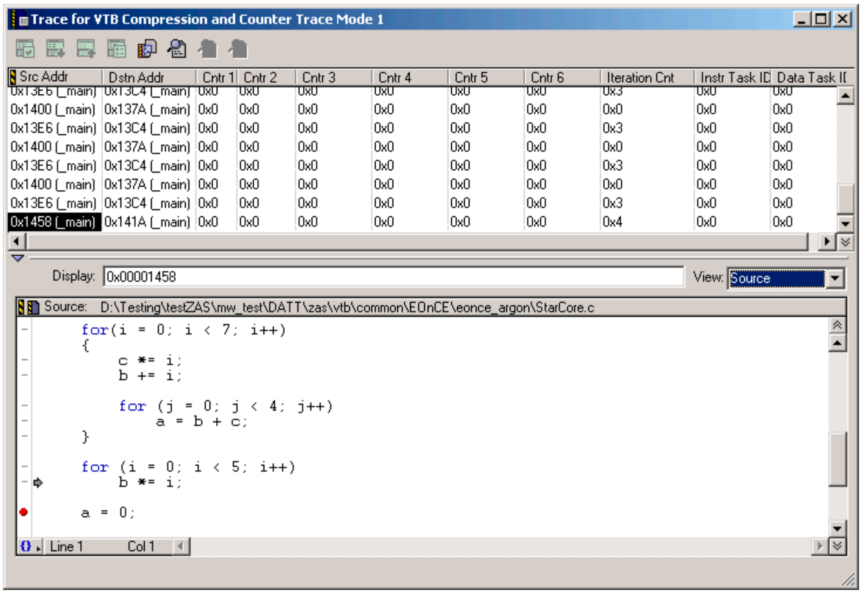
Display: 0x00006BDC View: Source

Source: D:\metrowerks\Zas1x_nightly_061406\(CodeWarrior_Examples)\StarCore_Examples\Profiler_Test\source\profiler.c

```
int computation_calls()                 /* COUNT -> 1 */
{
                                        /* COUNT -> 1 */
    computation(4 * NUMBER_OF_ITERATIONS);

                                        /* COUNT -> 1 */
    computation_calls_1(NUMBER_OF_ITERATIONS);

                                        /* COUNT -> 1 */
    computation_calls_3(NUMBER_OF_ITERATIONS);

    return 0;
}
```

Line 80 Col 3

Figure 3.28 Trace Window for Other Trace Modes


Trace for VTB Compression and Counter Trace Mode 1

Src Addr	Dstn Addr	Cntr 1	Cntr 2	Cntr 3	Cntr 4	Cntr 5	Cntr 6	Iteration Cnt	Instr Task ID	Data Task ID
0x13E6 (_main)	0x13C4 (_main)	0x0	0x0	0x0	0x0	0x0	0x0	0x3	0x0	0x0
0x1400 (_main)	0x137A (_main)	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0
0x13E6 (_main)	0x13C4 (_main)	0x0	0x0	0x0	0x0	0x0	0x0	0x3	0x0	0x0
0x1400 (_main)	0x137A (_main)	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0
0x13E6 (_main)	0x13C4 (_main)	0x0	0x0	0x0	0x0	0x0	0x0	0x3	0x0	0x0
0x1400 (_main)	0x137A (_main)	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0
0x13E6 (_main)	0x13C4 (_main)	0x0	0x0	0x0	0x0	0x0	0x0	0x3	0x0	0x0
0x1458 (_main)	0x141A (_main)	0x0	0x0	0x0	0x0	0x0	0x0	0x4	0x0	0x0

Display: 0x00001458 View: Source

Source: D:\Testing\testZAS\mw_test\DATT\zas\vtb\common\EOnCE\eonce_argon\StarCore.c

```
    for(i = 0; i < 7; i++)
    {
        c *= i;
        b += i;

        for (j = 0; j < 4; j++)
            a = b + c;
    }

    for (i = 0; i < 5; i++)
        b *= i;

    a = 0;
```

Line 1 Col 1

Assembler Preprocessors

Use this panel to specify the directories that the assembler searches for files, how the assembler handles these files, and to pass to the assembler the model number and revision of the processor you use.

Figure 3.29 shows the **Assembler Preprocessors** target settings panel. Table 3.25 explains each panel option.

Figure 3.29 Assembler Preprocessors Target Settings Panel

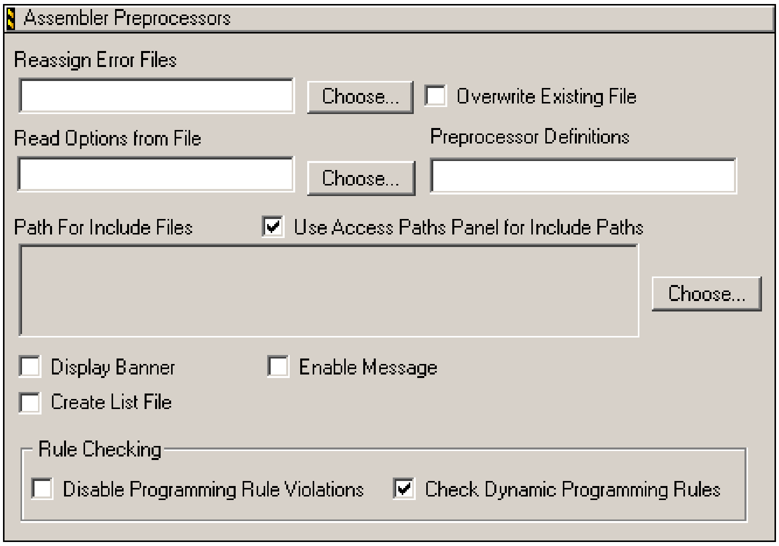


Table 3.25 Assembler Preprocessors Panel Options

Option	Explanation
Reassign Error Files	Enter the path to a file that the assembler uses in place of the default error file (`errfil`). Alternatively, click the **Choose** button, then use the resulting dialog box. The **Overwrite Existing File** checkbox controls how the assembler writes to the file that you specify in this text box.
Overwrite Existing File	Checked—The assembler overwrites the file that you specify in the **Reassign Error Files** text box. Cleared—The assembler appends information to the file that you specify in the **Reassign Error Files** text box.
Read Options from File	Enter the path to a file that specifies command-line options for assembler use. Alternatively, click the **Choose** button, then use the resulting dialog box.

Table 3.25 Assembler Preprocessors Panel Options (*continued*)

Option	Explanation
Preprocessor Definitions	Enter substitution strings that the assembler applies to all the assembly-language modules in the build target. Enter just the string portion of a substitution string. The IDE prepends the `-d` token to the text you enter. Further, separate each substitution string with a comma. For example, entering `opt1 x, opt2 y` produces this command line: `-d opt1 x -dopt2 y` Note that this option is similar to the `DEFINE` directive, but applies to *all* assembly-language modules in a build target.
Path For Include Files	This text box shows a list of search paths that the assembler uses when searching for include files. Click the **Choose** button, then use the resulting dialog box to specify a path. Click the **Choose** button multiple times to specify multiple paths. The assembler first looks for an include file in the current directory, or the directory that you specify in the `INCLUDE` directive. If the assembler does not find the file, it continues searching the paths shown in the **Path For Include Files** text box. The assembler keeps searching paths until it finds the include file or finishes searching the last path. The assembler appends to each path the string that you specify in the `INCLUDE` directive. Note that the assembler displays an error message if a header file is in a different directory from the referencing source file. Sometimes, the assembler also displays an error message if a header file is in the same directory as the referencing source file. If you see the message `Could not open source file myfile.h`, you must add the path for `myfile.h` to the **Path For Include Files** text box.
Use Access Paths Panel for Include Paths	Checked—The assembler uses the paths that you specify in the **Access Paths** target settings panel instead of the paths shown in the **Path For Include Files** text box. Cleared—The assembler uses the paths shown in the **Path For Include Files** text box.

Table 3.25 Assembler Preprocessors Panel Options (*continued*)

Option	Explanation
Display Banner	Checked—The assembler shows banner information. Cleared—The assembler hides banner information. Note that this option has no effect on hosts where the default setting is to hide the banner.
Enable Message	Checked—The assembler reports the progress of the assembly process to the standard error-output stream. For example, the assembler reports the beginning of each pass and the opening and closing of input files. You can use this information to monitor assembly process and ensure that it proceeds normally. Cleared—The assembler does not report assembly progress to the standard error-output stream.
Create List File	Checked—The assembler creates a list file named `lstfil.lst`. Cleared—The assembler does not create a list file.
Disable Programming Rule Violations	Checked—The assembler skips all checks for violations of StarCore DSP programming rules. Checking this checkbox causes the assembler to skip checks of both static rules and dynamic rules. Also, checking this checkbox causes the IDE to pass the `-u all` option to the assembler. Cleared—The assembler checks for violations of just the static StarCore DSP programming rules. Checking just for static-rule violations is the default behavior when you invoke the assembler from the command line.
Check Dynamic Programming Rules	Checked—The assembler reports violations of the just the dynamic StarCore DSP programming rules. Checking this checkbox causes the IDE to pass the `-s all` option to the assembler. Cleared—The assembler skips checks for violations of the dynamic rules. Skipping checks for dynamic-rule violations is the default behavior when you invoke the assembler from the command line.

Listing File Options

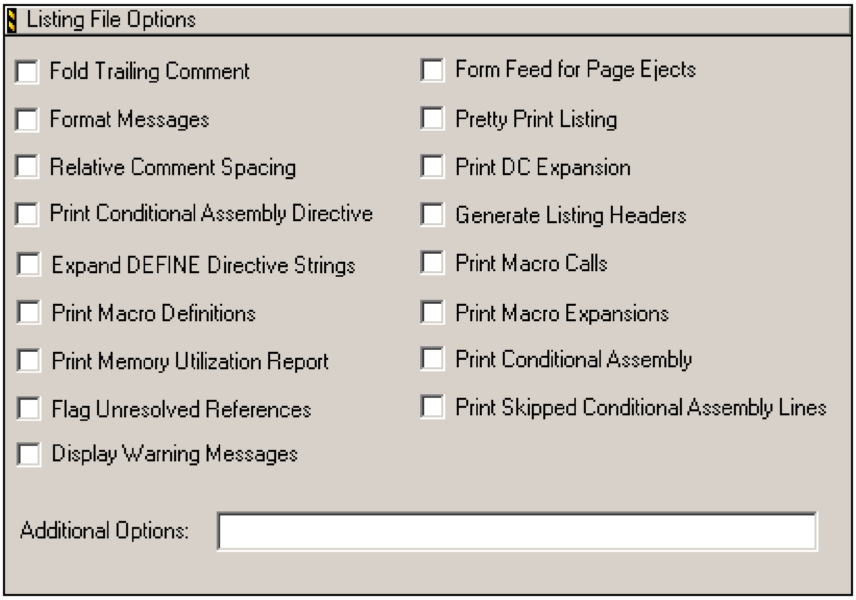
Use this panel to specify how the assembler formats the listing file it generates. Also, use this panel to pass command-line options to the assembler.

NOTE Use the **Additional Options** text box of the **Listing File Options** panel to specify options that you want to apply to *all* assembly-language files in the current build target.

Use the `OPT` directive for options that you want to apply to just the assembly-language source file in which the `OPT` directive appears.

Figure 3.30 shows the **Listing File Options** target settings panel. Table 3.26 explains each panel option.

Figure 3.30 Listing File Options Target Settings Panel


Listing File Options

- [ ] Fold Trailing Comment
- [ ] Format Messages
- [ ] Relative Comment Spacing
- [ ] Print Conditional Assembly Directive
- [ ] Expand DEFINE Directive Strings
- [ ] Print Macro Definitions
- [ ] Print Memory Utilization Report
- [ ] Flag Unresolved References
- [ ] Display Warning Messages
- [ ] Form Feed for Page Ejects
- [ ] Pretty Print Listing
- [ ] Print DC Expansion
- [ ] Generate Listing Headers
- [ ] Print Macro Calls
- [ ] Print Macro Expansions
- [ ] Print Conditional Assembly
- [ ] Print Skipped Conditional Assembly Lines

Additional Options:

Table 3.26 Listing File Options Panel Options

Option	Explanation
Fold Trailing Comment	Checked—The assembler folds a comment that trails a source-code statement underneath that statement, aligned with the opcode field. This setting corresponds to the `FC` option of the `OPT` directive and to the `-ofc` command-line option. Cleared—The assembler does not fold trailing comments.
Form Feed for Page Ejects	Checked—The assembler inserts form feeds into the listing file. Each form feed causes a printer to perform a page eject. This setting corresponds to the `FF` option of the `OPT` directive and to the `-off` command-line option. Cleared—The assembler does not insert form feeds into the listing file.
Format Messages	Checked—The assembler inserts format messages into the listing file such that the message text aligns and breaks at word boundaries. This setting corresponds to the `FM` option of the `OPT` directive and to the `-ofm` command-line option. Cleared—The assembler does not insert format messages into the listing file.
Pretty Print Listing	Checked—The assembler aligns fields of the listing file at fixed column positions (without regard to the format of the related source file). This setting corresponds to the `PP` option of the `OPT` directive and to the `-opp` command-line option. Cleared—The assembler does not align fields of the listing file at fixed column positions.
Relative Comment Spacing	Checked—The assembler uses relative comment spacing in the listing file. Checking this checkbox causes the position of comments in the listing file to float. This setting corresponds to the `RC` option of the `OPT` directive and to the `-orc` command-line option. Cleared—The assembler uses fixed comment spacing in the listing file.
Print DC Expansion	Checked—The assembler prints Define Constant (`DC`) expansions in the listing file. This setting corresponds to the `CEX` option of the `OPT` directive and to the `-ocex` command-line option. Cleared—The assembler does not print Define Constant expansions in the listing file.

Table 3.26 Listing File Options Panel Options (*continued*)

Option	Explanation
Print Conditional Assembly Directive	Checked—The assembler prints conditional-assembly directives in the listing file. This setting corresponds to the `CL` option of the `OPT` directive and to the `-ocl` command-line option. Cleared—The assembler does not print conditional-assembly directives in the listing file.
Generate Listing Headers	Checked—The assembler generates listing headers, titles, and subtitles in the listing file. This setting corresponds to the `HDR` option of the `OPT` directive and to the `-ohdr` command-line option. Cleared—The assembler does not generate listing headers, titles, and subtitles in the listing file.
Expand DEFINE Directive Strings	Checked—The assembler prints expanded `DEFINE` directives in the listing file. This setting corresponds to the `MD` option of the `OPT` directive and to the `-omd` command-line option. Cleared—The assembler does not print expanded `DEFINE` directives in the listing file.
Print Macro Calls	Checked—The assembler prints macro calls in the listing file. This setting corresponds to the `MC` option of the `OPT` directive and to the `-omc` command-line option. Cleared—The assembler does not print macro calls in the listing file.
Print Macro Definitions	Checked—The assembler prints macro definitions in the listing file. This setting corresponds to the `MD` option of the `OPT` directive and to the `-omd` command-line option. Cleared—The assembler does not print macro definitions in the listing file.
Print Macro Expansions	Checked—The assembler prints macro expansions in the listing file. This setting corresponds to the `MEX` option of the `OPT` directive and to the `-omex` command-line option. Cleared—The assembler does not print macro expansions in the listing file.

Table 3.26 Listing File Options Panel Options (*continued*)

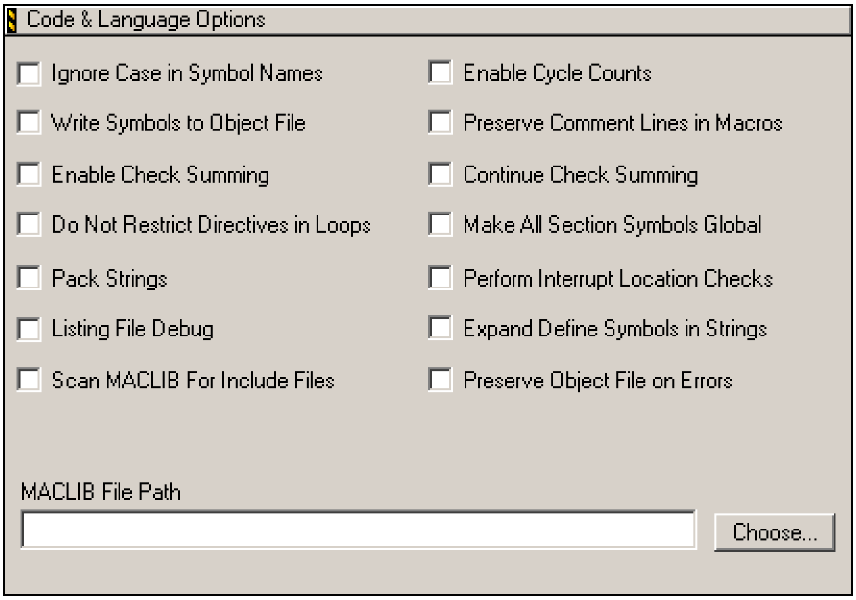
Option	Explanation
Print Memory Utilization Report	Checked—The assembler prints s a report of load and runtime memory-use information in the listing file. This setting corresponds to the `MU` option of the `OPT` directive and to the `-omu` command-line option. Cleared—The assembler does not print the memory-utilization report.
Print Conditional Assembly	Checked—The assembler prints conditional-assembly and section nesting-level information in the listing file. This setting corresponds to the `NL` option of the `OPT` directive and to the `-onl` command-line option. Cleared—The assembler does not print conditional-assembly and section nesting-level information in the listing file.
Flag Unresolved References	Checked—The assembler generates a warning at assembly time for each unresolved external reference. This setting, valid just in relocatable mode, corresponds to the `UR` option of the `OPT` directive and to the `-our` command-line option. Cleared—The assembler does not generate a warning at assembly time for each unresolved external reference.
Print Skipped Conditional Assembly Lines	Checked—The assembler prints in the listing file assembly-language statements skipped due to conditional assembly. This setting corresponds to the `U` option of the `OPT` directive and to the `-ou` command-line option. Cleared—The assembler does not print in the listing file assembly-language statements skipped due to conditional assembly.
Display Warning Messages	Checked—The assembler prints all warning messages in the listing file. This setting corresponds to the `W` option of the `OPT` directive and to the `-ow` command-line option. Cleared—The assembler does not display warning messages in the listing file.
Additional Options	Enter additional command-line options to pass to the assembler.

Code & Language Options

Use this panel to select code- and symbol-generation options for the StarCore assembler.

Figure 3.31 shows the **Code & Language Options** target settings panel. Table 3.27 explains each panel option.

Figure 3.31 Code & Language Options Target Settings Panel


Code & Language Options

- [ ] Ignore Case in Symbol Names
- [ ] Write Symbols to Object File
- [ ] Enable Check Summing
- [ ] Do Not Restrict Directives in Loops
- [ ] Pack Strings
- [ ] Listing File Debug
- [ ] Scan MACLIB For Include Files
- [ ] Enable Cycle Counts
- [ ] Preserve Comment Lines in Macros
- [ ] Continue Check Summing
- [ ] Make All Section Symbols Global
- [ ] Perform Interrupt Location Checks
- [ ] Expand Define Symbols in Strings
- [ ] Preserve Object File on Errors

MACLIB File Path

Table 3.27 Code & Language Options Panel Options

Option	Explanation
Ignore Case in Symbol Names	Checked—The assembler ignores the case of symbol, section, and macro names. This setting corresponds to the `IC` option of the `OPT` directive and to the `-oic` command-line option. Cleared—The assembler considers the case of symbol, section, and macro names.
Enable Cycle Counts	Checked—The assembler enables the cycle counter and clear total-cycle-count features. This setting corresponds to the `CC` option of the `OPT` directive and to the `-occ` command-line option. Checking this checkbox causes the listing file to show a cycle count for each instruction entry. Cleared—The assembler disables the cycle counter. Note that cycle counts assume a full instruction-fetch pipeline and no wait states.

Table 3.27 Code & Language Options Panel Options (*continued*)

Option	Explanation
Write Symbols to Object File	Checked—The assembler writes symbol information to the object files it generates. This setting corresponds to the `SO` option of the `OPT` directive and to the `-oso` command-line option. Cleared—The assembler does not write symbol information to assembler-generated object files.
Preserve Comment Lines in Macros	Checked—The assembler preserves comment lines in macros. This setting corresponds to the `CM` option of the `OPT` directive and to the `-ocm` command-line option. Cleared—The assembler does not preserve comment lines in macros.
Enable Check Summing	Checked—The assembler allows check summing of instruction and data values and to clear the cumulative checksum. Cleared—The assembler does not allow check summing of instruction and data values. You can use the `@CHK()` function to obtain the checksum value. Note that the assembler never preserves a comment line in a macro definition that starts with two consecutive semicolons (`;;`). This setting corresponds to the `CK` option of the `OPT` directive and to the `-ock` command-line option.
Continue Check Summing	Checked—The assembler re-enables check summing of instructions and data. This setting corresponds to the `CONTCK` option of the `OPT` directive and to the `-ocontck` command-line option. Checking this checkbox does not cause the assembler to clear the cumulative checksum value. Cleared—The assembler does not re-enable check summing of instructions and data.
Do Not Restrict Directives in Loops	Checked—The assembler suppresses error messages related to directives that might be invalid in `DO` loops. This setting corresponds to the `DLD` option of the `OPT` directive and to the `-odld` command-line option. Cleared—The assembler generates error messages related to directives that might be invalid in `DO` loops.

Table 3.27 Code & Language Options Panel Options (*continued*)

Option	Explanation
Make All Section Symbols Global	Checked—The assembler treats all sections as if you had declared them explicitly as `GLOBAL`. This setting corresponds to the `GL` option of the `OPT` directive and to the `-ogl` command-line option. You must check this checkbox before explicitly defining any section in a source file. Cleared—The assembler does not treat all sections as if you had declared them explicitly as `GLOBAL`.
Pack Strings	Checked—The assembler packs strings that appear in the Define Constant (`DC`) directive. This setting corresponds to the `PS` option of the `OPT` directive and to the `-ops` command-line option. The assembler packs individual bytes of strings into consecutive target words for the length of the string. Cleared—The assembler does not pack strings that appear in the `DC` directive.
Perform Interrupt Location Checks	Checked—The assembler checks for DSP instructions that cannot appear in the interrupt-vector locations of program memory. This setting corresponds to the `INTR` option of the `OPT` directive and to the `-ointr` command-line option. Cleared—The assembler does not check for DSP instructions that cannot appear in the interrupt-vector locations of program memory.
Listing File Debug	Checked—The assembler uses the source listing, instead of the assembly-language source file, as the debug source file. This setting corresponds to the `LDB` option of the `OPT` directive and to the `-oldb` command-line option. If you check this checkbox, you must also check the **Create List File** checkbox of the **Assembler Preprocessors** panel. Cleared—The assembler uses the assembly-language source file, instead of the source listing, as the debug source file.
Expand Define Symbols in Strings	Checked—The assembler expands `DEFINE` symbols in strings. This setting corresponds to the `DEX` option of the `OPT` directive and to the `-odex` command-line option. Cleared—The assembler does not expand `DEFINE` symbols in strings.

Table 3.27 Code & Language Options Panel Options (*continued*)

Option	Explanation
Scan MACLIB for Include Files	Checked—The assembler searches the `MACLIB` directory paths for include files. The assembler searches these paths in addition to the `INCLUDE` directory or the paths shown in the **Path For Include Files** text box of the **Assembler Preprocessors** panel. This setting corresponds to the `MI` option of the `OPT` directive and to the `-omi` command-line option. Cleared—The assembler searches for include files just in the `INCLUDE` directory or the paths shown in the **Path For Include Files** text box of the **Assembler Preprocessors** panel.
MACLIB File Path	Enter the path to the directory that contains macro definitions. This option corresponds to the `MACLIB` directive. Alternatively, click the **Choose** button, then use the resulting dialog box.
Preserve Object File on Errors	Checked—The assembler preserves object files if assembly errors occur. This setting corresponds to the `SVO` option of the `OPT` directive and to the `-osvo` command-line option. Cleared—The assembler discards object files if assembly errors occur.

C Language

Use this panel to configure settings that tell the compiler the version of the C language you use.

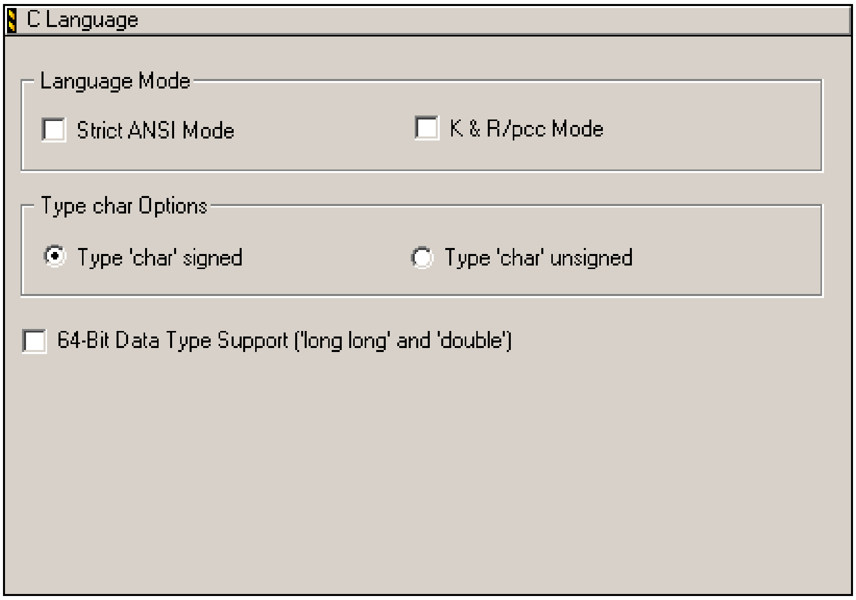
The CodeWarrior C compiler's default mode is ANSI/ISO mode with extensions.

If your C-language source code adheres to the ANSI/ISO specification with extensions, do *not* enable any options of this target settings panel. If your source code adheres strictly to the ANSI/ISO specification, check the **Strict ANSI Mode** checkbox. If your source code adheres to the Kernighan & Ritchie version of C, check the **K & R/pcc Mode** checkbox.

You can compile source files in just one C-language version at a given time. To compile source files in multiple versions, you must compile the source code sequentially, changing your version choice between compilations.

Figure 3.32 shows the **C Language** target settings panel. Table 3.28 explains each panel option.

Figure 3.32 C Language Target Settings Panel


C Language

Language Mode

☐ Strict ANSI Mode   ☐ K & R/pcc Mode

Type char Options

● Type 'char' signed   ○ Type 'char' unsigned

☐ 64-Bit Data Type Support ('long long' and 'double')

Table 3.28 C Language Panel Options

Option	Explanation
Strict ANSI Mode	Checked—The C compiler operates in strict ANSI mode. In this mode, the compiler strictly applies the rules of the ANSI/ISO specification to all input files. This setting is equivalent to the `-ansi` command-line option. The compiler issues a warning for each ANSI/ISO extension it finds. Cleared—The C compiler does not operate in strict ANSI mode.
K & R/pcc Mode	Checked—The C compiler operates in Kernighan & Ritchie mode. In this mode, the compiler applies the rules of Kernighan & Ritchie syntax to all input files. This setting is equivalent to the `-kr` command-line option. Cleared—The C compiler does not operate in Kernighan & Ritchie mode.

Table 3.28 C Language Panel Options (*continued*)

Option	Explanation
Type char Options	Use these option buttons to specify how the compiler interprets `char` data types: • **Type 'char' signed**—Select to have the compiler treat all `char` data types as signed. This setting is the default. • **Type 'char' unsigned**—Select to have the compiler treat all `char` data types as unsigned (as if you declared them `unsigned char`). This setting is equivalent to the `-usc` command line option
64-Bit Data Type Support	Checked—The C compiler supports 64-bit data types `long long` and `double`. A `long long` is a 64-bit integer. A `double` is a 64-bit double-precision floating-point value. Cleared—The C compiler does not support data types `long long` and `double`.

Compiler

Use this panel to specify the behavior of the Compiler for the current build target. Refer to the *Enterprise C Compiler User Guide* for complete documentation of this tool.

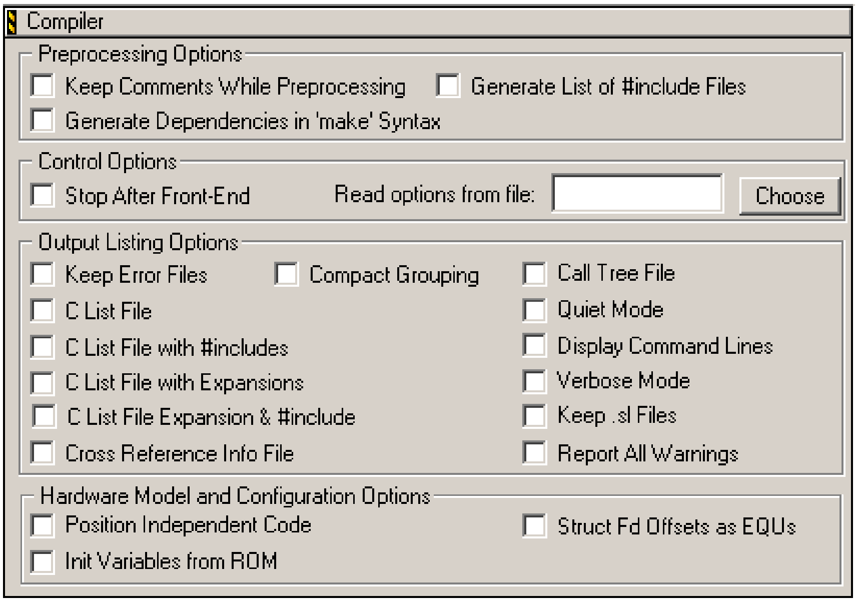
NOTE The compiler applies the preprocessing options of this panel when you select a C source file in the project window and select **Project > Preprocess**. Otherwise, the compiler does not apply the preprocessing options.

Figure 3.33 shows the **Compiler** target settings panel. The panel has these major groups of options:

- Preprocessor Options—control preprocessor behavior
- Control Options—control compiler and shell behavior
- Output Listing Options—control how the compiler formats its listing file and its error and warning messages
- Hardware Model and Configuration Options—control how the compiler structures the object code it generates

Table 3.29 explains each panel option.

Figure 3.33 Compiler Target Settings Panel



- [ ] Keep Comments While Preprocessing
- [ ] Generate List of #include Files
- [ ] Generate Dependencies in 'make' Syntax
- [ ] Stop After Front-End
- Read options from file:
- [ ] Keep Error Files
- [ ] Compact Grouping
- [ ] Call Tree File
- [ ] C List File
- [ ] Quiet Mode
- [ ] C List File with #includes
- [ ] Display Command Lines
- [ ] C List File with Expansions
- [ ] Verbose Mode
- [ ] C List File Expansion & #include
- [ ] Keep .sl Files
- [ ] Cross Reference Info File
- [ ] Report All Warnings
- [ ] Position Independent Code
- [ ] Struct Fd Offsets as EQUs
- [ ] Init Variables from ROM

Table 3.29 Compiler Panel Options

Option	Explanation
Keep Comments While Preprocessing	Checked—The preprocessor includes source-file comments in its output. This setting is equivalent to the `-C` command-line option. Cleared—The preprocessor discards source-file comments from its output.
Generate List of #include Files	Checked—The preprocessor lists the full path to each `#include` file that the selected source files reference. This setting is equivalent to the `-MH` command-line option. The list includes all nested `#include` files. Cleared—The preprocessor does not list the full path to each `#include` file that the selected source files reference.

Table 3.29 Compiler Panel Options (*continued*)

Option	Explanation
Generate Dependencies in ‘make’ Syntax	Checked—The preprocessor lists (in `MAKE` format) the name of each generated output file, along with the full paths to the source files on which the output file depends. This setting is equivalent to the `-M` command-line option. Cleared—The preprocessor does not list the names of generated output files and dependent source files.
Stop After Front-End	Checked—The compiler performs just front-end processing on the source files in the current build target. This setting is equivalent to the `-cfe` command-line option. Checking this checkbox is useful when you prepare source files for global optimization. Also, checking this checkbox causes the compiler to verify that your source files contain no syntax errors. Having no syntax errors is one of the essential requirements of StarCore build-tools processing. Cleared—The compiler performs full processing (including front-end processing) on the source files in the current build target.
Read options from file	Enter the path to a file that contains compiler command-line options. This setting is equivalent to the `-F` *`file`* command-line option. The filename must use the `.opt` extension. The shell treats the options in this file as if you passed them on the command-line. Each time you invoke the compiler, you can select a file with the set of options that suits your needs. Note that the IDE does not verify the validity of the options in the file.

Table 3.29 Compiler Panel Options (*continued*)

Option	Explanation
Keep Error Files	Checked—The compiler keeps the intermediate error files it generates, instead of displaying the error-file messages in the **Errors and Warnings** window. This setting is equivalent to the `-de` command-line option. Checking this checkbox causes the compiler to: • create an error file for each source file in the build target (whether or not the file has an error) • name each error file with the root filename of the related source file, followed by the suffix `.err`. For example, the error file for `main.c` is `main.err`. • put each error file in the project directory Cleared—The compiler displays error-file messages in the **Errors and Warnings** window.
Compact Grouping	Checked—The compiler places each instruction of a Variable-Length Execution Set (VLES) on the same line of the intermediate assembly-language source file that the compiler generates. This option is equivalent to the `-Xllt -mlp` command-line option. Cleared—The compiler places each VLES instruction on a line separate from the line of the compiler-generated intermediate assembly-language source file. Note that you must check the **Keep .sl Files** checkbox in order for this setting to work.
Call Tree File	Checked—The compiler creates a PostScript® file that contains a call-tree graph. This setting is equivalent to the `-dc 4` command-line option. You can print this file on a PostScript printer. Cleared—The compiler does not generate a call-tree graph.
C List File	Checked—The compiler creates a list file for each C source file in the build target. This setting is equivalent to the `-dL` command-line option. Each list file has just the contents of the related source file. Each list filename consists of the root filename of the related source file followed by the `.lis` extension. Cleared—The compiler does not create a list file for each C source file in the build target.

Table 3.29 Compiler Panel Options (*continued*)

Option	Explanation
C List File with #includes	Checked—The compiler creates a list file, with include-file information, for each C source file in the build target. This setting is equivalent to the `-dL1` command-line option. Each list file has the contents of the related source file along with the contents of each included file. The content of each include file appears on the line following the file's `#include` directive. Each list filename consists of the root filename of the related source file followed by the `.lis` extension. Cleared—The compiler does not create a list file, with include-file information, for each C source file in the build target.
C List File with Expansions	Checked—The compiler creates a list file, with expansion information, for each C source file in the build target. This setting is equivalent to the `-dL2` command-line option. Each list file contains the contents of the related source file and various expansions (such as for macros, line splices, and trigraphs). Each list filename consists of the root filename of the related source file followed by the `.lis` extension. Cleared—The compiler does not create a list file, with expansion information, for each C source file in the build target.
C List File Expansion & #include	Checked—The compiler creates a list file, with include-file and expansion information, for each C source file in the build target. This setting is equivalent to the `-dL3` command-line option. Each list file contains the contents of the related source file, the contents of each included file, and various expansions (such as for macros, line splices, and trigraphs). The content of each include file appears on the line following the file's `#include` directive. Each list filename consists of the root filename of the related source file followed by the `.lis` extension. Cleared—The compiler does not create a list file, with include-file and expansion information, for each C source file in the build target.
Cross Reference Info File	Checked—The compiler creates a cross-reference file for each C source file in the build target. This setting is equivalent to the `-dx` command-line option. Each cross-reference filename consists of the root filename of the related source file followed by the `.xrf` extension. Cleared—The compiler does not create a cross-reference file for each C source file in the build target.

Table 3.29 Compiler Panel Options (*continued*)

Option	Explanation
Quiet Mode	Checked—The IDE displays just error messages that the compiler emits. This setting is equivalent to the `-q` command-line option. Checking this checkbox causes the IDE to suppress warning and informational messages. Cleared—The IDE displays error, warning, and information messages that the compiler emits.
Display Command Lines	Checked—The IDE shows the command lines it passes to each build tool, without actually invoking those tools. This setting is equivalent to the `-n` command-line option. Checking this checkbox disables object-file creation. Use this option to verify that each command line includes the options you expect (based on your settings in each target settings panel). Cleared—The IDE hides the command lines it passes to each build tool.
Verbose Mode	Checked—The IDE shows each command line it passes to the shell, along with all progress, error, warning, and informational messages that those tools emit. This setting is equivalent to the `-v` command-line option. Cleared—The IDE hides each command line it passes to the shell.
Keep .sl Files	Checked—The compiler keeps the intermediate assembly-language source files (`.sl` files) it creates, instead of deleting them. This setting is equivalent to the `-s` command-line option. The compiler generates one `.sl` file for each C source file in the build target. Cleared—The compiler discards the intermediate assembly-language source files it creates.
Report All Warnings	Checked—The compiler reports all possible warnings to the IDE for display in the **Errors and Warnings** window. This setting is equivalent to the `-Wall` command-line option. Cleared—The compiler does not report all possible warnings to the IDE.

Table 3.29 Compiler Panel Options (*continued*)

Option	Explanation
Position Independent Code	Checked—The compiler generates position-independent code. This setting is equivalent to the `-pic` command-line option. Cleared—The compiler does not generate position-independent code.
Init Variables from ROM	Checked—The compiler places the initialization data for your program’s global variables in a separate section. This setting is equivalent to the `-mrom` command-line option. The IDE can manipulate the section at link time and load time. Cleared—The compiler bypasses placing the initialization data for your program’s global variables in a separate section. You can clear this checkbox as you develop your source code, because a separate loader program handles initializing your program’s global variables. After you finish development, you can check this checkbox, then place into ROM the segment with the initialization data for your global variables.
Struct Fd Offsets as EQUs	Checked—The compiler includes the offsets of C data-structure field definitions in each generated intermediate assembly-language source file. This setting is equivalent to the `-do` command-line option. Cleared—The compiler does not include the offsets of C data-structure field definitions in each generated intermediate assembly-language source file.

I/O & Preprocessors

Use this panel to specify additional directories that the compiler uses when searching for include files. Also, use this panel to define and undefine preprocessor macros.

Figure 3.34 shows the **I/O & Preprocessors** target settings panel. Table 3.30 explains each panel option.

Figure 3.34 I/O & Preprocessors Target Settings Panel

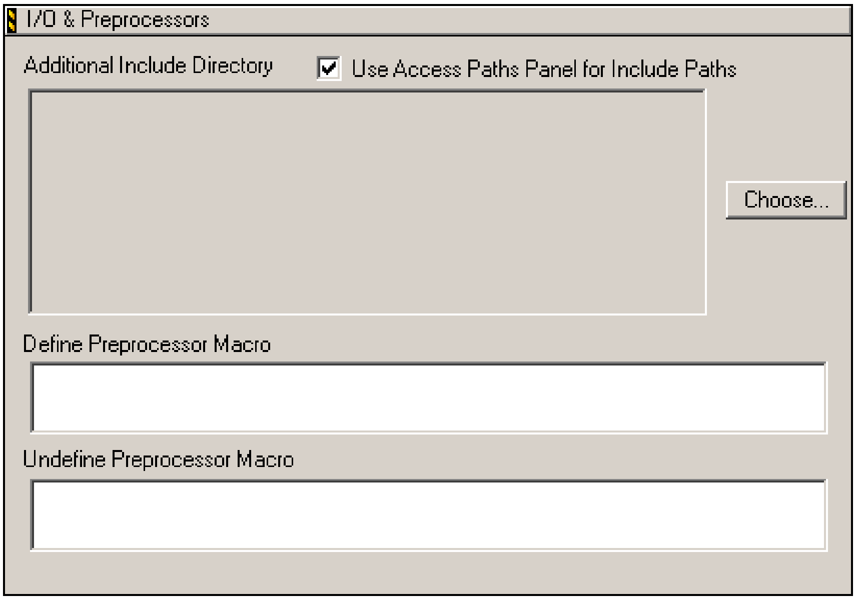

I/O & Preprocessors
Additional Include Directory
Use Access Paths Panel for Include Paths
Choose...
Define Preprocessor Macro
Undefine Preprocessor Macro

Table 3.30 I/O & Preprocessors Panel Options

Option	Explanation
Additional Include Directory	This text box shows a list of search paths that the compiler uses when searching for include files. To specify a path, click the **Choose** button and use the resulting dialog box. Click the **Choose** button multiple times to specify multiple paths. This setting is equivalent to the `-I` *`path`* command-line option.
Use Access Paths Panel for Include Paths	Checked—The compiler uses the paths that you specify in the **Access Paths** target settings panel instead of the paths shown in the **Additional Include Directory** text box. This setting is equivalent to the `-I` *`path`* command-line option. Cleared—The compiler uses the paths shown in the **Additional Include Directory** text box. For a computer that runs Windows® 2000, the maximum command-line length is 2,047 characters. For a computer that runs Windows XP, the maximum command-line length is 8,191 characters. If you see errors about too many include paths, try removing recursive-path definitions from the **Access Paths** target settings panel.

Table 3.30 I/O & Preprocessors Panel Options (*continued*)

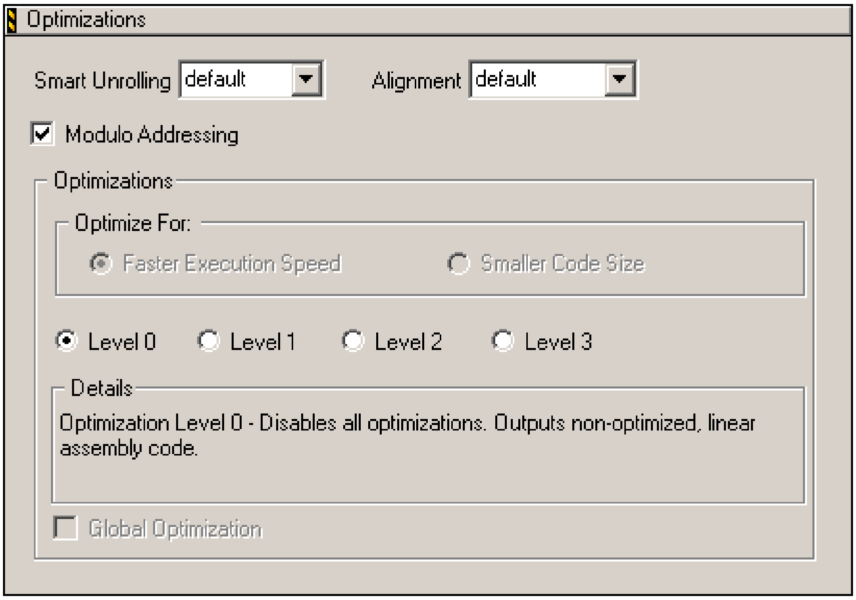
Option	Explanation
Define Preprocessor Macro	Use this text box to define preprocessor macros and optionally assign their values. This setting is equivalent to the `-D` *`name`*[=*`value`*] command-line option. To assign a value, use the equal sign (`=`) with no white space. Separate multiple macro definitions with a comma followed by a space. For example, this is a valid entry: `EXTENDED_FEATURE=ON, DEBUG_CHECK` Note that if you do not assign a macro value, the shell assigns a default value of 1. Also note that whitespace is required between macros (in the example above, the space between the comma and `DEBUG_CHECK` is required).
Undefine Preprocessor Macro	Use this text box to undefine preprocessor macros. Separate multiple macro undefinitions with commas. This setting is equivalent to the `-U` *`name`* command-line option. For example, this is a valid entry: `EXTENDED_FEATURE, DEBUG_CHECK` Note that the shell processes any **Undefine Preprocessor Macro** items after its processes all **Define Preprocessor Macro** items.

Optimizations

Use this panel to configure the optimizations that the C compiler performs. The compiler's optimizer can apply any of its optimizations in either global or non-global optimization mode. You can apply global optimization at the end of the development cycle, after compiling and optimizing all source files individually or in groups.

Figure 3.35 shows the **Optimizations** target settings panel. Table 3.31 explains each panel option. For Level 1, Level 2, or Level 3 optimizations, you can add the supplemental optimizations explained in Table 3.32 on page 112.

Figure 3.35 Optimizations Target Settings Panel



[✓] Modulo Addressing

[● Faster Execution Speed] ○ Smaller Code Size

[● Level 0] ○ Level 1 ○ Level 2 ○ Level 3

Optimization Level 0 - Disables all optimizations. Outputs non-optimized, linear assembly code.

☐ Global Optimization

Table 3.31 Optimizations Panel Options

Option	Explanation
Smart Unrolling	Use this list box to specify the *maximum* unrolling factor the compiler uses in the automatic- loop-unrolling optimization. This setting is equivalent to the `-u`*`level`* command-line option. Specify **default** to let the compiler select the unrolling factor. Note that if you specify **0**, the compiler does not unroll any loops. If you specify **4**, however, the compiler unrolls loops with a factor of either 2 or 4, depending on the gain that the compiler determines.
Alignment	Use this list box to specify the alignment level the compiler uses. This setting is equivalent to the `-align` *`level`* command-line option. • **default**—level 0 applies to size optimizations, and level 2 applies to speed optimizations • **0** — disable alignment • **1** — align hardware loops • **2** — align hardware and software loops • **3** — align all existing labels • **4** — align all existing labels and subroutine-return points Note that using a higher alignment constraint can not only increase execution speed but also increase object-code size.
Modulo Addressing	Checked—The compiler uses modulo addressing. This setting is equivalent to the `-mod` command-line option. Cleared—The compiler does not use modulo addressing.

Table 3.31 Optimizations Panel Options (*continued*)

Option	Explanation
Optimizations	These options control the optimizations the compiler performs. The Details area explains the optimizations the compiler performs at the specified level. • **Faster Execution Speed**—Select this option button to have the compiler optimize object code at the specified level such that the resulting binary file has a faster execution speed, as opposed to a smaller executable code size. This setting is equivalent to the `-Os` command-line option. • **Smaller Code Size**—Select this option button to have the compiler optimize object code at the specified level such that the resulting binary file has a smaller executable code size, as opposed to a faster execution speed. This setting is equivalent to the `-Os` command-line option.
Level 0	Select this option button to disable optimizations. This setting is equivalent to the `-O0` command-line option. The compiler generates unoptimized, linear assembly-language code.
Level 1	Select this option button to have the compiler perform all target-independent (that is, non-parallelized) optimizations, such as function inlining. This setting is equivalent to the `-O1` command-line option. The compiler omits all target-specific optimizations and generates linear assembly-language code.
Level 2	Select this option button to have the compiler perform all optimizations (both target-independent and target-specific). This setting is equivalent to the `-O2` command-line option. The compiler outputs optimized, non-linear, parallelized assembly-language code. This is the default optimization level.

Table 3.31 Optimizations Panel Options (*continued*)

Option	Explanation
Level 3	Select this option button to have the compiler perform all Level 2 optimizations and then have the low-level optimizer perform global-algorithm register allocation. This setting is equivalent to the `-O3` command-line option. Using this optimization level, the compiler generates code that is usually faster than that produced using Level 2 optimizations.
Global Optimization	Checked—The compiler applies the selected optimizations across all the files in the build target. This *global* optimization is the most effective. This setting is equivalent to the `-cfe` compiler command-line option followed by the `-Og` link-phase command-line option. Cleared—The compiler creates intermediate files that have the `.obj` file extension. Note that global optimization is available for all optimization levels except Level 0. If you enable global optimization, the IDE applies a value of -1 for code size and data size to a globally-optimized file, rather than reporting the code and data size in the project window.

Table 3.32 Supplemental Optimizations

Optimization Type	Explanation
Space	The optimizer favors program size over the optimization level that you specify. Programs or modules optimized for space require less memory but might sacrifice program-execution speed.
Cross-file	The optimizer applies the specified optimization level across all application files at the same time. This approach yields the most efficient program code. Combining space optimization at Level 3 (`-O3 -Os`) generates programs with smaller code size than those obtained with specifiers `-O2 -Os`. Cross-file optimization is complex process, so it significantly increases compilation time. Because of this increased compilation time, try to apply cross-file optimization at the end of the development cycle. For example, apply cross-file optimization after compiling and optimizing source files individually or in groups. The optimizer's default setting is not to perform cross-file optimization.

Passthrough, Hardware

Use this panel to specify command-line options that the shell program (scc) passes directly to individual build tools (such as the front-end of the compiler, the various optimizers, and the assembler).

NOTE The IDE applies the command-line options that you specify in this panel to the compilation of the C-language source files in a build target (even the options that you enter in the **To Assembler** text box).

To pass command-line options through to the assembler for application to the assembly-language files in a build target, use the **Read Options from File** text box of the **Assembler Preprocessors** panel or the **Additional Options** text box of the **Listing File Options** panel.

Figure 3.36 shows the **Passthrough, Hardware** target settings panel. Table 3.33 explains each panel option.

Figure 3.36 Passthrough, Hardware Target Settings Panel

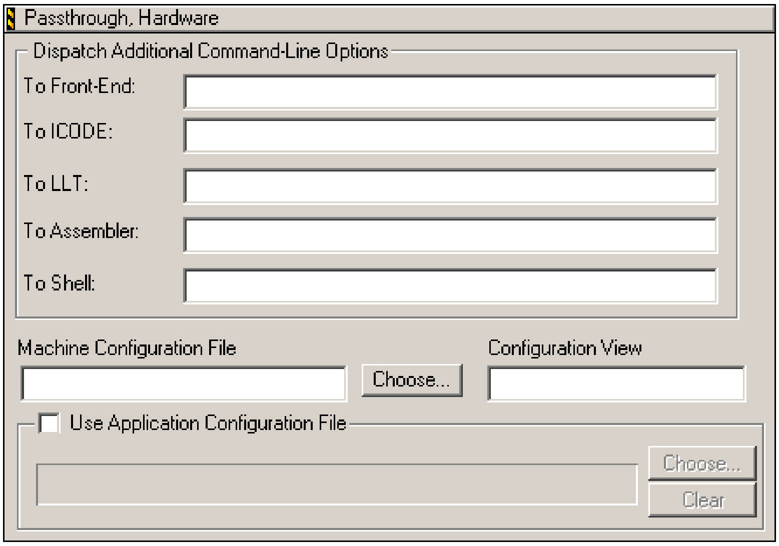


Table 3.33 Passthrough, Hardware Panel Options

Option	Explanation
To Front-End	Enter command-line options for the shell (scc) to pass to the front-end compiler. This setting is equivalent to the `-Xcfe` options command-line option.
To ICODE	Enter command-line options for the shell (scc) to pass to the compiler's high-level optimizer. This setting is equivalent to the `-Xicode` options command-line option.
To LLT	Enter command-line options for the shell (scc) to pass to the compiler's low-level optimizer. This setting is equivalent to the `-Xllt` options command-line option.
To Assembler	Enter command-line options for the shell (scc) to pass to the assembler. This setting is equivalent to the `-Xasm` options command-line option.

Table 3.33 Passthrough, Hardware Panel Options (*continued*)

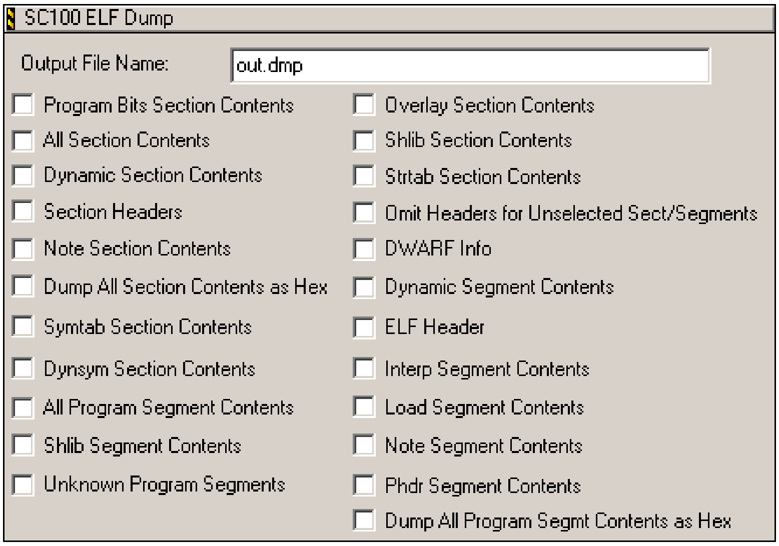
Option	Explanation
To Shell	Enter command-line options for the IDE to pass to the shell (scc). The IDE passes the options exactly as you type them and does not check for errors.
Machine Configuration File	Enter the path to a custom machine-configuration file that the build target uses. This setting is equivalent to the `-mc` *filename* command-line option. Alternatively, click the **Choose** button, then use the resulting dialog box.
Configuration View	Enter the application-configuration file view that the build target uses. This setting is equivalent to the `-view` *identifier* command-line option. If you use this text box, you must check the **Use Application Configuration File** checkbox and specify an application-configuration file.
Use Application Configuration File	Checked—The compiler uses a custom application-configuration file. This setting is equivalent to the `-ma` *`filename`* command line option. To specify the file, click the **Choose** button, then use the resulting dialog box. To remove the currently specified file, click the **Clear** button Cleared—The compiler uses the default application-configuration file. Use a custom application-configuration file to apply different settings and optimization levels to various files and functions of the build target. An application-configuration file must have the `.appli` filename extension.

SC100 ELF Dump

Use this panel to define the configure the behavior of the ELF file dump utility. To view this panel in the Target Settings window, you must go to the **Target Settings** panel and use the **Post-linker** list box to specify **SC100 ELF Dump**.

Figure 3.37 shows the **SC100 ELF Dump** target settings panel. Table 3.34 explains each panel option.

Figure 3.37 SC100 ELF Dump Target Settings Panel


SC100 ELF Dump

Output File Name: out.dmp

- [ ] Program Bits Section Contents
- [ ] All Section Contents
- [ ] Dynamic Section Contents
- [ ] Section Headers
- [ ] Note Section Contents
- [ ] Dump All Section Contents as Hex
- [ ] Symtab Section Contents
- [ ] Dynsym Section Contents
- [ ] All Program Segment Contents
- [ ] Shlib Segment Contents
- [ ] Unknown Program Segments
- [ ] Overlay Section Contents
- [ ] Shlib Section Contents
- [ ] Strtab Section Contents
- [ ] Omit Headers for Unselected Sect/Segments
- [ ] DWARF Info
- [ ] Dynamic Segment Contents
- [ ] ELF Header
- [ ] Interp Segment Contents
- [ ] Load Segment Contents
- [ ] Note Segment Contents
- [ ] Phdr Segment Contents
- [ ] Dump All Program Segmt Contents as Hex

Table 3.34 SC100 ELF Dump Panel Options

Option	Description
Output File Name	Enter the path of the file to which the ELF file dump utility writes its output. Note that if you leave this text box empty, the utility writes its output to the **Errors and Warnings** window.
Program Bits Section Contents	Checked—The utility includes the contents of all program bits sections in the dump. This setting is equivalent to the `-b` command-line option. Cleared—The utility does not include the contents of all program bits sections in the dump.
All Section Contents	Checked—The utility includes the contents of all sections in the dump. This setting is equivalent to the `-a` command-line option. Cleared—The utility does not include the contents of all sections in the dump.

Table 3.34 SC100 ELF Dump Panel Options (*continued*)

Option	Description
Dynamic Section Contents	Checked—The utility includes the contents of all dynamic sections in the dump. This setting is equivalent to the `-d` command-line option. Cleared—The utility does not include the contents of all dynamic sections in the dump. Note that this option does not apply to the SC140 core.
Section Headers	Checked—The utility includes the contents of all hash sections in the dump. This setting is equivalent to the `-h` command-line option. Cleared—The utility does not include the contents of all hash sections in the dump. Note that this option does not apply to the SC140 core.
Note Section Contents	Checked—The utility includes the contents of all note sections in the dump. This setting is equivalent to the `-n` command-line option. Cleared—The utility does not include the contents of all note sections in the dump.
Dump All Section Contents As Hex	Checked—The dump utility writes the contents of all sections in hexadecimal form. This setting is equivalent to the `-x` command-line option. Cleared—The dump utility does not write the contents of sections in hexadecimal form.
Symtab Section Contents	Checked—The utility includes the contents of all symtab sections in the dump. This setting is equivalent to the `-y` command-line option. Cleared—The utility does not include the contents of all symtab sections in the dump.
Dynsym Section Contents	Checked—The utility includes the contents of all dynsym sections in the dump. This setting is equivalent the `-z` command-line option. Cleared—The utility does not include the contents of all dynsym sections in the dump. Note that this option does not apply to the SC140 core.

Table 3.34 SC100 ELF Dump Panel Options (*continued*)

Option	Description
All Program Segment Contents	Checked—The utility includes the contents of all program segments in the dump. This setting is equivalent to the `-A` command-line option. Cleared—The utility does not include the contents of all program segments in the dump.
Shlib Segment Contents	Checked—The utility includes the contents of all shlib segments in the dump. This setting is equivalent to the `-S` command-line option. Cleared—The utility does not include the contents of all shlib segments in the dump. Note that this option does not apply to the SC140 core.
Unknown Program Segments	Checked—The utility includes the contents of all unknown type segments (in hexadecimal form) in the dump. This setting is equivalent to the `-U` command-line option. Cleared—The utility does not include the contents of all unknown type segments in the dump.
Overlay Section Contents	Checked—The utility includes the contents of all overlay table sections in the dump. This setting is equivalent to the `-o` command-line option. Cleared—The utility does not include the contents of all overlay table sections in the dump.
Shlib Section Contents	Checked—The utility includes the contents of all shlib sections in the dump. This setting is equivalent to the `-s` command-line option. Cleared—The utility does not include the contents of all shlib sections in the dump
Strtab Section Contents	Checked—The utility includes the contents of all strtab sections in the dump. This setting is equivalent to the `-t` command-line option. Cleared—The utility does not include the contents of all strtab sections in the dump.

Table 3.34 SC100 ELF Dump Panel Options (*continued*)

Option	Description
Omit Headers for Unselected Sect/ Segments	Checked—The utility limits header information to the specified sections and segments. This setting is equivalent to the `-q` command-line option. Cleared—The utility does not limit header information to the specified sections and segments.
DWARF Info	Checked—The utility includes DWARF-formatted informational data in the dump. This setting is equivalent to the `-g` command-line option. Cleared—The utility does not include DWARF informational data in the dump. The informational data is useful for debugging symbolics information for sections.
Dynamic Segment Contents	Checked—The utility includes the contents of all dynamic segments in the dump. This setting is equivalent to the `-D` command-line option. Cleared—The utility does not include the contents of all dynamic segments in the dump. Note that this option does not apply to the SC140 core.
ELF Header	Checked—The utility includes ELF-header information in the dump. This setting is equivalent to the `-E` command-line option, and is the default behavior of the dump utility. Cleared—The utility does not include ELF-header information in the dump.
Interp Segment Contents	Checked—The utility includes the contents of all interp segments in the dump. This setting is equivalent to the `-I` command-line option. Cleared—The utility does not include the contents of all interp segments in the dump.
Load Segment Contents	Checked—The utility includes the contents of all load segments in the dump. This setting is equivalent to the `-L` command-line option. Cleared—The utility does not include the contents of all load segments in the dump.

Table 3.34 SC100 ELF Dump Panel Options (*continued*)

Option	Description
Note Segment Contents	Checked—The utility includes the contents of all note segments in the dump. This setting is equivalent to the `-N` command-line option. Cleared—The utility does not include the contents of all note segments in the dump. Note that this option does not apply to the SC140 core.
Phdr Segment Contents	Checked—The utility includes the contents of all phdr segments in the dump. This setting is equivalent to the `-P` command-line option. Cleared—The utility does not include the contents of all phdr segments in the dump.
Dump All Program Segmt Contents as Hex	Checked—The utility writes the contents of all segments in hexadecimal form. This setting is equivalent to the `-X` command-line option. Cleared—The utility does not write the contents of segments in hexadecimal form.

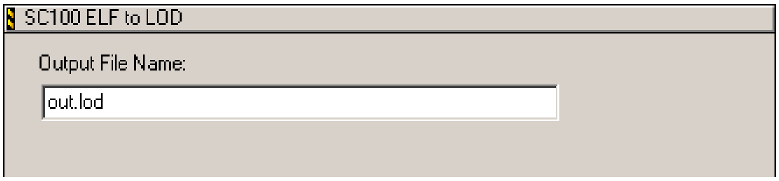
SC100 ELF to LOD

Use this panel to define the name of the LOD file that the elflod utility generates. Figure 3.38 shows the **SC100 ELF to LOD** target settings panel.

NOTE The **SC100 ELF to LOD** panel name does not appear in the panel list of the **Target Settings** window unless you select SC100 ELF to LOD from the Post-linker listbox of the Target Settings panel.

This panel has the **Output File Name** text box. Enter in this text box the path of the file to which the ELF to LOD post-linker writes data.

Figure 3.38 SC100 ELF to LOD Target Settings Panel



Output File Name:
out.lod

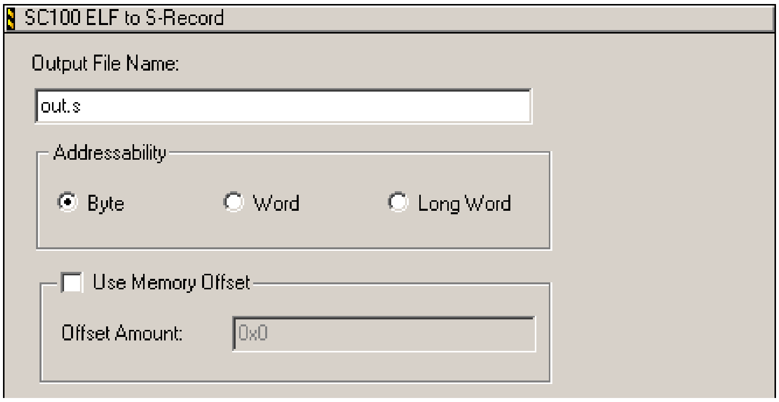
SC100 ELF to S-Record

Use the **SC100 ELF to S-Record** target settings panel to define the name, addressability, and memory offset of the S-Record file generated by the elfsrec utility.

NOTE The **SC100 ELF to S-Record** panel name does not appear in the panel list of the **Target Settings** window unless you select SC100 ELF to S-Record from the Post-linker listbox of the Target Settings panel.

Figure 3.39 shows the **SC100 ELF to S-Record** target settings panel.

Figure 3.39 SC100 ELF to S-Record Target Settings Panel


SC100 ELF to S-Record

Output File Name: out.s

Addressability: [✓ Byte] Word Long Word

☐ Use Memory Offset

Offset Amount: 0x0

Table 3.35 lists and defines each option of the **SC100 ELF to S-Record** settings panel.

Table 3.35 SC100 ELF to S-Record Panel Options

Option	Description
Output File Name	Use this text box to specify the path and name of the file to which the ELF to S-Record post-linker writes S-Records.
Addressability	Use the radio buttons in this group to select byte-, word-, or long word-addressabilty for the generated S-Record file.
Use Memory Offset	Check this box to instruct the elfsrec post-linker to add a memory offset to the memory address of each line of the S-Record file.
Offset Amount	Use this text box to enter the memory offset the ELF to S-Record post- linker applies to the generated file. You can enter this value in hexadecimal or decimal.

Profiler

Use this panel to configure the CodeWarrior profiler so it can interact with your target and collect information about the program running on that target. Some options configure how the profiler collects information from the Debugging and Profiling Unit (DPU).

NOTE See Using the Profiler for procedures you must complete before you can use the profiler.

Figure 3.40 shows the **Profiler** target settings panel. Table 3.36 explains each panel option.

NOTE The DPU-specific settings in this panel correspond directly to DPU configuration registers.

Figure 3.40 Profiler Target Settings Panel

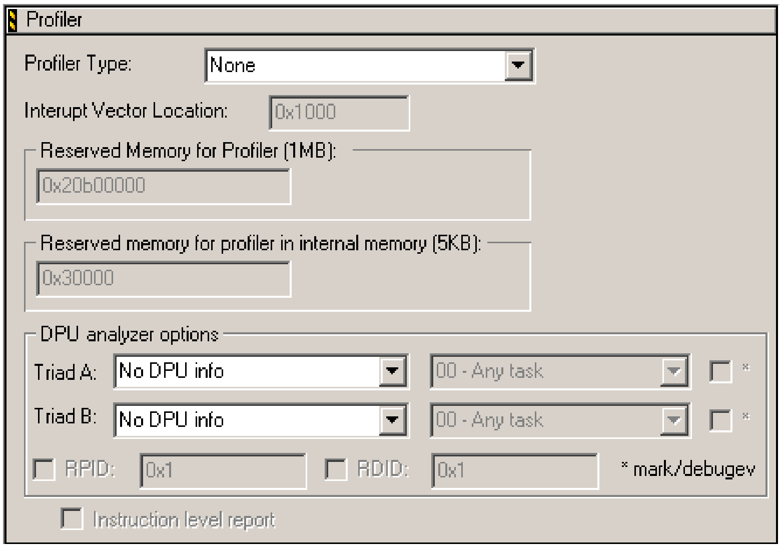

Profiler
Profiler Type: None
Interupt Vector Location: 0x1000
Reserved Memory for Profiler (1MB): 0x20b00000
Reserved memory for profiler in internal memory (5KB): 0x30000
DPU analyzer options
Triad A: No DPU info | 00 - Any task | ☐ *
Triad B: No DPU info | 00 - Any task | ☐ *
☐ RPID: 0x1 ☐ RDID: 0x1 * mark/debugev
☐ Instruction level report

Table 3.36 Profiler Panel Options

Option	Explanation
Profiler Type	Use this list box to specify the type of profiler you use.
Interrupt Vector Location	Enter the address (in hexadecimal) of the vector to the interrupt service routine for the on-chip profiler to use.

Table 3.36 Profiler Panel Options (*continued*)

Option	Explanation
Reserved Memory for Profiler	Enter the base address (in hexadecimal) of a 1-megabyte buffer in external memory for the on-chip profiler to use.
Reserved memory for profiler in internal memory	Enter the base address (in hexadecimal) of a 5-kilobyte buffer of internal memory for the on-chip profiler to use.
Triad A	Use this list box to specify the DPU information that the profiler collects for this triad. Specifying an option other than **None** enables the corresponding User Privilege Level list box.
Triad B	Use this list box to specify the DPU information that the profiler collects for this triad. Specifying an option other than **None** enables the corresponding User Privilege Level list box.
User Privilege Level	This list box becomes enabled after you use the **Triad A** or **Triad B** list box to specify an option other than **None**. Use this list box to specify the privilege level of collected DPU information. Specifying **01 - User tasks (+TID)** or **11 - Supervisor tasks (+TID)** enables the **RPID** and **RDID** options
* mark/debugev	This checkbox becomes enabled after you use the **Triad A** or **Triad B** list box to specify an option other than **None**. Checked—For each triad, the profiler collects DPU information just for code inside `mark` and `debugev` statements Cleared—For each triad, the profiler collects DPU information for the entire application If you want to accurately evaluate specific code regions, insert `mark` and `debugev` statements into your code and check this checkbox.
RPID	Checked—The profiler uses the specified Process Identifier (PID) as a filter for counting triads. The profiler counts just those events associated with the PID that you specify in the corresponding text box. This feature corresponds to configuring the `DP_CR[TIDCM]` bits. Cleared—The profiler counts all triads. In order to use this option, you must specify a User Privilege Level that allows PID filtering.

Table 3.36 Profiler Panel Options (*continued*)

Option	Explanation
RDID	Checked—The profiler uses the specified Data Identifier (DID) as a filter for counting triads. The profiler counts just those events associated with the DID that you specify in the corresponding text box. This feature corresponds to configuring the `DP_CR[TIDCM]` bits. Cleared—The profiler counts all triads. In order to use this option, you must specify a User Privilege Level that allows DID filtering.
Instruction Level Report	Checked—The profiler performs an instruction-level trace. Cleared—The profiler performs a change-of-flow trace. In order to enable this checkbox, you must use the **Profiler Type** list box to specify **On Host**.

Memory-Configuration Files

The Remote Debug Options panel lets you specify a memory-configuration file for your project. A memory configuration file:

- contains commands that define valid accessible areas of memory for your specific board
- indicates memory regions that have special access considerations (such as access size)

Memory-Configuration File Syntax

The commands in a memory configuration file must adhere to these syntax rules:

- All syntax is *not* case sensitive.
- The debugger ignores all white space and tab characters.
- Valid comments use C or C++ style syntax.
- Valid numbers can be in any of these formats:
  - hexadecimal—uses the prefix `0x`; examples: `0x00002222`, `0xA`, `0xCAfeBeaD`
  - octal—uses the prefix `0`; examples: `0123`, `0456`
  - decimal—starts with a number from `1` to `9`; examples: `12`, `126`, `823643`

Memory-Configuration File Commands

This section lists available commands that you can use in a memory configuration file:

reservedchar

Allows you to specify a reserved character for the memory configuration file.

`reservedchar` *`char`*

Parameters

`char`

Any one-byte character.

Remarks

When a read from a reserved memory range or invalid address occurs, the debugger fills the resulting memory buffer with this reserved character.

Example

```
reservedchar 0xBA
```

range

Allows you to specify a memory range for reading or writing, and the attributes of that range.

`range` *`loAddr hiAddr sizeCode access`*

Parameters

`loAddr`

The starting address of the memory range.

`hiAddr`

The ending address of the memory range.

`sizeCode`

Specifies the size, in bytes, that the debug monitor or emulator uses for memory accesses.

`access`

Can be one of the following: `Read`, `Write`, or `ReadWrite`. Use this parameter to make certain areas of your memory map read-only, write-only, or read/write only to the debugger.

Examples

```
range 0xFF000000 0xFF0000FF 4 Read

range 0xFF000100 0xFF0001FF 2 Write

range 0xFF000200 0xFFFFFFFF 1 ReadWrite
```

reserved

Allows you to specify a reserved memory range.

`reserved` *`loAddr hiAddr`*

Parameters

`loAddr`

The starting address of the memory range.

`hiAddr`

The ending address of the memory range.

Remarks

When the debugger tries to read from this location, it fills the memory buffer the character you specify with `reservedchar`. When the debugger tries to write to any of the locations in this range, no write takes place.

Example

```
reserved 0xFF000024 0xFF00002F
```

4 Debugging

This chapter explains how to use the CodeWarrior™ debugger to debug source-code features specific to MXC05, i.300-30, MXC275-30, MXC300-10, and MXC300-30 architectures. The *IDE User's Guide* explains the standard features of the CodeWarrior debugger.

This chapter has these topics:

Cache Window

Use the Cache window to view L1 cache (such as instruction cache or data cache). Also, you can use the Cache window to view L2 cache for targets that support it, such as the i.300-30, MXC 05, and MXC300-30 architectures. To open a Cache window, select **Data > View Cache**, then select a cache from the submenu that appears.

In the Cache window, the **Set**, **Way**, and **Address** columns display the corresponding cache attributes for the displayed lines. Each line appears as a group of **Word** columns.

Table 4.1 shows the basis by which each supported architecture reports cache-validity information. In the Cache window, valid data appears in bold text.

Table 4.1 Cache-Validity Reporting by Architecture

Architecture	Cache	Sector-by-Sector Basis	Line-by-Line Basis
i.300-30	L1	✓	
	L2		✓
MXC 05	L1	✓	
	L2		✓
MXC275-20	L1	✓	
MXC275-30	L1	✓	
MXC300-10	L1	✓	
MXC300-30	L1	✓	
	L2		✓

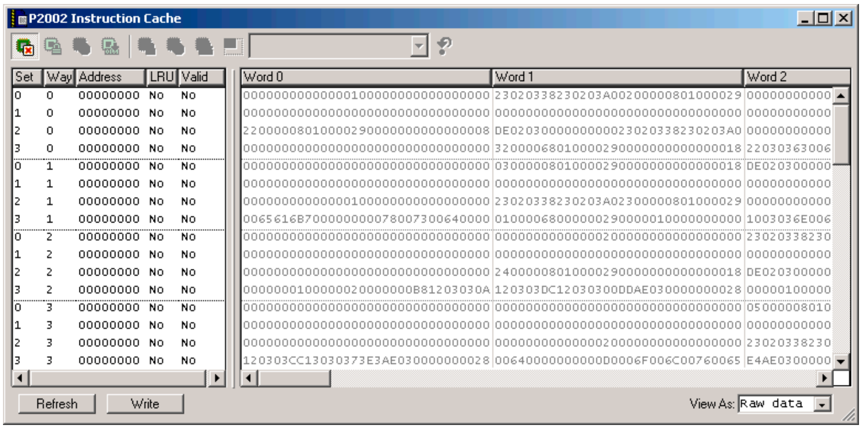
You can select and modify individual cache lines in the window. To commit your changes to the cache memory on the target hardware, click the **Write** button. Red text denotes recently changed data.

NOTE The Cache window disables (grays out) toolbar buttons for unsupported operations. You can still use some toolbar buttons in the Cache window to perform more general cache functions (such as enabling the cache, flushing the cache, and so on).

NOTE For the MXC275-20, the Cache window displays all L1 ICache values as zero.

Figure 4.1 shows a sample Cache window. Table 4.2 explains the items in the window.

Figure 4.1 Cache Window



Set	Way	Address	LRU	Valid	Word 0	Word 1	Word 2
0	0	00000000	No	No	00000000000000010000000000000000	23020338230203A00200000801000029	00000000000
1	0	00000000	No	No	00000000000000000000000000000000	00000000000000000000000000000000	00000000000
2	0	00000000	No	No	22000008010000290000000000000008	DE02030000000000230203382302 03A0	00000000000
3	0	00000000	No	No	00000000000000000000000000000000	32000068010000290000000000000018	22030363006
0	1	00000000	No	No	00000000000000000000000000000000	03000008010000290000000000000018	DE020300000
1	1	00000000	No	No	00000000000000000000000000000000	00000000000000000000000000000000	00000000000
2	1	00000000	No	No	00000000000000010000000000000000	23020338230203A02300000801000029	00000000000
3	1	00000000	No	No	0065616B700000000078007300640000	01000068000000290000010000000000	1003036E006
0	2	00000000	No	No	00000000000000000000000000000000	00000000000000020000000000000000	23020338230
1	2	00000000	No	No	00000000000000000000000000000000	00000000000000000000000000000000	00000000000
2	2	00000000	No	No	00000000000000000000000000000000	24000008010000290000000000000018	DE020300000
3	2	00000000	No	No	0000000100000002000000B81203030A	120303DC12030300DDAE030000000028	00000100000
0	3	00000000	No	No	00000000000000000000000000000000	00000000000000000000000000000000	05000008010
1	3	00000000	No	No	00000000000000000000000000000000	00000000000000000000000000000000	00000000000
2	3	00000000	No	No	00000000000000000000000000000000	00000000000000020000000000000000	23020338230
3	3	00000000	No	No	120303CC13030373E3AE030000000028	00640000000000D0006F006C00760065	E4AE0300000

View As: Raw data

Table 4.2 Cache Window Items

Item	Icon	Explanation
Enable/Disable cache		Enabled—The cache is on Disabled—The cache is off
Lock/Unlock cache		Enabled—The cache is locked (which prevents it from fetching new lines or discarding current valid lines) Disabled—The cache is unlocked (which allows it to fetch new lines and discard current valid lines)

Table 4.2 Cache Window Items (*continued*)

Item	Icon	Explanation
Invalidate cache		Click to invalidate the entire contents of the cache
Flush cache		Click to flush the entire contents of the cache. Flushing the cache involves committing uncommitted data to the next level of the memory hierarchy, then invalidating the data within the cache.
Lock/Unlock line		Enabled—Locks the selected cache lines Disabled—Unlocks the selected cache lines
Invalidate line		Click to invalidate the selected cache lines
Flush line		Click to flush the entire contents of the selected cache lines
Activate/ Deactivate LRU display		Click to toggle the display of the Least Recently Used attribute for each cache line within the display. The value **YES** denotes the least recently used line within each set.
Lock Ways list box (in window toolbar)		Specify the cache ways that should be locked. Locking a cache way means that the data contained in that way must not change. If the cache needs to discard a line, it will not discard locked lines (such as lines explicitly locked, or lines belonging to locked ways).
Inverse LRU		Click to display the inverse of the Least Recently Used attribute for each cache line within the display
Refresh button		Click to have the IDE clear the entire contents of the cache, re-read status information from the target hardware, and update the Cache window display

Table 4.2 Cache Window Items (*continued*)

Item	Icon	Explanation
Write button		Click to have the IDE commit content changes from the Cache window to the cache memory on the target hardware (if the target hardware supports doing so)
View As list box		Specify the format of the data that appears in the Cache window: • Raw data—The Cache window does not format the data • Disassembly—The Cache window formats the data as opcode disassembly

Command-Line Debugging

You can debug from the command line as well as from within the CodeWarrior IDE. The *IDE Automation Guide* covers general command-line debugging features.

Custom Commands for the Nexus Interface

This section explains customized commands for the Nexus debugging interface. You can use these commands in the Command window of the CodeWarrior IDE.

annotate

Use this command to produce an ASCII-formatted output of program trace collected during the debugging session. The output lists all branches and the instructions executed between branches.

Syntax

```
cmdwin::nexus::annotate exec storageName outputFilePath
```

```
cmdwin::nexus::annotate branches storageName outputFilePath
```

Parameters

`storageName`

The type of storage for which to output the annotated program trace.

`outputFilePath`

The path to the ASCII-formatted output file that the `annotate` command creates.

Comments

For more information about this command, select **View** > **Command Window** in the CodeWarrior IDE, then enter this line in the window that appears:

```
help cmdwin::nexus::annotate
```

config

Use this command to produce C source code for the Nexus configuration of the current build target of the default project. The generated source code initializes all relevant registers with the current Nexus settings from the target-settings panels.

Syntax

```
cmdwin::nexus::config xport outputFilePath
```

Parameters

`outputFilePath`

The path to the output file that the `config` command creates.

Comments

For more information about this command, select **View** > **Command Window** in the CodeWarrior IDE, then enter this line in the window that appears:

```
help cmdwin::nexus::config
```

merge

Use this command to merge all trace-data stores into one ASCII-formatted file. This command sorts messages by the index field.

Syntax

```
cmdwin::nexus::merge outputFilePath
```

Parameters

`outputFilePath`

The path to the output file that the `merge` command creates.

Comments

For more information about this command, select **View** > **Command Window** in the CodeWarrior IDE, then enter this line in the window that appears:

```
help cmdwin::nexus::merge
```

xport

Use this command to produce an ASCII dump for specified trace-data stores. This command is not specific to the Nexus interface, and will work for all data stores (such as the Virtual Trace Buffer).

Syntax

```
cmdwin::trace::xport list
cmdwin::trace::xport all outputDirectory
cmdwin::trace::xport databaseName outputFilePath
```

Parameters

`list`

Provides a list of all available stores.

`all outputDirectory`

Exports all available data stores to the specified output directory.

`databaseName`

The name of the data store.

`outputFilePath`

The path to the output file that the `xport` command generates.

Comments

For more information about this command, select **View** > **Command Window** in the CodeWarrior IDE, then enter this line in the window that appears:

```
help cmdwin::trace::xport
```

Cycle Counter in the Simulator

You can get the cumulative machine-cycle count and the machine-instruction count when you use the simulator to debug your program.

NOTE Due to the nature of the simulator, cycle counting is accurate only when executing continuously (rather than single-stepping through instructions). The cycle counter is more useful for profiling than for interactive use.

Cycle Counts in the P2002 Simulator

The cycle-counter register value in the P2002 simulator presents just core cycles without taking into account cache delays. If you want cycle counts to take cache delays into account, try these methods:

- Use the runsim profiler option (`runsim -p`). It provides information per function.
- Use the `time()` call from the program.
- Use the DPU counters from the program.

Working with Cycle Counts

This section explains how to reset cycle counts and how to determine cycles for an algorithm.

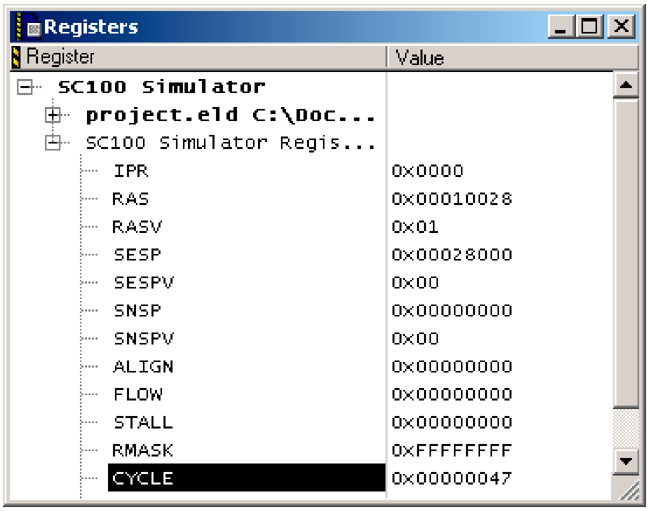
Resetting the Machine Cycle Count

To reset the machine-cycle count, follow these steps:

1. Select **View > Registers**

   The simulator’s **Registers** window (Figure 4.2) appears.

Figure 4.2 Simulator Registers Window



Register	Value
SC100 Simulator	
project.eld C:\Doc...	
SC100 Simulator Regis...	
IPR	0x0000
RAS	0x00010028
RASV	0x01
SESP	0x00028000
SESPV	0x00
SNSP	0x00000000
SNSPV	0x00
ALIGN	0x00000000
FLOW	0x00000000
STALL	0x00000000
RMASK	0xFFFFFFFF
CYCLE	0x00000047

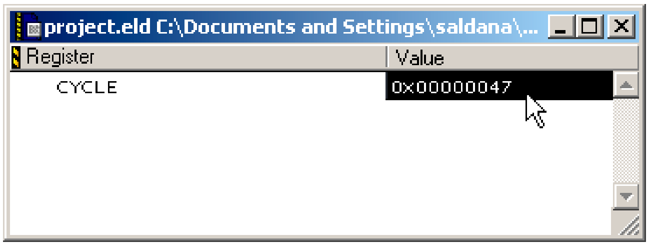
2. Click the tree control of the **SC100 Simulator Registers** node in the tree.

   The node expands to show a list of the simulator's registers.

3. Double-click the `CYCLE` register name.

   The `CYCLE` register and its value appear in a register window (Figure 4.3).

Figure 4.3 The CYCLE Register in the Debugger Register Window



Register	Value
CYCLE	0x00000047

4. Double-click the current value of the `CYCLE` register.

   The value becomes editable.

5. Change the value of the `CYCLE` register to zero (hexadecimal value `0x00000000`).

6. Press the Enter key.

Determine Machine Cycles for an Algorithm

To determine the number of machine cycles the simulator uses to execute a particular algorithm, follow these steps:

1. Set a breakpoint before the beginning of the algorithm.
2. Set a breakpoint after the end of the algorithm.
3. Execute the program to the first breakpoint.
4. Resetting the Machine Cycle Count.
5. Execute the program to the second breakpoint.

   The value `CYCLE` register in the **Registers** window shows the machine cycles used.

Debugging an Already Running Program

You can use the IDE to start a program on target hardware, then disconnect from that program such that it continues to run on the hardware. Later, you can use the **Attach to Process** command of the IDE to reconnect to the program. In this way, you can debug a program already running on the target hardware.

This section shows an example of how to debug an already-running program. In this example, you:

- configure a sample project (the project has a simple incrementing counter that you will observe throughout this section)
- start a process (in this case, the counter) executing on the target hardware
- detach the IDE from the process (leaving the process running on the target hardware)
- attach the IDE to the process (so that the IDE once again can control the process running on the target hardware)
- demonstrate that the process continued to run while detached from the IDE (by examining the counter value)

Configuring a Sample Project

This procedure shows how to configure a sample project to connect to target hardware. In this example, you configure a project to connect to MXC 05 target hardware. The project contains code for a simple `while` loop that will execute on the MXC 05 hardware.

NOTE If you do not have MXC 05 hardware, modify these instructions for use with your particular hardware. Use the `while` loop of the sample MXC 05 project as a guide for coding a similar `while` loop for your particular hardware.

Follow these steps to configure the sample project:

1. In the CodeWarrior IDE, select **File > New**.

   The **New** window appears.
2. In the **Project** page, select **StarCore SDMA Stationery**.
3. Enter a name and specify a location in which to save the new project.
4. Click the **OK** button.

   The **New Project** dialog box appears.
5. Expand the **MXC_05** group.
6. Select **StarCore** under the MXC_05 group.
7. Click the **OK** button.

   A new project window appears in the IDE.
8. Specify a remote connection for the target hardware.
   a. Select **Edit > Preferences**.

      The **IDE Preferences** window appears.
   b. Select **Remote Connections** from the **IDE Prefence Panels** list on the left-hand side of the window.

      The **Remote Connections** preference panel appears in the window.
   c. From the list of remote connections, select the connection for your target hardware.

NOTE If you have not used this preference panel to define a remote connection, see the *IDE User's Guide* for more information.

   d. Click the **OK** button.

      The IDE saves your changes and closes the IDE Preferences window.
9. Configure the project's remote-debugging settings to connect to the target hardware.
   a. Select **Edit > StarCore Stationery Settings**.

      The Target Settings window appears for this project.
   b. Select **Remote Debugging** from the **Target Settings Panels** list on the left-hand side of the window.

      The **Remote Debugging** settings panel appears in the window.

c. Use the **Connection** list box to specify the remote connection that you selected in the **Remote Connections** preference panel (in an earlier step).

NOTE The connection you specify contains the information that the IDE uses to connect to the target hardware.

d. Click the **OK** button.

The IDE saves your changes and closes the Target Settings window.

Starting a Process

This procedure shows how to start an example process running on the target hardware. The example process is a simple `while` loop that increments a counter.

1. Select **Project > Debug**.

   The IDE downloads the `while` loop code to the target hardware. The debugger window appears. The IDE halts execution at the beginning of the main function in the code.

2. Select **Project > Run**.

   The IDE starts execution of the `while` loop code. The counter begins incrementing as it runs on the target hardware.

Detaching from the Process

This procedure shows how to detach the IDE from a process currently under its control. Rather than kill the process, you instruct the IDE to leave the process running on the target hardware.

Follow these steps to detach from the process:

1. Bring forward the debugger window of the process from which you want to detach the IDE.

2. Select **File > Close**.

   A dialog box appears, asking whether you want to resume the process, kill it, or cancel the operation.

3. Click the **Resume** button.

   The IDE closes the debugger window. Rather than terminating the process, the IDE just detaches from that process. The process continues to run on the target hardware. In this example, the counter of the detached process continues to increment as it runs on the target hardware.

Attaching to the Process

This procedure shows how to attach the IDE to the process already running on the target hardware. After you attach the IDE to the process, you can once again debug it. In this example case, to prove that the process continued running on the target hardware, you can stop its execution and examine its most recent counter value.

NOTE Follow these steps after you detach from a process, as explained in the previous procedure. Otherwise, you will not have an already-running program for the IDE to debug.

Follow these steps to attach to the process:

1. Bring forward the IDE.
2. Select **Debug** > **Attach to Process**.

   The IDE attaches to the process running on the target hardware. The debugger window for that process appears in the IDE. The IDE can now control the process once again.
3. Select **Debug** > **Stop**.

   The IDE halts execution of the `while` loop code. Now examine the counter value in the debugger window. Notice that the counter continued incrementing as the code continued running on the target hardware.
4. Select **Debug** > **Kill**.

   The IDE terminates the process running on the target hardware and closes the debugger window.

Debugging an .eld File without a Project

You can use the IDE to debug an `.eld` file that does not have an associated CodeWarrior project. When you debug an `.eld` file this way, the IDE uses a default project named `SC100_Default_Project.xml` to create a new project that contains the file. You can modify the default project to better suit your needs.

Creating a Project for an .eld File

To debug an `.eld` file that does not have an associated CodeWarrior project, follow these steps:

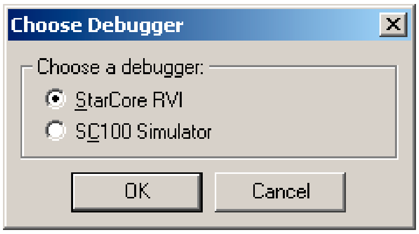
1. Start the CodeWarrior IDE.
2. Select **File** > **Open**.

   A standard dialog box appears.
3. Use the dialog box to open the `.eld` file that you want to debug.

   The **Choose Debugger** dialog box (Figure 4.4) appears.

NOTE Alternatively, instead of using the dialog box to open the `.eld` file, you can drag and drop the file onto the IDE.

Figure 4.4 The Choose Debugger Dialog Box



4. In the Choose Debugger dialog box, click the option button next to the appropriate debugger.
5. Click the **OK** button.

   The IDE creates a CodeWarrior project for the `.eld` file and displays the result in a project window.
6. Select **Edit** > *Target* **Settings**, where *Target* is the name of the current build target.

   The Target Settings window appears.
7. From the **Target Settings Panels** list on the left side of the Target Settings window, select **SC100 Debugger Target**.

   The **SC100 Debugger Target** panel appears on the right side of the **Target Settings** window.
8. Use the **Target** list box of the SC100 Debugger Target panel to specify the appropriate debugging target.

NOTE If your source-code files reside in a different directory than the `.eld` file, use the **Access Paths** panel to specify the paths to the source-code files.

9. In the Target Settings window, click the **Apply** button.

   The IDE saves your configuration.

10. Select **Project** > **Debug**.

    The IDE starts a new debugging session.

NOTE If you debug a `.eld` file without a corresponding CodeWarrior project, the IDE sets the **Build before running** option of the **Build Settings** preference panel to **Never**.

Consequently, if you open another project after debugging a `.eld` file, you must set the **Build before running** option to **Ask** or **Always** before you can build that project.

Modifying the SC100 Default Project File

When you drag and drop an `.eld` file into the IDE, it uses the `SC100_Default_Project.xml` file to create a new project. You can modify this default project file to better suit your needs. For example, you can change the target hardware defined in the file.

To change the target hardware in the default SC100 project file, follow these steps:

1. Select **File** > **Open**.

   A standard dialog box appears.

2. Use the dialog box to select the default SC100 project file, named `SC100_Default_Project.xml`.

NOTE The default SC100 project file resides in this location, where *CodeWarrior* is the path to your CodeWarrior installation:

CodeWarrior`\bin\Plugins\Support\SC100_Default_Project.xml`

3. Click the **Open** button.

   The content of the default SC100 project file appears in a new editor window.

4. Bring forward the editor window that shows the default SC100 project file content.
5. Select **Search** > **Find**.

   The **Find** dialog box appears.

6. Enter this text in the **Find** text box: `SC_Debug_Target_Protocol`

7. Click the **Find** button.

   The IDE finds the specified text in the default SC100 project file. The editor window scrolls to reveal the found text.

NOTE The found text should be part of this line:

```
<SETTING><NAME>SCC_Debug_Target_Protocol</NAME>
<VALUE>Platform 2002 Simulator</VALUE></SETTING>
```

If the found text does is not part of this line, repeatedly select **Search** > **Find Next** until you find the line.

8. Change the `<VALUE>` element text to reflect the target hardware you want to use.

   For example, to debug the project on the i.300-30 target hardware, change the specified line to this:

```
<SETTING><NAME>SC_Debug_Target_Protocol</NAME>
<VALUE>i.300-30 Hardware</VALUE></SETTING>
```

NOTE The text that you enter for the `<VALUE>` element *must* match one of the values listed in the **Target** list box of the **SC100 Debugger Target** settings panel.

9. Select **File** > **Save**.

   The IDE saves your changes to the default SC100 project file.

10. Select **File** > **Close**.
11. Select **File** > **Exit**.
12. Re-launch the CodeWarrior IDE.

    The IDE loads the changes in your modified SC100 default project file.

Now you can start with an executable file and create a project that uses the target hardware you specified in the modified SC100 default project file.

Debugging Multiple ELF Files

The build process does not always place the symbolics information for external code into the application itself. For example, if your application makes calls to code that resides in ROM or a shared library, your application might not always contain symbolics information for that code. The debugger can debug just the code for which it has symbolics information. In order to debug an application that makes calls to external code, you must debug multiple ELF files.

For example, to debug the application that makes calls to code in ROM, you would debug these files:

- the ELF file for the application itself
- the ELF file corresponding to the code that resides in ROM
- any other ELF files corresponding to valid memory that you want to debug

Depending on the symbolics information present in the additional ELF files, you can debug the code corresponding to the external code just as you would debug the code corresponding to the application itself. For example, by debugging multiple ELF files you can step through, examine the stack crawl of, and set breakpoints in the external code.

To debug multiple ELF files, follow these steps:

1. Determine the additional ELF files that you must debug together with the ELF file for the current build target.
2. Select **Edit >** ***Targetname*** **Settings**, where ***Targetname*** is the name of the current build target.

   The Target Settings window appears.
3. From the **Target Settings Panels** list on the left side of the Target Settings window, select **Other Executables**.

   The **Other Executables** panel appears.
4. For *each* additional ELF file:

   a. Click the **Add** button.

      The **Debug Additional Executable** dialog box appears.

   b. Click the **Choose** button.

      The **Choose an Executable to Debug** dialog box appears.

   c. Use the dialog box to select the ELF file.

   d. Click the **Open** button.

      The path to the ELF file appears in the **File location** text box.

   e. Click the **OK** button.

The ELF file appears as a new entry in the Other Executables list.

Initialization File

An *initialization file* is a text file that tells the debugger how to initialize the hardware after reset but before downloading executable code. Use initialization-file commands to write values to various registers, core registers, and memory locations.

To use an initialization file, you must check the **Use Initialization File** checkbox and specify the name of the initialization file in the **SC100 Debugger Target** panel.

This section has these topics:

- Example Initialization File
- Setting the IMMR Value
- Initialization File Commands

Example Initialization File

Listing 4.1 shows part of an initialization file. Initialization files reside in this directory, where *CodeWarrior* is the path to the directory in which you installed your CodeWarrior product:

- Windows:
  CodeWarrior`\StarCore_Support\Initialization_Files\RegisterConfigFiles\`
- Solaris:
  CodeWarrior`/starcore_support/Initialization_Files/RegisterConfigFiles/`

You can customize initialization files to better suit your needs. See "Initialization File Commands" on page 145 for more information.

Listing 4.1 Excerpt from an Initialization File

```
POST-RESET-ON
stopCore

; Initialize M2 RAM

writemem16 0xfff5cc78 0x0036
writemem16 0xfff5cc7c 0x0007
```

Setting the IMMR Value

The Internal Memory Map Register (IMMR) holds the base address for PPC-bus memory-mapped registers. You can write to memory-mapped registers using either the register name or the register address.

The debugger uses the value of the IMMR to determine the address of other PPC-bus memory-mapped registers.

The debugger recognizes a change in the IMMR only if you write to the IMMR by name (not by address) in the initialization file.

If you initialize the IMMR by address, the debugger behaves as if you left the IMMR unchanged. In that case, the debugger uses the default reset value for the IMMR (`0xF0000000`) as the base address for PPC-bus memory-mapped registers when performing all other reads and writes to those registers.

NOTE The only exception to this rule is if you previously changed the value of the IMMR register by name.

Initialization File Commands

Several initialization-file commands exist that let you:

- Write to a register or memory location of a specified device in the JTAG chain
- Write to a register or memory location of a default device (specified in the **SC100 Debugger Target** panel of the current project) in the JTAG device chain

Table 4.3 Initialization File Commands

DSPJTAGClockSpeed	PRE-RESET-ON
PRE-RESET-OFF	POST-RESET-ON
POST-RESET-OFF	setMMRBase16
writeDevicemem8	writeDevicemem16
writeDevicemem32	writeDevicemem64
writemem8	writemem16
writemem32	writemem64
writemmr8	writemmr16
writemmr32	writemmr64
writereg8	writereg16
writereg32	writereg40

DSPJTAGClockSpeed

Sets the JTAG clock speed.

`DSPJTAGClockSpeed` *`speed`*

Parameter

`speed`

Kilohertz rate, such as 1000.

Remarks

The IDE uses the JTAG clock speed you specify to set the debug port clock frequency for the USBTAP or EthernetTAP.

PRE-RESET-ON

Specifies commands to execute before a reset.

`PRE-RESET-ON`

Remarks

Either command `PRE-RESET-OFF` or `POST-RESET-ON` halts this functionality. For example, if the initialization file includes both `PRE-RESET-ON` and `PRE-RESET-OFF`, the debugger executes all the intervening commands before the reset.

If the initialization file does not include either `PRE-RESET-ON` or `POST-RESET-ON`, the debugger executes all the commands before the reset.

PRE-RESET-OFF

Indicates the final command to execute before the reset.

`PRE-RESET-OFF`

POST-RESET-ON

Indicates commands to execute after the reset.

`POST-RESET-ON`

Remarks

The command `POST-RESET-OFF` halts this functionality. For example, if the initialization file includes both `POST-RESET-ON` and `POST-RESET-OFF`, the debugger executes all the intervening commands after the reset.

If the initialization file does not include either `PRE-RESET-ON` or `POST-RESET-ON`, the debugger executes all the commands before the reset.

POST-RESET-OFF

Indicates the final command to be executed after the reset.

```
POST-RESET-OFF
```

setMMRBase16

Provides the debugger with the base value of a Memory Mapped Register (MMR).

`setMMRBase16` *`MMR_register_name base_value`*

Parameters

`MMR_register_name`

The name of a StarCore memory-mapped register. For example, `IMMR`.

NOTE In most cases, you can specify `IMMR` (Internal Memory Map Register) for this parameter.

`base_value`

The base value of the specified MMR. The debugger uses this value to determine the location of the MMR.

Remarks

Use this command to attach the debugger to a running program on a board. This scenario requires the command because, at attach-time, the debugger cannot determine the MMR's base value from a register-initialization file.

Typically, you use the `setMMRBase16` command this way:

1. Display the **SC100 Debugger Target** target settings panel.
2. Check the **Do Not Reset PC** checkbox of this panel.
3. Check the **Use Initialization File** checkbox of this panel and enter the path of the initialization file that contains your `setMMRBase16` command.
4. From the IDE's menu bar, select **Debug** > **Attach**

writeDevicemem8

Writes an 8-bit value to the specified memory location of the specified device on the JTAG chain.

`writeDevicemem8` *`JTAG_index memory_location value`*

Parameters

`JTAG_index`

JTAG chain device identifier.

`memory_location`

Address in device memory.

`value`

Value to write (in decimal or hexadecimal notation).

writeDevicemem16

Writes a 16-bit value to the specified memory location of the specified device on the JTAG chain.

`writeDevicemem16` *`JTAG_index memory_location value`*

Parameters

`JTAG_index`

JTAG chain device identifier.

`memory_location`

Address in device memory.

`value`

Value to write (in decimal or hexadecimal notation).

writeDevicemem32

Writes a 32-bit value to the specified memory location of the specified device on the JTAG chain.

`writeDevicemem32` *`JTAG_index memory_location value`*

Parameters

`JTAG_index`

JTAG chain device identifier.

`memory_location`

Address in device memory.

`value`

Value to write (in decimal or hexadecimal notation).

writeDevicemem64

Writes a 64-bit value to the specified memory location of the specified device on the JTAG chain.

`writeDevicemem64` *`JTAG_index memory_location value`*

Parameters

`JTAG_index`

JTAG chain device identifier.

`memory_location`

Address in device memory.

`value`

Value to write (in decimal or hexadecimal notation).

writemem8

Writes an 8-bit value to memory.

`writemem8` *`memory_location value`*

Parameters

`memory_location`

Address in device memory.

`value`

Value to write (in decimal or hexadecimal notation).

writemem16

Writes a 16-bit value to memory.

`writemem16` *`memory_location value`*

Parameters

`memory_location`

Address in device memory.

`value`

Value to write (in decimal or hexadecimal notation).

writemem32

Writes a 32-bit value to memory.

`writemem32` *`memory_location value`*

Parameters

`memory_location`

Address in device memory.

`value`

Value to write (in decimal or hexadecimal notation).

writemem64

Writes a 64-bit value to memory.

`writemem64` *`memory_location value`*

Parameters

`memory_location`

Address in device memory.

`value`

Value to write (in decimal or hexadecimal notation).

writemmr8

Writes to an 8-bit, memory-mapped register.

`writemmr8` *`memory_mapped_register value`*

Parameters

`memory_mapped_register`

Register name.

`value`

Value to write (in decimal or hexadecimal notation).

writemmr16

Writes to a 16-bit, memory-mapped register.

`writemmr16` *`memory_mapped_register value`*

Parameters

`memory_mapped_register`

Register name.

`value`

Value to write (in decimal or hexadecimal notation).

writemmr32

Writes to a 32-bit, memory-mapped register.

`writemmr32` *`memory_mapped_register value`*

Parameters

`memory_mapped_register`

Register name.

`value`

Value to write (in decimal or hexadecimal notation).

writemmr64

Writes to a 64-bit, memory-mapped register.

`writemmr64` *`memory_mapped_register value`*

Parameters

`memory_mapped_register`

Register name.

`value`

Value to write (in decimal or hexadecimal notation).

writereg8

Writes to an 8-bit core register.

`writereg8` *`core_register value`*

Parameters

`core_register`

Register name.

`value`

Value to write (in decimal or hexadecimal notation).

writereg16

Writes to a 16-bit core register.

`writereg16` *`core_register value`*

Parameters

`core_register`

Register name.

`value`

Value to write (in decimal or hexadecimal notation).

writereg32

Writes to a 32-bit core register.

`writereg32` *`core_register`* `value`

Parameters

`core_register`

Register name.

`value`

Value to write (in decimal or hexadecimal notation).

writereg40

Writes to a 40-bit core register.

`writereg40` *`core_register value`*

Parameters

`core_register`

Register name.

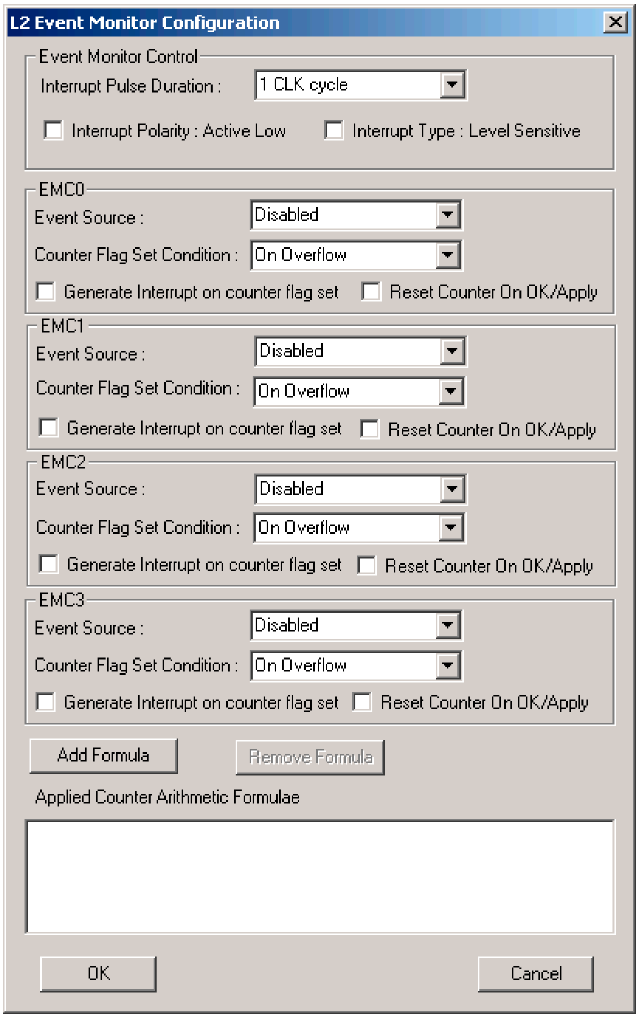
`value`

Value to write (in decimal or hexadecimal notation).

L2 Event Monitor

Use the Level 2 (L2) event monitor to configure counters for L2 cache-controller events and to examine collected counter data. Figure 4.5 shows the L2 Event Monitor Configuration window. Table 4.4 explains each option in the window. To open this window, select **Project > Debug** to start a debugging session, then select **Debug > L2 Event Monitor**.

Figure 4.5 L2 Event Monitor Configuration Window



L2 Event Monitor Configuration

Event Monitor Control

Interrupt Pulse Duration : 1 CLK cycle

☐ Interrupt Polarity : Active Low ☐ Interrupt Type : Level Sensitive

EMC0

Event Source : Disabled

Counter Flag Set Condition : On Overflow

☐ Generate Interrupt on counter flag set ☐ Reset Counter On OK/Apply

EMC1

Event Source : Disabled

Counter Flag Set Condition : On Overflow

☐ Generate Interrupt on counter flag set ☐ Reset Counter On OK/Apply

EMC2

Event Source : Disabled

Counter Flag Set Condition : On Overflow

☐ Generate Interrupt on counter flag set ☐ Reset Counter On OK/Apply

EMC3

Event Source : Disabled

Counter Flag Set Condition : On Overflow

☐ Generate Interrupt on counter flag set ☐ Reset Counter On OK/Apply

Applied Counter Arithmetic Formulae

Table 4.4 L2 Event Monitor Configuration Window Options

Option	Explanation
Interrupt Pulse Duration	This setting takes effect when you clear the Interrupt Type: Level Sensitive checkbox. Use this list box to specify the number of clock cycles for which an interrupt line remains active: The IDE writes the appropriate values to bits 3 through 5 of the Monitor Control (EMMC) register.
Interrupt Polarity: Active Low	Checked—The IDE configures the interrupt as active low. Cleared—The IDE configures the interrupt as active high. The IDE writes the appropriate value to bit 2 of the EMMC register.
Interrupt Type: Level Sensitive	Checked—Interrupts activate on a level change. Cleared—Interrupts activate on an edge change. The IDE writes the appropriate value to bit 1 of the EMMC register.
Event Source	For the corresponding event counter, specify the type of event that the event counter counts. The IDE writes the appropriate values to bits 2 through 5 of the counter configuration register EMCC*x*, where *x* is the specified event counter.
Counter Flag Set Condition	For the corresponding event counter, specify whether to set the counter flag on an overflow condition or on an increment condition. The IDE writes the appropriate value to bit 1 of the EMCC*x* register, where *x* is the specified event counter.
Generate Interrupt on counter flag set	Checked—The corresponding event counter generates an interrupt after meeting the Counter Flag Set Condition that you specify. Cleared—The corresponding event counter does not generate an interrupt after meeting the Counter Flag Set Condition that you specify. The IDE writes the appropriate value to bit 0 of the EMCC*x* register, where *x* is the specified event counter.

Table 4.4 L2 Event Monitor Configuration Window Options (*continued*)

Option	Explanation
Reset Count	For the corresponding event counter, click to reset the counter value. The IDE clears bits 8 through 10 of the EMMC*x* register, where *x* is the specified event counter.
Counter Arithmetics	Click to open the **Counter Arithmetic Formula** window (Figure 4.6). Use this window to create an expression that performs counter arithmetic. In the Counter Arithmetic Formula window, click the **OK** button to add the expression to the Applied Counter Arithmetic Formulae list. Click the **Cancel** button to discard the expression and close the window.
Applied Counter Arithmetic Formulae	Each time you click the **OK** button in the **Counter Arithmetic Formula** window, the expression you created appears in this list. The L2 Event Monitor computes the value of each formula in the list. Each computed value appears in the **L2 Event Counter Report** window (Figure 4.7).
OK	Click to save the current settings, then open the L2 Event Counter Report Window.
Apply	Click to save the current settings.
Cancel	Click to discard the current settings, then close the window.

Figure 4.6 Counter Arithmetic Formula Window

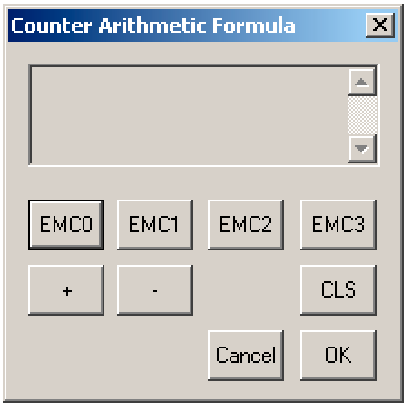
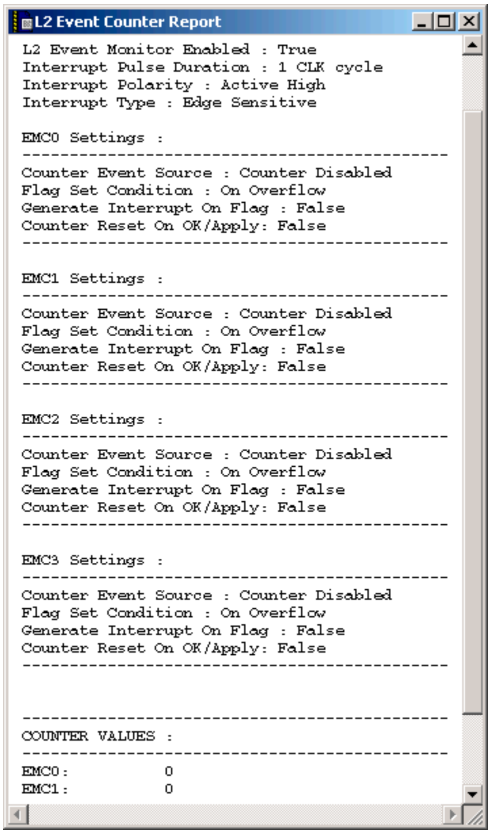


Figure 4.7 shows the L2 Event Counter Report window. This debugger-console window shows the current configuration, counter values, and computed counter arithmetic formulae. The IDE reads the counter registers EMC*x*, where *x* is a specific counter register, to determine the current counter values.

To open the L2 Event Counter Report window, configure settings in the L2 Event Monitor Configuration Window, then click the **OK** button.

The L2 Event Counter Report window can display significantly different values (even `CLK` ticks) depending on whether you enable *L1* cache. For example, if you enable L1 cache and begin with both L1 and L2 cache empty, you might see 0x4000 cycles on the first run and less than 32 at the second run. This situation is normal for hit/miss counters because L1 cache, not L2 cache, services the requests. Also, you might see different values for `CLK` because L2 cache enters idle mode (`CLKEN` inactive) after a number of inactive cycles.

Figure 4.7 L2 Event Counter Report Window



```
L2 Event Monitor Enabled : True
Interrupt Pulse Duration : 1 CLK cycle
Interrupt Polarity : Active High
Interrupt Type : Edge Sensitive

EMC0 Settings :
---------------------------------------------
Counter Event Source : Counter Disabled
Flag Set Condition : On Overflow
Generate Interrupt On Flag : False
Counter Reset On OK/Apply: False
---------------------------------------------

EMC1 Settings :
---------------------------------------------
Counter Event Source : Counter Disabled
Flag Set Condition : On Overflow
Generate Interrupt On Flag : False
Counter Reset On OK/Apply: False
---------------------------------------------

EMC2 Settings :
---------------------------------------------
Counter Event Source : Counter Disabled
Flag Set Condition : On Overflow
Generate Interrupt On Flag : False
Counter Reset On OK/Apply: False
---------------------------------------------

EMC3 Settings :
---------------------------------------------
Counter Event Source : Counter Disabled
Flag Set Condition : On Overflow
Generate Interrupt On Flag : False
Counter Reset On OK/Apply: False
---------------------------------------------


---------------------------------------------
COUNTER VALUES :
---------------------------------------------
EMC0:           0
EMC1:           0
```

Kernel Awareness

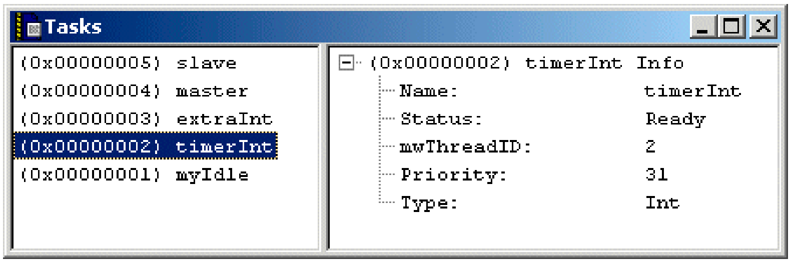
You can specify that you use a supported real-time operating system (RTOS) by using the **Kernel Awareness** list box of the **SC100 Debugger Target** panel to specify an RTOS.

If you do not use an RTOS, use the **Kernel Awareness** list box to specify **None**.

When you debug an application using the Enea OSE RTOS, the IDE displays a menu called OSE. This menu has one command, Task Info.

When you select **OSE > Task Info**, the **Tasks** window (Figure 4.8) appears. This window shows information about all running tasks.

Figure 4.8 Tasks Window



```
(0x00000005) slave            (0x00000002) timerInt Info
(0x00000004) master              Name:            timerInt
(0x00000003) extraInt            Status:          Ready
(0x00000002) timerInt            mwThreadID:      2
(0x00000001) myIdle              Priority:        31
                                 Type:            Int
```

Clicking a task name in the left pane of the Tasks window selects a task. The right pane of the window shows information about the currently selected task.

Table 4.5 explains the Tasks window descriptors.

Table 4.5 Tasks Window Descriptors

Label	Description
Name	The name of the task.
Status	The current status of the task (for example, Ready, Running, or Stopped).
mwThreadID	The ID number assigned to the task thread.

Table 4.5 Tasks Window Descriptors (*continued*)

Label	Description
Priority	An integer value that indicates the priority for running a task.
Type	The type of the task: • Prio — prioritized task • Bkgr — background task • Int — interrupt task • Time — timer interrupt task • Phan — phantom task • Kill — previously killed task • Illg — Invalid (illegal) task • Idle — idle task

Manipulating Target Memory

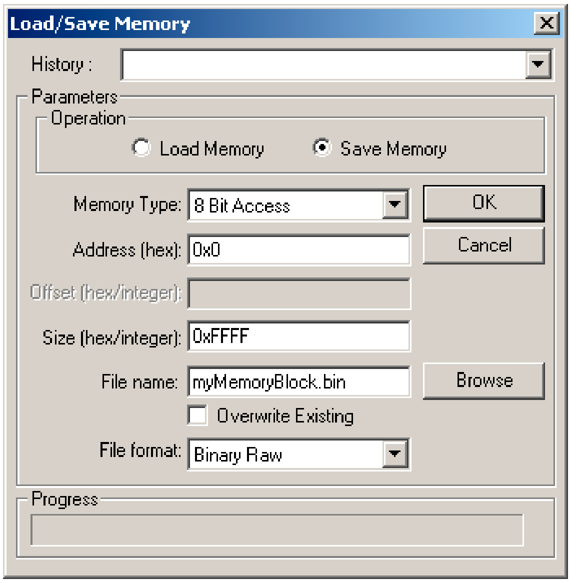
The Debug menu provides two commands that let you manipulate target memory while you debug source code:

- Load/Save Memory
- Fill Memory

Load/Save Memory

To load or save the contents of your target memory, select **Debug** > **Load/Save Memory**. The **Load/Save Memory** dialog box appears. Figure 4.9 shows this dialog box. Table 4.6 explains each dialog-box option.

Figure 4.9 Load/Save Memory Dialog Box



Memory Type: 8 Bit Access
Address (hex): 0x0
Offset (hex/integer):
Size (hex/integer): 0xFFFF
File name: myMemoryBlock.bin
[ ] Overwrite Existing
File format: Binary Raw
Operation: Load Memory [✓ Save Memory]

Table 4.6 Load/Save Memory Dialog Box Options

Option	Explanation
History	This list box shows all previous load- and save-memory operations. Select a previous load or save operation to repeat that action.
Operation	These option buttons let you select between load operations and save operations.
• Load Memory	Click this option button to select a load-memory operation.
• Save Memory	Click this option button to select a save-memory operation.
Memory Type	This list box lets you select the size of the memory units. You can select: • 8-bit access • 16-bit access • 32-bit access
Address	Enter in this text box the memory address where you want to start loading or saving memory.

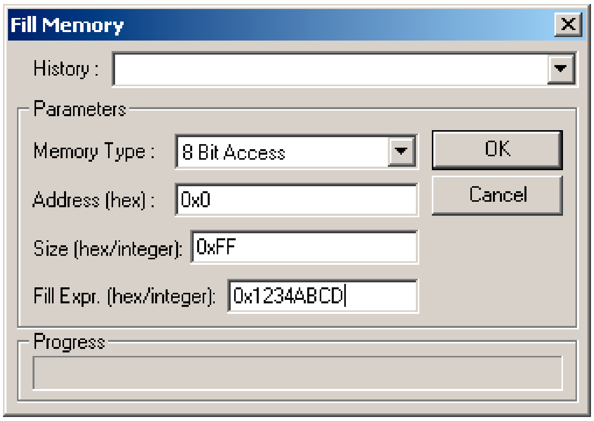
Table 4.6 Load/Save Memory Dialog Box Options (*continued*)

Option	Explanation
Size	The Size text box lets you specify the size in bytes of the memory region you want to load or save.
Filename	Enter in this text box the path to the file you wish to use for the desired memory operation. Alternatively, click the **Browse** button to open a dialog box that you can use to specify the file.
Overwrite Existing	Check this checkbox to overwrite the contents of the file that you specify in the **Filename** text box. Clear this checkbox to append to the contents of the file that you specify in the **Filename** text box. Note that this option is available only when you perform Save operations.
File Format	The File Formats listbox lets you specify the format of the data within the file. You can select from: • Binary Raw—A binary file containing an uninterrupted stream of data • Text Decimal—A text file in which each memory unit is represented by a signed decimal value. • Text Fixed—A text file in which each memory unit is represented by a 32-bit fixed point value. • Text Fractional—A text file in which each memory unit is represented by a floating point number. • Text Hex—A text file in which each memory unit is represented by a hexadecimal value. • Text Unsigned Decimal—A text file in which each memory unit is represented by an unsigned decimal value.

Fill Memory

To fill a memory region of your target board with a given value, select **Debug** > **Fill Memory**. The **Fill Memory** dialog box appears. Figure 4.10 shows this dialog box. Table 4.7 explains each dialog-box option.

Figure 4.10 Fill Memory Dialog Box



History :
Memory Type : 8 Bit Access
Address (hex) : 0x0
Size (hex/integer): 0xFF
Fill Expr. (hex/integer): 0x1234ABCD

Table 4.7 Fill Memory Dialog Box Options

Option	Explanation
History	This list box lists all the previous fill operations. Select a previous fill operation to repeat that action.
Memory Type	This list box lets you select the size of the memory units. You can select: • 8-bit access • 16-bit access • 32-bit access
Address	Enter in this text box the memory address at which to start filling target memory.
Size	Enter in this text box the size (in bytes) of the memory region that you want to fill.
Fill Expr.	Enter in this text box the value with which to fill the memory region

Memory Management Unit Config

P2002 architectures have access to a memory-management unit (MMU). This unit allows different user tasks or *programs* (usually in the context of an RTOS) to use the same areas of memory. To use the MMU, you set up a mapping for data and instruction addresses, then enable address translation. The mapping links virtual addresses to physical addresses. Translation occurs before software acts on the addresses.

You use the MMU Config window to specify the mapping information from virtual addresses to physical addresses. If neither you nor the application (such as the RTOS or user program) specifies the mapping information, processing treats all addresses as physical addresses.

After you specify a memory layout in the `.lcf` file, you use the MMU Config window to specify mapping information. The IDE uses this information to generate MMU initialization code.

For more information about the registers and bitfields that the MMU Config window configures, refer to the *SC140e Platform 2002 User's Guide*.

Opening the MMU Config Window

To open the MMU Config window, select **View** > **MMU Config**.

Generating Initialization Code for the MMU

To have the IDE generate MMU initialization code from the current settings in all pages of the MMU Config window, follow these steps:

1. Use the list box next to the **Generate code** button to specify that you want to generate **C** or **asm** (assembly) initialization code for the MMU.
2. Click the **Generate code** button.

   The IDE validates the configuration, then generates the initialization code. A **Save As** dialog box appears.
3. Use the dialog box to save the generated initialization code to a file.

Writing the Configuration to the MMU

To write the current configuration in all pages to the MMU, click the **Write** button.

Refreshing the MMU Config Window

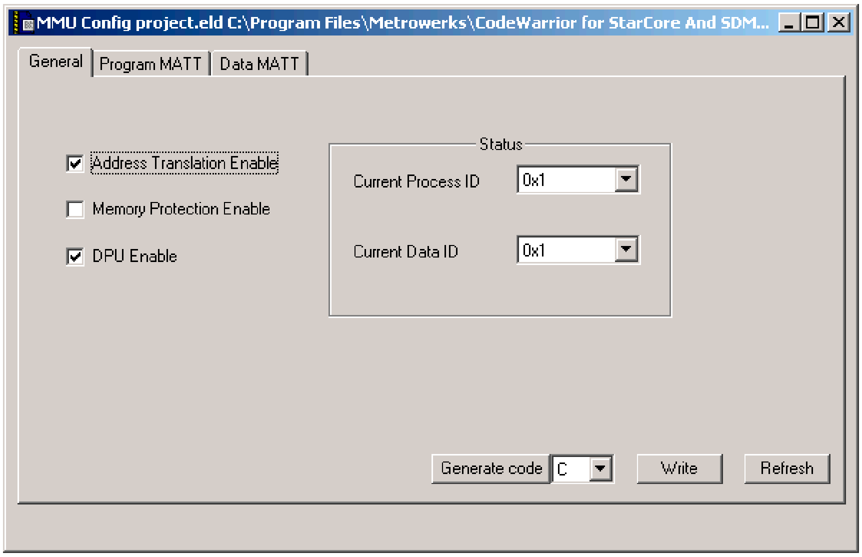
To update the information shown in all pages of the MMU Config window, click the **Refresh** button.

General

Use this page to specify the target hardware for which the IDE generates initialization code.

Figure 4.11 shows the **General** page of the MMU Config window. Table 4.8 explains each panel option.

Figure 4.11 MMU Config Window—General Page


MMU Config project.eld C:\Program Files\Metrowerks\CodeWarrior for StarCore And SDM...

[✓] Address Translation Enable
[ ] Memory Protection Enable
[✓] DPU Enable

Status
Current Process ID: 0x1
Current Data ID: 0x1

Generate code: C

Table 4.8 MMU Config Window Options—General Page

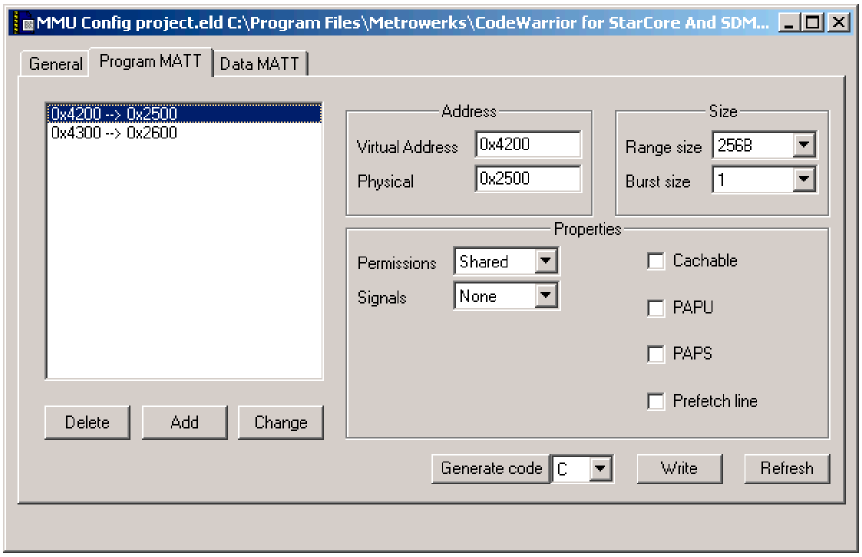
Option	Explanation
Address Translation Enable	Checked—Enables address translation. For example, translation occurs from a virtual address to a physical address. Cleared—Disables address translation. For example, translation does not occur from a virtual address to a physical address. This option corresponds to the Address Translation Enable (ATE) bit of the MMU Control Register (M_CR).
Memory Protection Enable	Checked—Enables protection checking for all enabled segment descriptors. With this option checked, the system consumes more power. Cleared—Disables protection checking for all enabled segment descriptors. With this option cleared, the system consumes less power. This option corresponds to the Memory Protection Enable (MPE) bit of the MMU Control Register (M_CR).
DPU Enable	Checked—Enables the Debug and Profiling Unit (DPU). Cleared—Disables the DPU. With this option cleared, DPU registers are disabled for read and write accesses. This option corresponds to the Debug and Profiling Unit Enable (DPUE) bit of the MMU Control Register (M_CR).
Current Process ID	Specify the currently running process identifier. This option corresponds to the Current Program ID (CPID) bits of the Current Program ID Register (M_CPID).
Current Data ID	Specify the currently running process-data identifier. This option corresponds to the Current Data ID (CDID) bits of the Current Data ID Register (M_CDID).

Program MATT

Use this page to specify descriptors for the program memory-address translation table (MATT).

Figure 4.12 shows the **Program MATT** page of the MMU Config window. Table 4.9 explains each panel option.

Figure 4.12 MMU Config Window—Program MATT Page


MMU Config project.eld C:\Program Files\Metrowerks\CodeWarrior for StarCore And SDM...

0x4200 --> 0x2500
0x4300 --> 0x2600

Address
Virtual Address: 0x4200
Physical: 0x2500

Size
Range size: 256B
Burst size: 1

Properties
Permissions: Shared
Signals: None
☐ Cachable
☐ PAPU
☐ PAPS
☐ Prefetch line

Generate code: C

Table 4.9 MMU Config Window Options—Program MATT Page

Option	Explanation
Descriptor List	Shows the current list of program MATT descriptors.
Virtual Address	Enter the virtual base address of the program segment. This option corresponds to the Program Segment Virtual Base Address and Size (PSVBAS) bits of the Program Segment Descriptor Registers A (M_PSDAx) that configure the virtual base address.

Table 4.9 MMU Config Window Options—Program MATT Page (*continued*)

Option	Explanation
Physical	Enter the most-significant part of the physical address to use for translation. The value that you specify with the Range size list box determines the size of the most-significant part. This option corresponds to the Program Segment Physical Base Address (PSPBA) bits of the Program Segment Descriptor Registers B (M_PSDBx).
Range size	Specify the size of the program segment. This option corresponds to the Program Segment Virtual Base Address and Size (PSVBAS) bits of the Program Segment Descriptor Registers A (M_PSDAx) that configure the size.
Burst Size	Specify the number of transactions (beats) on the bus that the bus controller cannot interrupt. This burst size applies in the region to a cacheable segment. This option corresponds to the Program Burst Size (PBS) bits of the Program Segment Descriptor Registers B (M_PSDBx).
Permissions	Specify whether to share the program segment. This option corresponds to the System/Shared Virtual Program Memory (SSVPM) bit of the Program Segment Descriptor Registers A (M_PSDAx).
Signals	Specify whether to use special attribute signals to activate a cache-coherency snooper on an external system. You can use Global Program 0 (GP0), Global Program 1 (GP1), or both special attribute signals. This option corresponds to the GP0 and GP1 bits of the Program Segment Descriptor Registers B (M_PSDBx).
Cacheable	Checked—Enables caching of the segment in instruction cache. Cleared—Disables caching of the segment in instruction cache. This option corresponds to the Instruction Cacheability (IC) bit of the Program Segment Descriptor Registers A (M_PSDAx).

Table 4.9 MMU Config Window Options—Program MATT Page (*continued*)

Option	Explanation
PAPU	Checked—The segment has user-level fetch permission for program accesses. If you check the PAPS option as well, you disable program-protection checks for this segment. Cleared—The segment does not have user-level fetch permission for program accesses. This option corresponds to the Program Access Permission in User Level (PAPU) bit of the Program Segment Descriptor Registers A (M_PSDAx).
PAPS	Checked—The segment has supervisor-level fetch permission for program accesses. If you check the PAPU option as well, you disable program-protection checks for this segment. Cleared—The segment does not have supervisor-level fetch permission for program accesses. This option corresponds to the Program Access Permission in Supervisor Level (PAPS) bit of the Program Segment Descriptor Registers A (M_PSDAx).
Prefetch line	Checked—Enables the fetch unit's program-line pre-fetch to a segment cacheable in instruction cache. Cleared—Disables the fetch unit's program-line pre-fetch to a segment cacheable in instruction cache. This option corresponds to the Program Pre-fetch Line Enable (PPFE) bit of the Program Segment Descriptor Registers B (M_PSDBx).

Data MATT

Use this page to specify descriptors for the data memory-address translation table.

Figure 4.13 shows the **Data MATT** page of the MMU Config window. Table 4.10 explains each panel option.

Figure 4.13 MMU Config Window—Data MATT Page

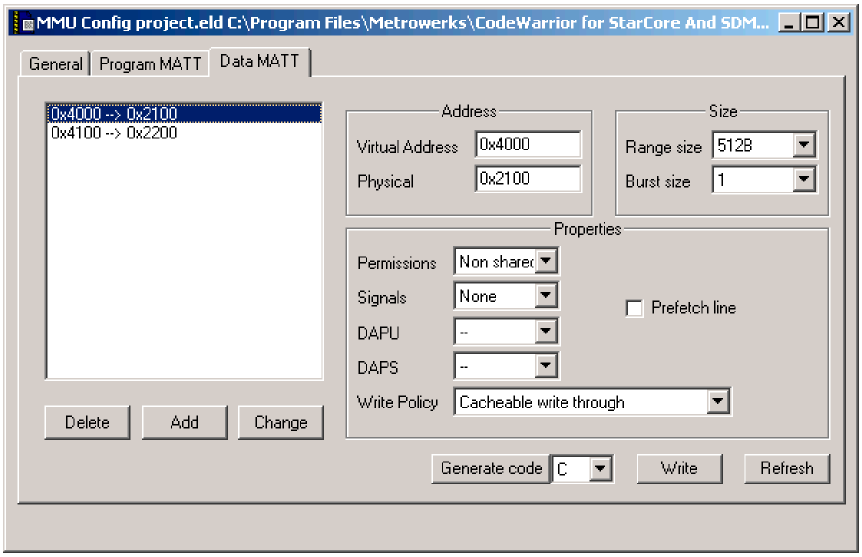


Table 4.10 MMU Config Window Options—Data MATT Page

Option	Explanation
Descriptor List	Shows the current list of data MATT descriptors.
Virtual Address	Enter the virtual base address of the data segment. This option corresponds to the Data Segment Virtual Base Address and Size (DSVBAS) bits of the Data Segment Descriptor Registers A (M_DSDAx) that configure the virtual base address.
Physical	Enter the most-significant part of the physical address to use for translation. The value that you specify with the Range size list box determines the size of the most-significant part. This option corresponds to the Data Segment Physical Base Address (DSPBA) bits of the Data Segment Descriptor Registers B (M_DSDBx).

Table 4.10 MMU Config Window Options—Data MATT Page (*continued*)

Option	Explanation
Range size	Specify the size of the data segment. This option corresponds to the Data Segment Virtual Base Address and Size (DSVBAS) bits of the Data Segment Descriptor Registers A (M_DSDAx) that configure the size.
Burst Size	Specify the number of transactions (beats) on the bus that the bus controller cannot interrupt. This burst size applies in the region to a cacheable segment. This option corresponds to the Data Burst Size (DBS) bits of the Data Segment Descriptor Registers B (M_DSDBx).
Permissions	Specify whether to share the data segment. This option corresponds to the Supervisor/Shared Virtual Data Memory (SSVDM) bit of the Data Segment Descriptor Registers A (M_DSDAx).
Signals	Specify whether to use special attribute signals to activate a cache-coherency snooper on an external system. You can use Global Data 0 (GD0), Global Data 1 (GD1), or both special attribute signals. This option corresponds to the GD0 and GD1 bits of the Data Segment Descriptor Registers B (M_DSDBx).
DAPU	Specify whether to allow user-level read, write, both, or neither types of data access. This option corresponds to the Data Access Permission in User Level (DAPU) bits of the Data Segment Descriptor Registers A (M_DSDAx).
DAPS	Specify whether to allow supervisor-level read (r-), write (-w), both (rw), or neither (--) types of data access. This option corresponds to the Data Access Permission in Supervisor Level (DAPS) bits of the Data Segment Descriptor Registers A (M_DSDAx).

Table 4.10 MMU Config Window Options—Data MATT Page (*continued*)

Option	Explanation
Write Policy	Specify the policy to use for data writes and cache: • **Cacheable write through**—Every write operation updates both cache and higher-level memory. The core does not stall until the access completes, unless a read-after-write condition occurs. The cache updates a cacheable write-through access on a hit only. • **Cacheable write back**—Write operations go through the data cache and the write-back buffer. A write-back operation writes information only to the valid bit resolution in the cache. • **Non Cacheable write through**—The core does not stall until the access completes, unless a read-after-write condition occurs, and the region is non-cacheable. • **Non Cacheable write through with stall**—The core stalls until the access completes, and the region is non-cacheable. This option corresponds to the Data Write Policy (DWP) bits of the Data Segment Descriptor Registers B (M_DSDBx).
Prefetch line	Checked—Enables the fetch unit's data-line pre-fetch to a segment cacheable in data cache. Cleared—Disables the fetch unit's data-line pre-fetch to a segment cacheable in data cache. This option corresponds to Data Pre-fetch Line Enable (DPFE) bit of the Data Segment Descriptor Registers B (M_DSDBx).

Nexus Trace

This section explains how to use the IDE's Nexus trace feature. You can use Nexus trace to complete these tasks:

- use Enhanced On-Chip Emulation (EOnCE™) debugging to trace program flow in specific parts of code
- trace data changes in specific memory ranges
- use Data Acquisition trace to monitor write operations to a specific address
- use Nexus watchpoint messaging to capture watchpoint events

For an example of using Nexus trace, complete these tasks in the order shown:

1. Configuring a Project for Nexus Trace
2. Configuring Nexus Trace Settings in a Project
3. Configuring EOnCE Settings in a Project
4. Collecting and Analyzing Nexus Trace

"Using Nexus Trace Highlighting" on page 190 gives an example of using the IDE's Nexus trace-highlighting feature.

Configuring a Project for Nexus Trace

Follow these steps to configure a project to use Nexus trace:

1. Open a sample project.
   a. Select **Start > Programs > Freescale CodeWarrior > CodeWarrior for StarCore And SDMA *number* > CodeWarrior IDE** from the Windows® task bar, where ***number*** is the CodeWarrior product's version number.

      The IDE starts. The main CodeWarrior window appears.

   b. From the CodeWarrior menu bar, select **File > Open**.

      The **Open** dialog box appears.

   c. Use the **Open** dialog box to find and open the `Nexus` directory at this location, where *CWInstall* is the path to the root level of your CodeWarrior installation:

      CWInstall`\(CodeWarrior_Examples)\StarCore_Examples\Nexus`

   d. Select the `nexus_example.mcp` file.
   e. Click the **Open** button.

      The IDE opens the project. The project window appears, docked at the left side of the main window.

2. Configure the project's remote connection.

NOTE As an example, this step shows how to create a remote connection for MXC-05 hardware. If you have different hardware, use this step as a guide to create an appropriate remote connection.

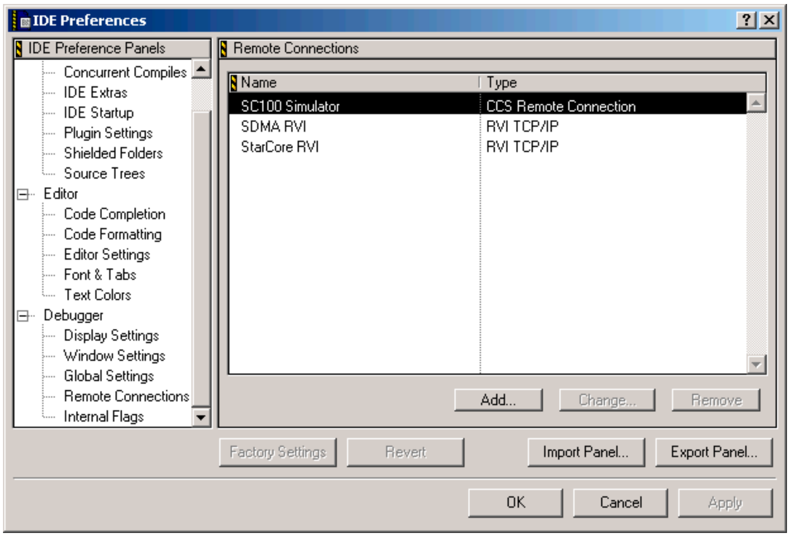
   a. Select **Edit > Preferences** from the CodeWarrior menu bar.

      The **IDE Preferences** window appears.

b. Select **Remote Connections** from the **IDE Preference Panels** list on the left side of the window.

The panel's content appears in the window.

Figure 4.14 IDE Preferences Window — Remote Connections

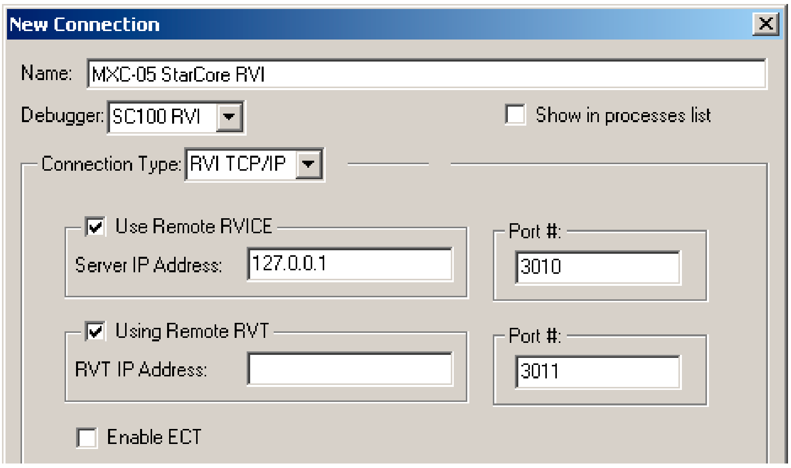


Name	Type
SC100 Simulator	CCS Remote Connection
SDMA RVI	RVI TCP/IP
StarCore RVI	RVI TCP/IP

c. Click the **Add** button.

The **New Connection** dialog box appears.

Figure 4.15 New Connection Dialog Box



Name: MXC-05 StarCore RVI
Debugger: SC100 RVI
Show in processes list ☐
Connection Type: RVI TCP/IP
Use Remote RVICE ☑
Server IP Address: 127.0.0.1
Port #: 3010
Using Remote RVT ☑
RVT IP Address:
Port #: 3011
Enable ECT ☐

d. In the **Name** text box, enter: `MXC-05 StarCore RVI`

e. Use the **Debugger** list box to specify: **SC100 RVI**

f. In the **Server IP Address** text box, enter the IP address of the RVICE server.

g. Click the **OK** button.

The IDE saves the new remote-connection configuration. The name of the new remote connection appears in the panel's list.

h. Click the **OK** button.

The IDE Preferences window closes.

3. Configure remote debugging for the project.

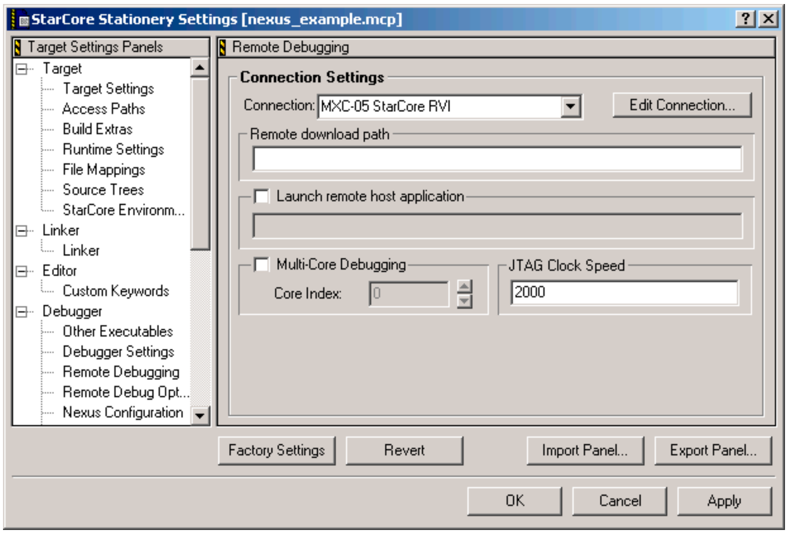
a. Select **Edit > StarCore Stationery Settings** from the CodeWarrior menu bar.

The Target settings window appears.

b. Select **Remote Debugging** from the **Target Settings Panels** list on the left side of the window.

The panel's content appears in the window.

Figure 4.16 Target Settings Window — Remote Debugging



StarCore Stationery Settings [nexus_example.mcp]

Remote Debugging

Connection Settings

Connection: MXC-05 StarCore RVI

Remote download path: (empty)

Launch remote host application: unchecked

Multi-Core Debugging: unchecked; Core Index: 0

JTAG Clock Speed: 2000

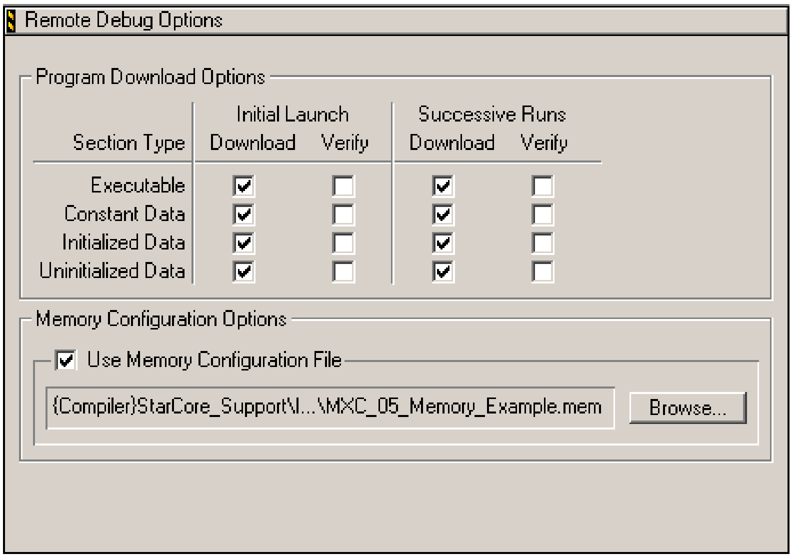
c. Use the **Connection** list box to specify the remote connection that you created in step 2.

4. Configure remote-debugging options for the project.

a. Select **Remote Debug Options** from the **Target Settings Panels** list on the left side of the window.

   The panel's content appears in the window.

Figure 4.17 Target Settings Window — Remote Debug Options



Remote Debug Options

Program Download Options

Section Type	Initial Launch Download	Initial Launch Verify	Successive Runs Download	Successive Runs Verify
Executable	✓		✓	
Constant Data	✓		✓	
Initialized Data	✓		✓	
Uninitialized Data	✓		✓	

Memory Configuration Options

✓ Use Memory Configuration File

{Compiler}StarCore_Support\I...\MXC_05_Memory_Example.mem

b. Check the **Use Memory Configuration File** checkbox.

c. Click the **Browse** button.

   The **Choose the Debugger Memory Configuration File** dialog box appears.

d. Use the dialog box to navigate to

   CWInstall`\StarCore_Support\Initialization_Files\`
   `MemoryConfigFiles`

   where *CWInstall* is the path to the root level of your CodeWarrior installation.

e. Use the dialog box to select the appropriate memory-configuration file for your hardware. For example, select `MXC_05_Memory_Example.mem` for MXC-05 hardware.

f. Use the **Path Type** list box to specify: **Compiler Relative**

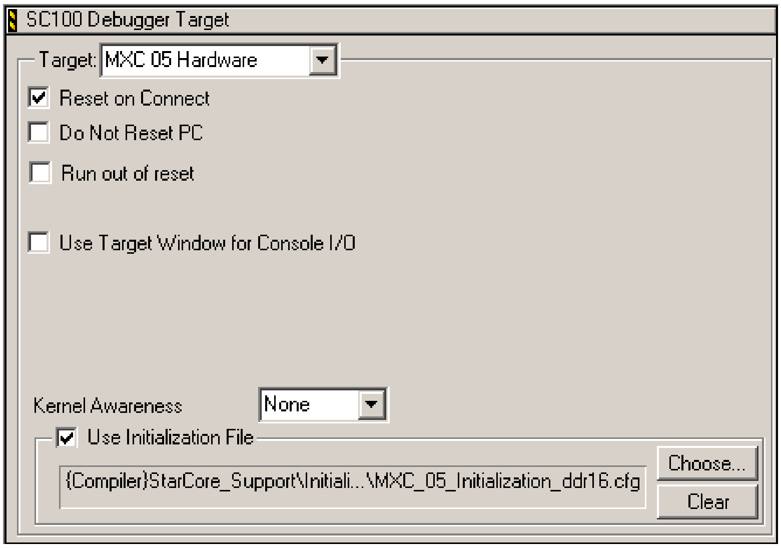
g. Click the **Open** button.

   The path to the selected memory-configuration file now appears to the left of the **Browse** button.

5. Configure the SC100 debugger target for the project.
   a. Select the **SC100 Debugger Target** from the **Target Settings Panels** list on the left side of the window.

      The panel's content appears in the window.

Figure 4.18 Target Settings Window — SC100 Debugger Target


SC100 Debugger Target

Target: MXC 05 Hardware

[✓] Reset on Connect

[ ] Do Not Reset PC

[ ] Run out of reset

[ ] Use Target Window for Console I/O

Kernel Awareness: None

[✓] Use Initialization File

{Compiler}StarCore_Support\Initiali...\MXC_05_Initialization_ddr16.cfg

   b. Use the **Target** list box to specify the appropriate target for your hardware.
   c. Check the **Use Initialization File** checkbox.
   d. Click the **Choose** button.

      The **Choose the StarCore Register Init File** dialog box appears.
   e. Use dialog box to navigate to

      CWInstall`\StarCore_Support\Initialization_Files\`
      `RegisterConfigFiles`

      where *CWInstall* is the path to the root level of your CodeWarrior installation.
   f. Use the dialog box to select the appropriate register-initialization file for your hardware. For example, select

      ```
      MXC_05\MXC_05_Initialization_ddr16.cfg
      ```

      for MXC-05 hardware.

g. Use the **Path Type** list box to specify: **Compiler Relative**

h. Click the **Open** button.

The path to the selected initialization file now appears to the left of the **Choose** button.

6. Build and debug the project.

a. Select **Project** > **Make** from the CodeWarrior menu bar.

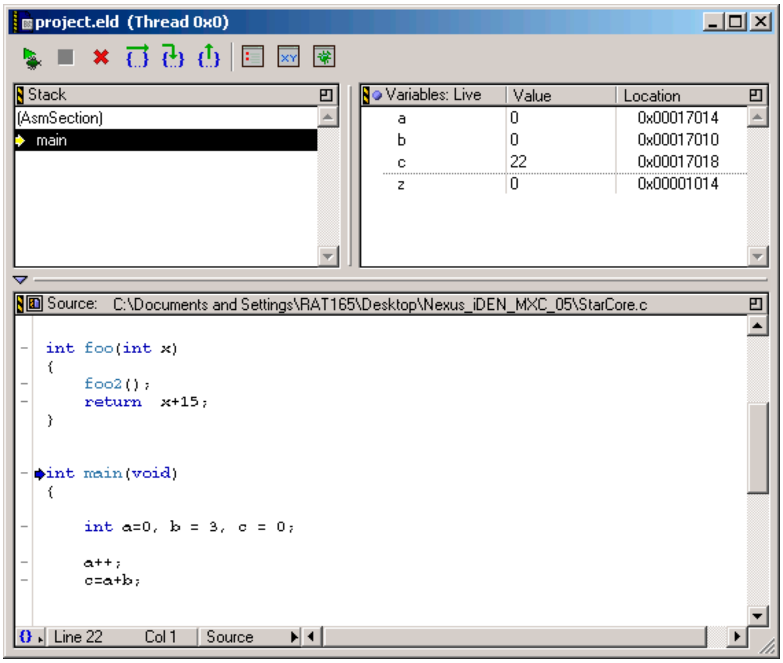
The IDE builds (compiles, assembles, and links) source code, then generates an executable program.

b. Select **Project** > **Debug**.

The debugging session starts. A thread window appears.

NOTE The thread window shows the relationship among variables, their values, and their addresses.

Figure 4.19 Thread Window



Variables: Live	Value	Location
a	0	0x00017014
b	0	0x00017010
c	22	0x00017018
z	0	0x00001014

```
int foo(int x)
{
    foo2();
    return  x+15;
}

int main(void)
{

    int a=0, b = 3, c = 0;

    a++;
    c=a+b;
```

Configuring Nexus Trace Settings in a Project

After you complete the steps in “Configuring a Project for Nexus Trace” on page 172, you can configure Nexus trace settings. Follow these steps:

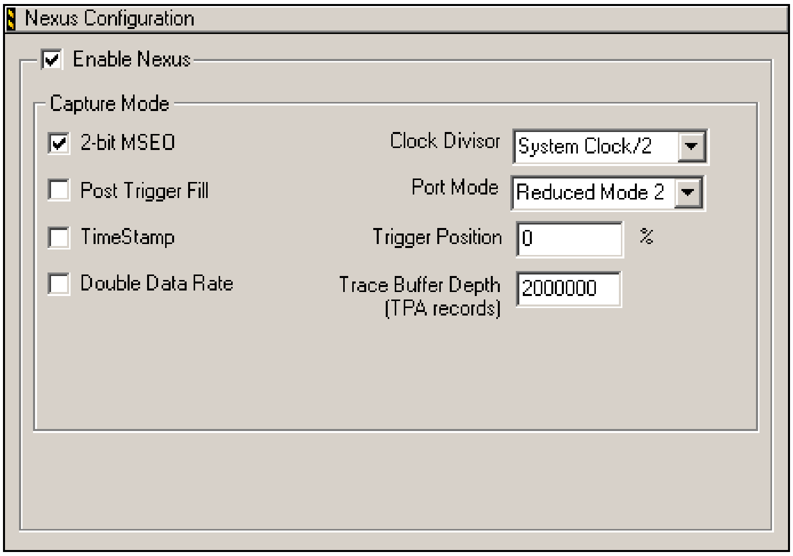
1. Specify the Nexus configuration.
   a. Select **Edit > StarCore Stationery Settings** from the CodeWarrior menu bar.

      The Target Settings window appears.

   b. Position the Target Settings window so that the thread window’s variable values and addresses remain visible.
   c. Select **Nexus Configuration** from the **Target Settings Panels** list on the left side of the window.

      The panel’s content appears in the window.

Figure 4.20 Target Settings Window — Nexus Configuration



Nexus Configuration

[✓] Enable Nexus

Capture Mode

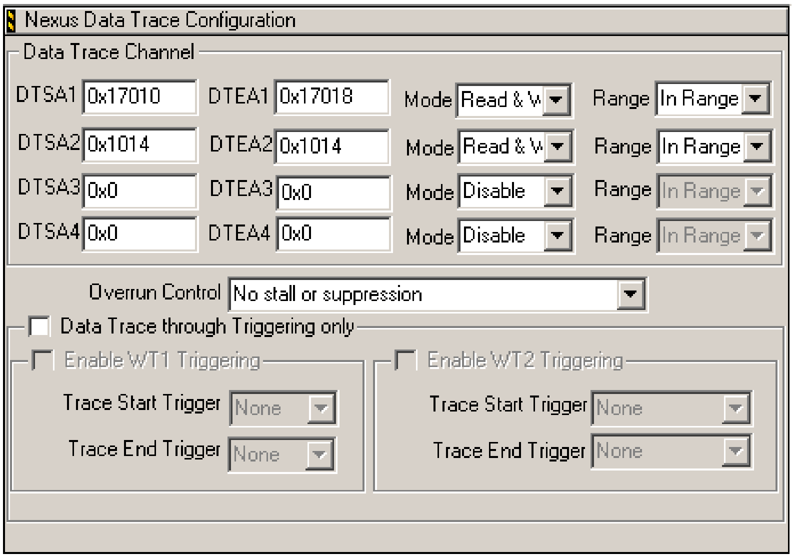
[✓] 2-bit MSEO
[ ] Post Trigger Fill
[ ] TimeStamp
[ ] Double Data Rate

Clock Divisor: System Clock/2
Port Mode: Reduced Mode 2
Trigger Position: 0 %
Trace Buffer Depth (TPA records): 2000000

   d. Check the **Enable Nexus** checkbox.
2. Specify the Nexus data trace configuration.
   a. Select **Nexus Data Trace Configuration** from the **Target Settings Panels** list on the left side of the window.

      The panel’s content appears in the window.

Figure 4.21 Target Settings Window — Nexus Data Trace Configuration



Nexus Data Trace Configuration

Data Trace Channel

DTSA1	0x17010	DTEA1	0x17018	Mode	Read & W	Range	In Range
DTSA2	0x1014	DTEA2	0x1014	Mode	Read & W	Range	In Range
DTSA3	0x0	DTEA3	0x0	Mode	Disable	Range	In Range
DTSA4	0x0	DTEA4	0x0	Mode	Disable	Range	In Range

Overrun Control: No stall or suppression

☐ Data Trace through Triggering only

☐ Enable WT1 Triggering
Trace Start Trigger: None
Trace End Trigger: None

☐ Enable WT2 Triggering
Trace Start Trigger: None
Trace End Trigger: None

b. Use the thread window to determine the range of addresses for the variables `a`, `b`, and `c`.

c. Enter the starting address in the **DTSA1** text box.

d. Enter the ending address in the **DTEA1** text box.

e. Enter the address of the variable `z` in the **DTSA2** and **DTEA2** text boxes.

f. Use the **Mode** list boxes to specify **Read & Write** for both the **DTSA1**/**DTEA1** pair and the **DTSA2**/**DTEA2** pair.

The IDE configures the Nexus trace to generate trace messages for read events and write events.

g. Use the **Range** list boxes to specify **In Range** for both the **DTSA1**/**DTEA1** pair and the **DTSA2**/**DTEA2** pair.

The IDE configures the Nexus trace to monitor just the addresses within the specified ranges.

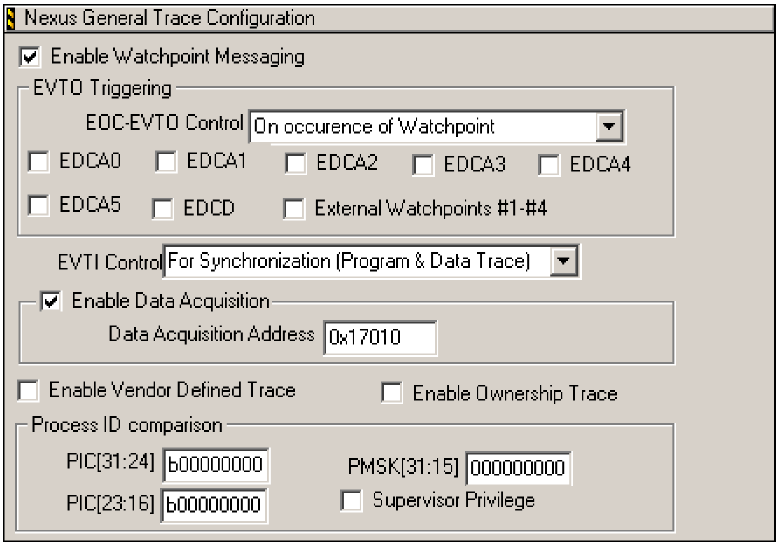
NOTE Restricting the range of addresses for data-trace channels and data trace through EOnCE triggering is a good way to collect just the most relevant data.

3. Specify the Nexus general trace configuration.

   a. Select **Nexus General Trace Configuration** from the **Target Settings Panels** list on the left side of the window.

      The panel's content appears in the window.

Figure 4.22 Target Settings Window — Nexus General Trace Configuration


Nexus General Trace Configuration

[✓] Enable Watchpoint Messaging

EVTO Triggering
- EOC-EVTO Control: On occurence of Watchpoint
- [ ] EDCA0 [ ] EDCA1 [ ] EDCA2 [ ] EDCA3 [ ] EDCA4
- [ ] EDCA5 [ ] EDCD [ ] External Watchpoints #1-#4

EVTI Control: For Synchronization (Program & Data Trace)

[✓] Enable Data Acquisition
- Data Acquisition Address: 0x17010

[ ] Enable Vendor Defined Trace [ ] Enable Ownership Trace

Process ID comparison
- PIC[31:24]: b00000000
- PIC[23:16]: b00000000
- PMSK[31:15]: 000000000
- [ ] Supervisor Privilege

   b. Check the **Enable Watchpoint Messaging** checkbox.

      The IDE configures Nexus debugging to generate watchpoint messages.

   c. Check the **Enable Data Acquisition** checkbox.

      The IDE configures Nexus debugging to generate a data-acquisition message for each write to the specified **Data Acquisition Address**.

   d. Use the thread window to determine the address of the variable `b`.

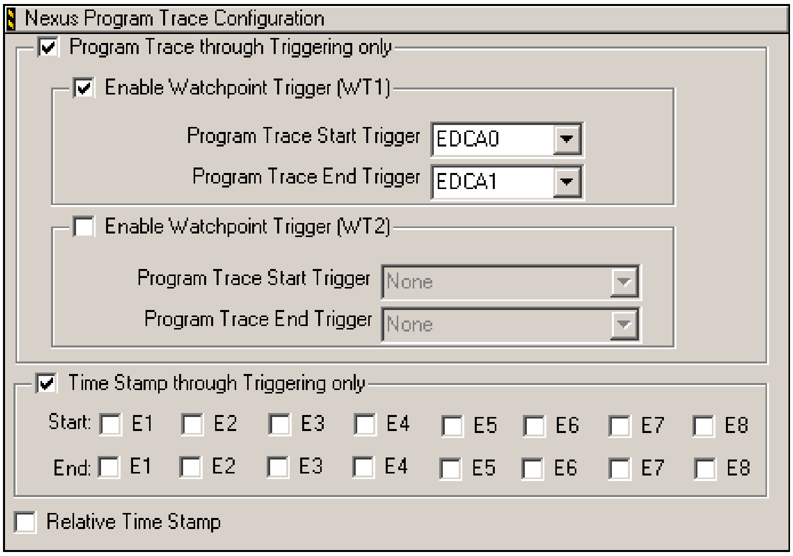
   e. Enter the address of the variable `b` in the **Data Acquisition Address** text box.

4. Specify the Nexus program trace configuration.

   a. Select **Nexus Program Trace Configuration** from the **Target Settings Panels** list on the left side of the window.

      The panel's content appears in the window.

Figure 4.23 Target Settings Window — Nexus Program Trace Configuration



Nexus Program Trace Configuration

- [✓] Program Trace through Triggering only
  - [✓] Enable Watchpoint Trigger (WT1)
    - Program Trace Start Trigger: EDCA0
    - Program Trace End Trigger: EDCA1
  - [ ] Enable Watchpoint Trigger (WT2)
    - Program Trace Start Trigger: None
    - Program Trace End Trigger: None
- [✓] Time Stamp through Triggering only
  - Start: [ ] E1 [ ] E2 [ ] E3 [ ] E4 [ ] E5 [ ] E6 [ ] E7 [ ] E8
  - End: [ ] E1 [ ] E2 [ ] E3 [ ] E4 [ ] E5 [ ] E6 [ ] E7 [ ] E8
- [ ] Relative Time Stamp

b. Check the **Program Trace through Triggering only** checkbox.

The IDE configures Nexus debugging to generate program trace based on EOnCE trigger events.

c. Check the **Enable Watchpoint Trigger (WT1)** checkbox.

The IDE configures Nexus debugging to collect program trace based on EOnCE trigger events on WT1.

d. Use the **Program Trace Start Trigger** list box to specify **EDCA0**.

The IDE configures program trace to begin on the watchpoint EDCA0.

e. Use the **Program Trace End Trigger** list box to specify **EDCA1**.

The IDE configures program trace to end on the watchpoint EDCA1.

f. Check the **Time Stamp through Triggering only** checkbox.

The IDE configures Nexus debugging to generate timestamps just for Embedded Cross Trigger (ECT) events.

NOTE Restricting the range of addresses for program trace through EOnCE triggering is a good way to collect just the most relevant data.

g. Click the **OK** button.

The Target Settings window closes. The thread window remains open.

Configuring EOnCE Settings in a Project

After you complete the steps in “Configuring Nexus Trace Settings in a Project” on page 178, you can configure EOnCE settings. Follow these steps:

1. Open the EOnCE settings file.
   a. Select **Debug** > **EOnCE** > **Open EOnCE Configuration**.

      The **Open** dialog box appears.

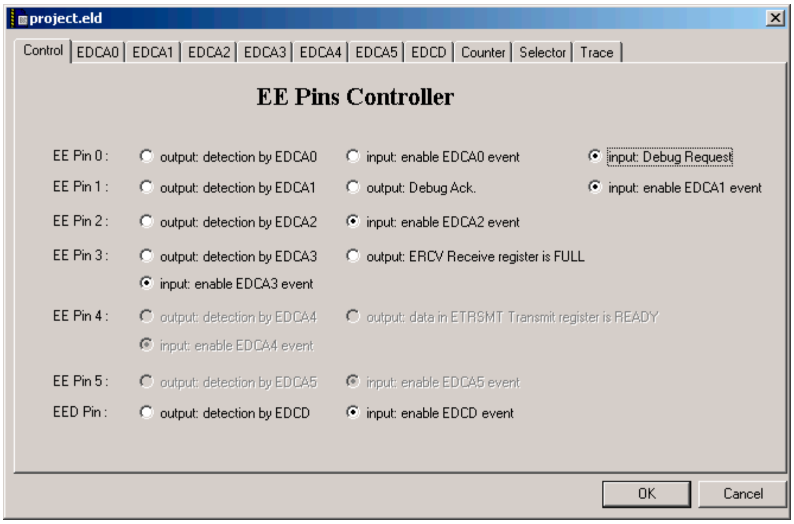
   b. Use the dialog box to find and open the `Nexus` directory at this location, where *CWInstall* is the path to the root level of your CodeWarrior installation:

      CWInstall`\(CodeWarrior_Examples)\StarCore_Examples\Nexus`

   c. Select the `nexus_eonce_config.cfg` file.
   d. Click the **Open** button.

      The EOnCE configuration window appears.

Figure 4.24 EOnCE Configuration Window



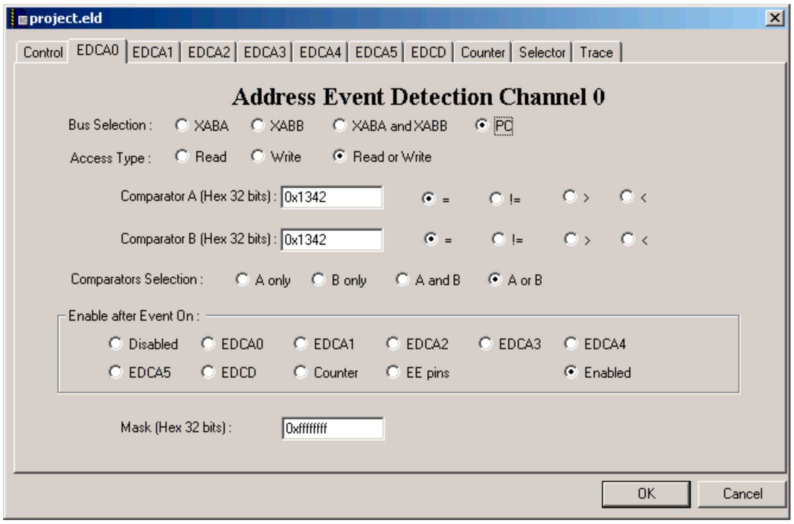
EE Pins Controller

EE Pin 0 :	output: detection by EDCA0	input: enable EDCA0 event	[✓ input: Debug Request]
EE Pin 1 :	output: detection by EDCA1	output: Debug Ack.	[✓ input: enable EDCA1 event]
EE Pin 2 :	output: detection by EDCA2	[✓ input: enable EDCA2 event]	
EE Pin 3 :	output: detection by EDCA3	output: ERCV Receive register is FULL	
	[✓ input: enable EDCA3 event]		
EE Pin 4 :	output: detection by EDCA4	output: data in ETRSMT Transmit register is READY	
	[✓ input: enable EDCA4 event]		
EE Pin 5 :	output: detection by EDCA5	[✓ input: enable EDCA5 event]	
EED Pin :	output: detection by EDCD	[✓ input: enable EDCD event]	

2. Configure the **EDCA0** page.
   a. Click the **EDCA0** tab.

      The **Address Event Detection Channel 0** page appears in the window.

Figure 4.25 EOnCE Configuration Window — EDCA0 Page


project.eld

Control | EDCA0 | EDCA1 | EDCA2 | EDCA3 | EDCA4 | EDCA5 | EDCD | Counter | Selector | Trace

Address Event Detection Channel 0

Bus Selection : XABA XABB XABA and XABB [✓ PC]

Access Type : Read Write [✓ Read or Write]

Comparator A (Hex 32 bits) : 0x1342 [✓ =] != > <

Comparator B (Hex 32 bits) : 0x1342 [✓ =] != > <

Comparators Selection : A only B only A and B [✓ A or B]

Enable after Event On : Disabled EDCA0 EDCA1 EDCA2 EDCA3 EDCA4 EDCA5 EDCD Counter EE pins [✓ Enabled]

Mask (Hex 32 bits) : 0xffffffff

b. In the **Bus Selection** group, click the **PC** option button.

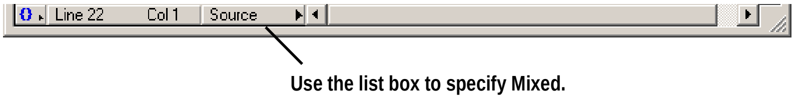
The IDE configures EOnCE settings to detect events based on source-code instruction addresses.

c. In the **Access Type** group, click the **Read or Write** option button.

The IDE configures EOnCE settings to detect both read and write accesses.

d. At the bottom of the thread window, use the source-view list box to view **Mixed** code (both source code and the corresponding assembly-language instructions).

Figure 4.26 Thread Window — Viewing Mixed



e. Use the thread window to scroll to this comment line in the source code:

```
// trigger program trace ON (EDCA0)
```

f. Use the thread window to determine the address of the assembly-language instruction that appears *just before* the comment line.

g. In the EOnCE Configuration window, enter the address (in hexadecimal notation) into the **Comparator A** and **Comparator B** text boxes.

h. Click the = option button next to both the **Comparator A** and **Comparator B** text boxes.

i. In the **Comparators Selection** group, click the **A or B** option button.

j. In the **Enable after Event On** group, click the **Enabled** option button.

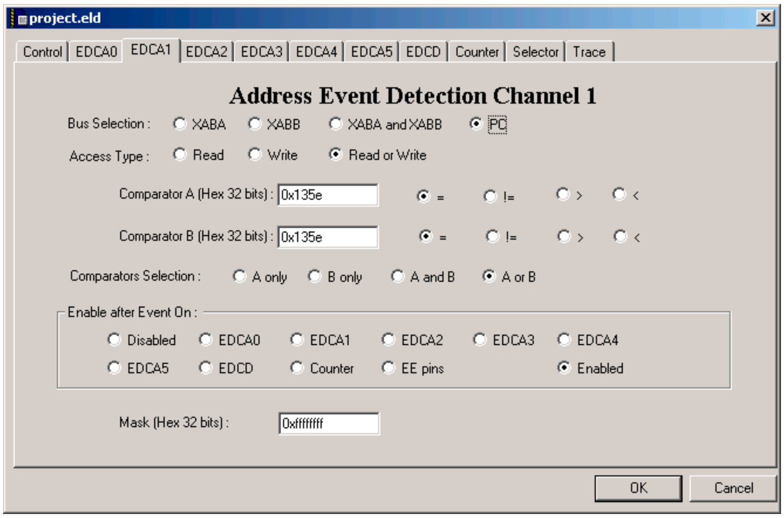
The IDE configures EOnCE debugging to always compare Comparator A to Comparator B (and not rely on a specific event to trigger the comparison).

3. Configure the **EDCA1** page.

a. Click the **EDCA1** tab.

The **Address Event Detection Channel 1** page appears in the window.

Figure 4.27 EOnCE Configuration Window — EDCA1 Page



project.eld

Control | EDCA0 | EDCA1 | EDCA2 | EDCA3 | EDCA4 | EDCA5 | EDCD | Counter | Selector | Trace

Address Event Detection Channel 1

Bus Selection : XABA XABB XABA and XABB [✓ PC]

Access Type : Read Write [✓ Read or Write]

Comparator A (Hex 32 bits) : 0x135e [✓ =] != > <

Comparator B (Hex 32 bits) : 0x135e [✓ =] != > <

Comparators Selection : A only B only A and B [✓ A or B]

Enable after Event On : Disabled EDCA0 EDCA1 EDCA2 EDCA3 EDCA4 EDCA5 EDCD Counter EE pins [✓ Enabled]

Mask (Hex 32 bits) : 0xffffffff

b. In the **Bus Selection** group, click the **PC** option button.

The IDE configures EOnCE settings to detect events based on source-code instruction addresses.

c. In the **Access Type** group, click the **Read or Write** option button.

The IDE configures EOnCE settings to detect both read and write accesses.

d. Use the thread window to scroll to this comment line in the source code:

```
// trigger program trace OFF (EDCA1)
```

e. Use the thread window to determine the address of the assembly-language instruction that appears *just after* the comment line.

f. In the EOnCE Configuration window, enter the address (in hexadecimal notation) into the **Comparator A** and **Comparator B** text boxes.

g. Click the = option button next to both the **Comparator A** and **Comparator B** text boxes.

h. In the **Comparators Selection** group, click the **A or B** option button.

i. In the **Enable after Event On** group, click the **Enabled** option button.

The IDE configures EOnCE debugging to always compare Comparator A to Comparator B (and not rely on a specific event to trigger the comparison).

j. Click the **OK** button.

The EOnCE configuration window disappears. A dialog box appears, prompting you to choose whether to overwrite the current EOnCE settings file.

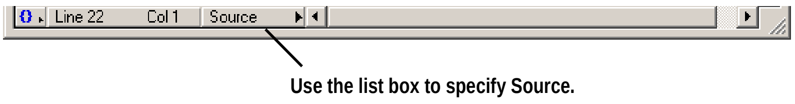
k. Click the **Cancel** button.

The dialog box closes.

NOTE By clicking the **Cancel** button, the settings in the EOnCE configuration window apply just to the current debugging session. If you click the **OK** button instead, the IDE overwrites the settings in the `project.eld.eonce` file with those you just specified in the EOnCE Configuration window.

l. At the bottom of the thread window, use the source-view list box to view **Source** code.

Figure 4.28 Thread Window — Viewing Source



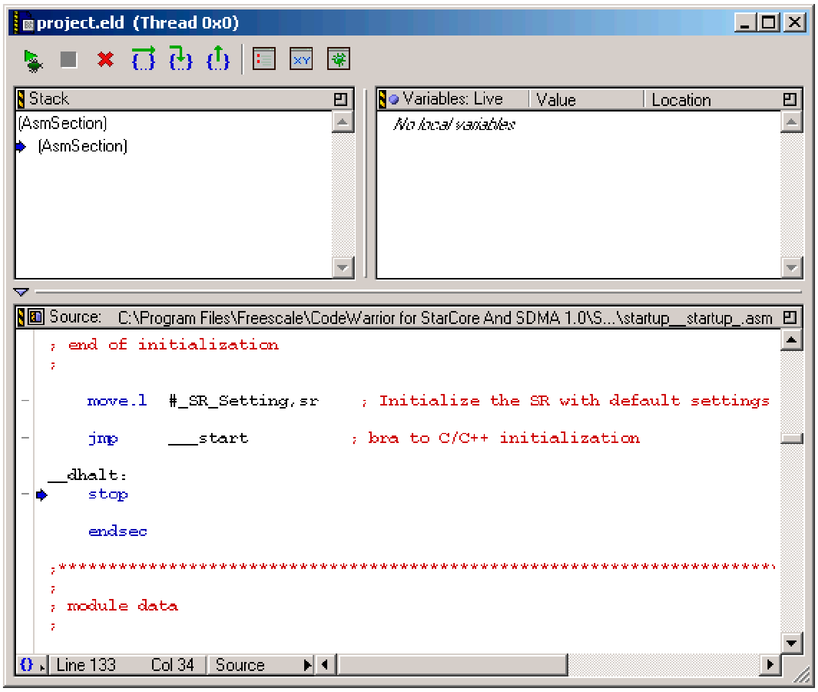
Collecting and Analyzing Nexus Trace

After you complete the steps in "Configuring EOnCE Settings in a Project" on page 182, you can use the IDE to collect and analyze Nexus trace. Follow these steps:

1. From the CodeWarrior menu bar, select **Project** > **Run**.

   The IDE runs the project to completion. A new thread window appears.

Figure 4.29 Thread Window



(AsmSection)
(AsmSection)

No local variables

Source: C:\Program Files\Freescale\CodeWarrior for StarCore And SDMA 1.0\S...\startup__startup_.asm

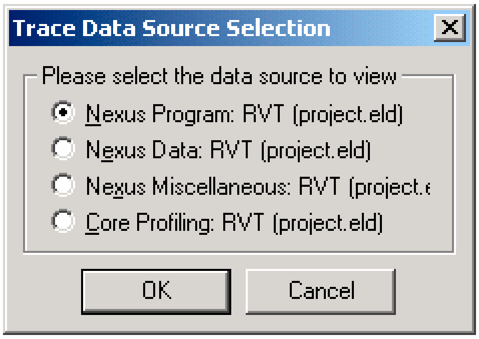
```
; end of initialization
;

    move.l   #_SR_Setting,sr     ; Initialize the SR with default settings

    jmp      ___start           ; bra to C/C++ initialization

__dhalt:
    stop

    endsec

;************************************************************************
;
; module data
;
```

Line 133 Col 34 Source

2. Examine the Nexus program trace.
   a. Select **Data > View Trace**.

      The **Trace Data Source Selection** dialog box appears.

Figure 4.30 Trace Data Source Selection Dialog Box


Trace Data Source Selection

Please select the data source to view

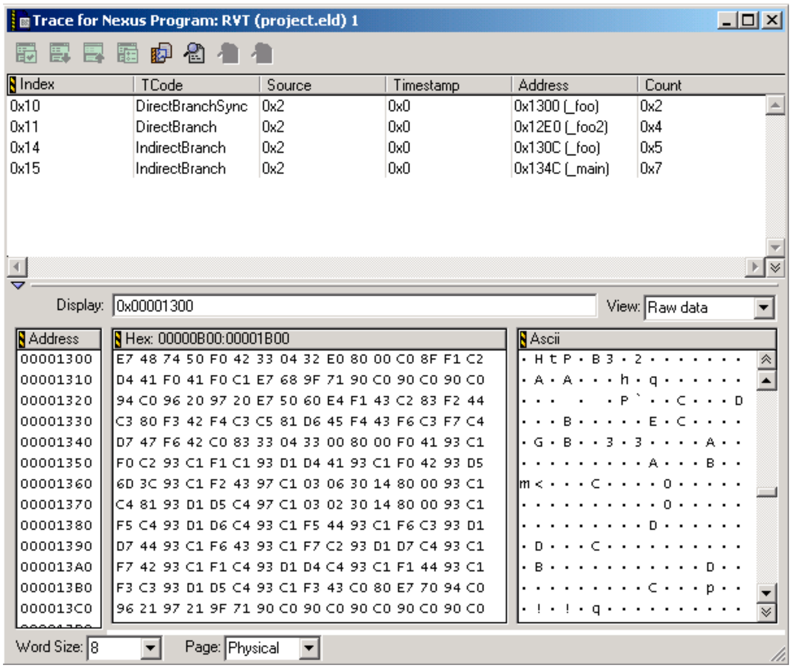
- (●) Nexus Program: RVT (project.eld)
- ( ) Nexus Data: RVT (project.eld)
- ( ) Nexus Miscellaneous: RVT (project.e
- ( ) Core Profiling: RVT (project.eld)

b. Select the **Nexus Program** option button.

c. Click the **OK** button.

The **Trace for Nexus Program** window appears.

Figure 4.31 Trace for Nexus Program Window



Index	TCode	Source	Timestamp	Address	Count
0x10	DirectBranchSync	0x2	0x0	0x1300 (_foo)	0x2
0x11	DirectBranch	0x2	0x0	0x12E0 (_foo2)	0x4
0x14	IndirectBranch	0x2	0x0	0x130C (_foo)	0x5
0x15	IndirectBranch	0x2	0x0	0x134C (_main)	0x7

Display: 0x00001300 View: Raw data

```
Address   Hex: 00000B00:00001B00                               Ascii
00001300  E7 48 74 50 F0 42 33 04 32 E0 80 00 C0 8F F1 C2      · H t P · B 3 · 2 · · · · · · ·
00001310  D4 41 F0 41 F0 C1 E7 68 9F 71 90 C0 90 C0 90 C0      · A · A · · · h · q · · · · · ·
00001320  94 C0 96 20 97 20 E7 50 60 E4 F1 43 C2 83 F2 44      · · ·   ·   · P ` · · C · · · D
00001330  C3 80 F3 42 F4 C3 C5 81 D6 45 F4 43 F6 C3 F7 C4      · · · B · · · · · E · C · · · ·
00001340  D7 47 F6 42 C0 83 33 04 33 00 80 00 F0 41 93 C1      · G · B · · 3 · 3 · · · · A · ·
00001350  F0 C2 93 C1 F1 C1 93 D1 D4 41 93 C1 F0 42 93 D5      · · · · · · · · · A · · · B · ·
00001360  6D 3C 93 C1 F2 43 97 C1 03 06 30 14 80 00 93 C1      m < · · · C · · · · 0 · · · · ·
00001370  C4 81 93 D1 D5 C4 97 C1 03 02 30 14 80 00 93 C1      · · · · · · · · · · 0 · · · · ·
00001380  F5 C4 93 D1 D6 C4 93 C1 F5 44 93 C1 F6 C3 93 D1      · · · · · · · · · D · · · · · ·
00001390  D7 44 93 C1 F6 43 93 C1 F7 C2 93 D1 D7 C4 93 C1      · D · · · C · · · · · · · · · ·
000013A0  F7 42 93 C1 F1 C4 93 D1 D4 C4 93 C1 F1 44 93 C1      · B · · · · · · · · · · · D · ·
000013B0  F3 C3 93 D1 D5 C4 93 C1 F3 43 C0 80 E7 70 94 C0      · · · · · · · · · C · · · p · ·
000013C0  96 21 97 21 9F 71 90 C0 90 C0 90 C0 90 C0 90 C0      · ! · ! · q · · · · · · · · · ·
```

Word Size: 8 Page: Physical

NOTE Remember that you configured the **Program Trace through Triggering only**, **Program Trace Start Trigger**, and **Program Trace End Trigger** options so that the collected trace reflected program execution from the first source-code comment to the second source-code comment.

d. Use the program-trace window to examine the program's flow of control for the triggered block of code.

The window shows how the flow went first to function `foo()`, then to `foo2()`, then back to `foo()`, and finally back to `main()`.

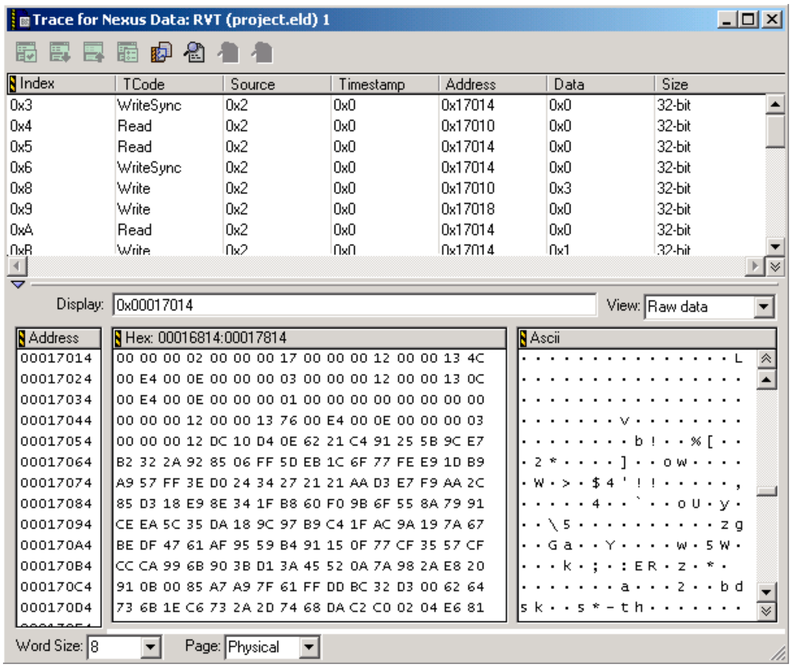
NOTE You can click the address associated with each function to view the source code of that function.

3. Examine the Nexus data trace.
   a. Select **Data** > **View Trace**.

      The **Trace Data Source Selection** dialog box appears.
   b. Select the **Nexus Data** option button.
   c. Click the **OK** button.

      The **Trace for Nexus Data** window appears.

Figure 4.32 Trace for Nexus Data Window



Index	TCode	Source	Timestamp	Address	Data	Size
0x3	WriteSync	0x2	0x0	0x17014	0x0	32-bit
0x4	Read	0x2	0x0	0x17010	0x0	32-bit
0x5	Read	0x2	0x0	0x17014	0x0	32-bit
0x6	WriteSync	0x2	0x0	0x17014	0x0	32-bit
0x8	Write	0x2	0x0	0x17010	0x3	32-bit
0x9	Write	0x2	0x0	0x17018	0x0	32-bit
0xA	Read	0x2	0x0	0x17014	0x0	32-bit
0xB	Write	0x2	0x0	0x17014	0x1	32-bit

Display: 0x00017014   View: Raw data

```
Address   Hex: 00016814:00017814                        Ascii
00017014  00 00 00 02 00 00 00 17 00 00 00 12 00 00 13 4C  • • • • • • • • • • • • • • • L
00017024  00 E4 00 0E 00 00 00 03 00 00 00 12 00 00 13 0C  • • • • • • • • • • • • • • • •
00017034  00 E4 00 0E 00 00 00 01 00 00 00 00 00 00 00 00  • • • • • • • • • • • • • • • •
00017044  00 00 00 12 00 00 13 76 00 E4 00 0E 00 00 00 03  • • • • • • • v • • • • • • • •
00017054  00 00 00 12 DC 10 D4 0E 62 21 C4 91 25 5B 9C E7  • • • • • • • • b ! • • % [ • •
00017064  B2 32 2A 92 85 06 FF 5D EB 1C 6F 77 FE E9 1D B9  • 2 * • • • • ] • • o w • • • •
00017074  A9 57 FF 3E D0 24 34 27 21 21 AA D3 E7 F9 AA 2C  • W • > • $ 4 ' ! ! • • • • • ,
00017084  85 D3 18 E9 8E 34 1F B8 60 F0 9B 6F 55 8A 79 91  • • • • • 4 • • ` • • o U • y •
00017094  CE EA 5C 35 DA 18 9C 97 B9 C4 1F AC 9A 19 7A 67  • • \ 5 • • • • • • • • • • z g
000170A4  BE DF 47 61 AF 95 59 B4 91 15 0F 77 CF 35 57 CF  • • G a • • Y • • • • w • 5 W •
000170B4  CC CA 99 6B 90 3B D1 3A 45 52 0A 7A 98 2A E8 20  • • • k • ; • : E R • z • * •
000170C4  91 0B 00 85 A7 A9 7F 61 FF DD BC 32 D3 00 62 64  • • • • • • • a • • • 2 • • b d
000170D4  73 6B 1E C6 73 2A 2D 74 68 DA C2 C0 02 04 E6 81  s k • • s * - t h • • • • • • •
```

Word Size: 8   Page: Physical

NOTE Remember that you configured the starting and ending addresses of Data Trace Channel 1 (**DTSA1/DTEA1**) and Data Trace Channel 2 (**DTSA2/DTEA2**) so that the collected trace covered the addresses of specific source-code variables.

   d. Use the data-trace window to examine read and write events for the specified address range of the source-code variables.

4. Examine the Nexus miscellaneous trace.
   a. Select **Data** > **View Trace**.

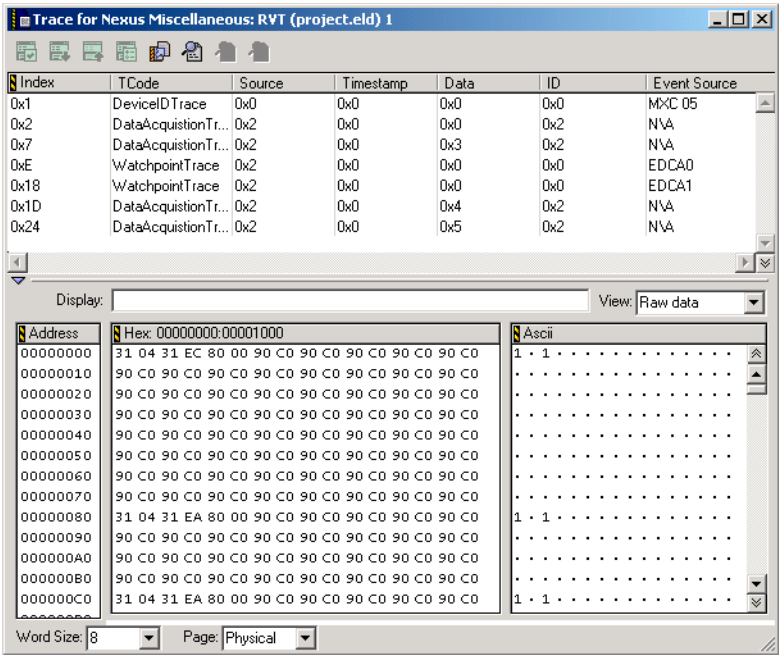
      The **Trace Data Source Selection** dialog box appears.

b. Select the **Nexus Miscellaneous** option button.

c. Click the **OK** button.

The **Trace for Nexus Miscellaneous** window appears.

Figure 4.33 Trace for Nexus Miscellaneous Window



Trace for Nexus Miscellaneous: RVT (project.eld) 1

Index	TCode	Source	Timestamp	Data	ID	Event Source
0x1	DeviceIDTrace	0x0	0x0	0x0	0x0	MXC 05
0x2	DataAcquistionTr...	0x2	0x0	0x0	0x2	N\A
0x7	DataAcquistionTr...	0x2	0x0	0x3	0x2	N\A
0xE	WatchpointTrace	0x2	0x0	0x0	0x0	EDCA0
0x18	WatchpointTrace	0x2	0x0	0x0	0x0	EDCA1
0x1D	DataAcquistionTr...	0x2	0x0	0x4	0x2	N\A
0x24	DataAcquistionTr...	0x2	0x0	0x5	0x2	N\A

Display: View: Raw data

```
Address   Hex: 00000000:00001000                            Ascii
00000000  31 04 31 EC 80 00 90 C0 90 C0 90 C0 90 C0 90 C0   1 · 1 · · · · · · · · · · · · ·
00000010  90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0   · · · · · · · · · · · · · · · ·
00000020  90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0   · · · · · · · · · · · · · · · ·
00000030  90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0   · · · · · · · · · · · · · · · ·
00000040  90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0   · · · · · · · · · · · · · · · ·
00000050  90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0   · · · · · · · · · · · · · · · ·
00000060  90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0   · · · · · · · · · · · · · · · ·
00000070  90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0   · · · · · · · · · · · · · · · ·
00000080  31 04 31 EA 80 00 90 C0 90 C0 90 C0 90 C0 90 C0   1 · 1 · · · · · · · · · · · · ·
00000090  90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0   · · · · · · · · · · · · · · · ·
000000A0  90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0   · · · · · · · · · · · · · · · ·
000000B0  90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0 90 C0   · · · · · · · · · · · · · · · ·
000000C0  31 04 31 EA 80 00 90 C0 90 C0 90 C0 90 C0 90 C0   1 · 1 · · · · · · · · · · · · ·
```

Word Size: 8 Page: Physical

NOTE Remember that you configured the **Enable Watchpoint Messaging**, **Enable Data Acquisition**, and **Data Acquisition** options so that the collected trace corresponds to the variable `b` in the source code.

d. Use the miscellaneous-trace window to examine the watchpoint messages resulting from the EOnCE watchpoint triggers (**EDCA0** and **EDCA1**) and the data-acquisition messages resulting from the write events to the variable `b` in the source code.

NOTE You can use the **Index** column of each Nexus trace window to follow the order of operation from data trace to program trace to miscellaneous trace.

5. End the debugging session.

a. Select **Debug** > **Kill**.

The IDE ends the debugging session and closes the thread window.

b. Close the **StarCore Debugger Console** window.

c. Close the **RVI Information Console for StarCore** window.

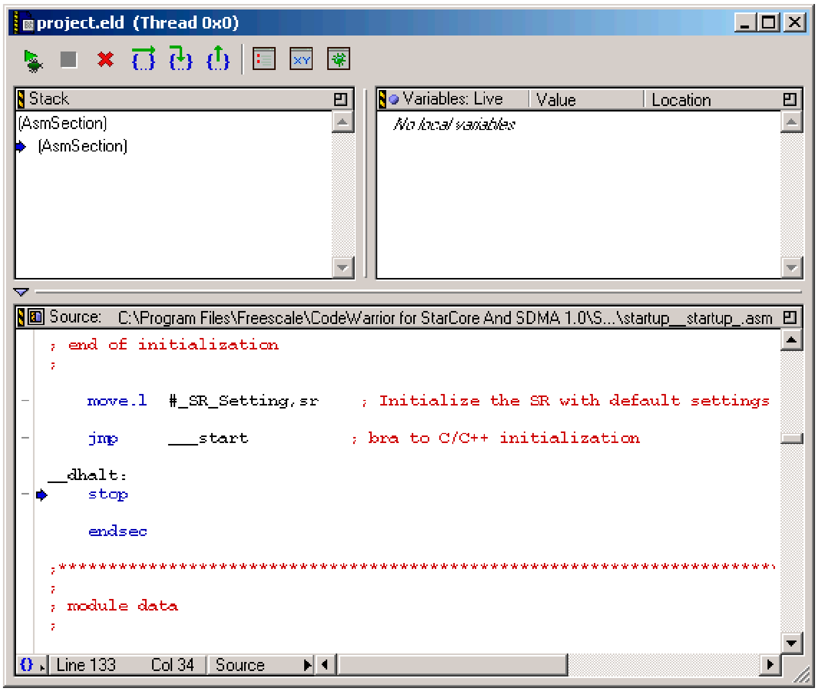
Using Nexus Trace Highlighting

After you complete the steps in "Configuring EOnCE Settings in a Project" on page 182, you can use the IDE's Nexus trace-highlighting feature to see the instructions that will execute between the current branch and the next branch of a program. Follow these steps:

1. From the CodeWarrior menu bar, select **Project** > **Run**.

   The IDE runs the project to completion. A new thread window appears.

Figure 4.34 Thread Window


project.eld (Thread 0x0)
Stack
Variables: Live
Value
Location
Source: C:\Program Files\Freescale\CodeWarrior for StarCore And SDMA 1.0\S...\startup__startup_.asm
Line 133
Col 34
Source

(AsmSection)
(AsmSection)

No local variables

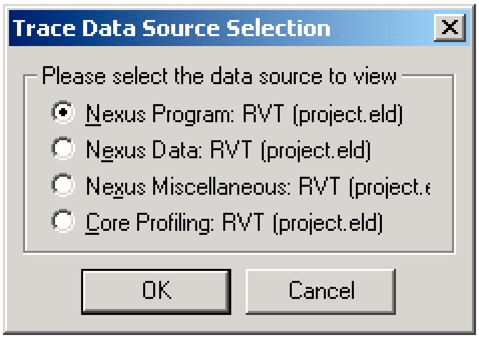
```
; end of initialization
;

    move.l   #_SR_Setting,sr     ; Initialize the SR with default settings

    jmp      ___start           ; bra to C/C++ initialization

__dhalt:
    stop

    endsec

;*************************************************************************
;
; module data
;
```

2. Examine the Nexus program trace.

a. Select **Data** > **View Trace**.

The **Trace Data Source Selection** dialog box appears.

Figure 4.35 Trace Data Source Selection Dialog Box

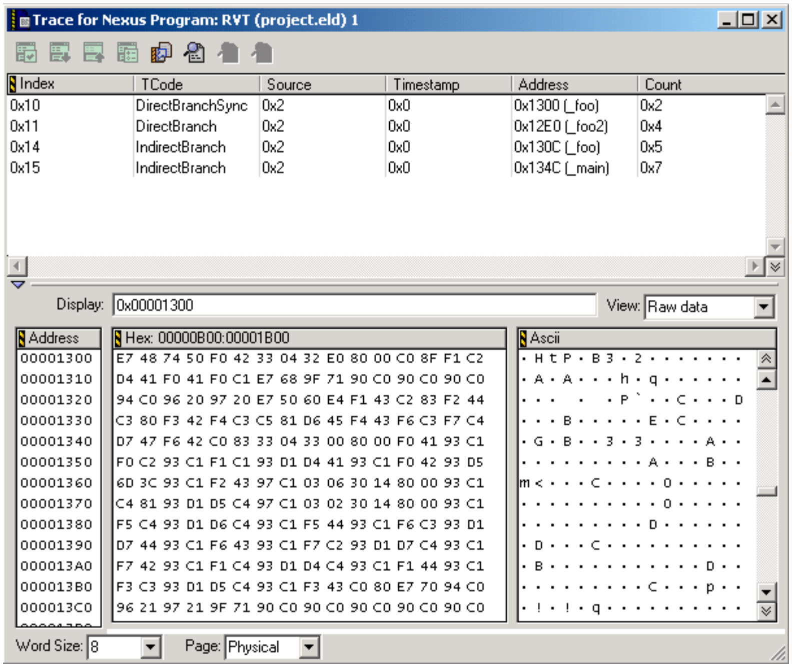

Trace Data Source Selection
Please select the data source to view
Nexus Program: RVT (project.eld)
Nexus Data: RVT (project.eld)
Nexus Miscellaneous: RVT (project.e
Core Profiling: RVT (project.eld)
OK
Cancel

b. Select the **Nexus Program** option button.

c. Click the **OK** button.

The **Trace for Nexus Program** window appears.

Figure 4.36 Trace for Nexus Program Window



Trace for Nexus Program: RVT (project.eld) 1

Index	TCode	Source	Timestamp	Address	Count
0x10	DirectBranchSync	0x2	0x0	0x1300 (_foo)	0x2
0x11	DirectBranch	0x2	0x0	0x12E0 (_foo2)	0x4
0x14	IndirectBranch	0x2	0x0	0x130C (_foo)	0x5
0x15	IndirectBranch	0x2	0x0	0x134C (_main)	0x7

Display: 0x00001300 View: Raw data

```
Address   Hex: 00000B00:00001B00                        Ascii
00001300  E7 48 74 50 F0 42 33 04 32 E0 80 00 C0 8F F1 C2  · H t P · B 3 · 2 · · · · · · ·
00001310  D4 41 F0 41 F0 C1 E7 68 9F 71 90 C0 90 C0 90 C0  · A · A · · · h · q · · · · · ·
00001320  94 C0 96 20 97 20 E7 50 60 E4 F1 43 C2 83 F2 44  · · ·   ·   · P ` · · C · · · D
00001330  C3 80 F3 42 F4 C3 C5 81 D6 45 F4 43 F6 C3 F7 C4  · · · B · · · · · E · C · · · ·
00001340  D7 47 F6 42 C0 83 33 04 33 00 80 00 F0 41 93 C1  · G · B · · 3 · 3 · · · · A · ·
00001350  F0 C2 93 C1 F1 C1 93 D1 D4 41 93 C1 F0 42 93 D5  · · · · · · · · · A · · · B · ·
00001360  6D 3C 93 C1 F2 43 97 C1 03 06 30 14 80 00 93 C1  m < · · · C · · · · 0 · · · · ·
00001370  C4 81 93 D1 D5 C4 97 C1 03 02 30 14 80 00 93 C1  · · · · · · · · · · 0 · · · · ·
00001380  F5 C4 93 D1 D6 C4 93 C1 F5 44 93 C1 F6 C3 93 D1  · · · · · · · · · D · · · · · ·
00001390  D7 44 93 C1 F6 43 93 C1 F7 C2 93 D1 D7 C4 93 C1  · D · · · C · · · · · · · · · ·
000013A0  F7 42 93 C1 F1 C4 93 D1 D4 C4 93 C1 F1 44 93 C1  · B · · · · · · · · · · · D · ·
000013B0  F3 C3 93 D1 D5 C4 93 C1 F3 43 C0 80 E7 70 94 C0  · · · · · · · · · C · · · p · ·
000013C0  96 21 97 21 9F 71 90 C0 90 C0 90 C0 90 C0 90 C0  · ! · ! · q · · · · · · · · · ·
```

Word Size: 8 Page: Physical

NOTE Remember that you configured the **Program Trace through Triggering only**, **Program Trace Start Trigger**, and **Program Trace End Trigger** options so that the collected trace reflected program execution from the first source-code comment to the second source-code comment.

3. In the **Trace for Nexus Program** window, use the **View** list box to specify **Disassembly**.

   The corresponding code appears in the bottom pane of the window.

4. In the top pane of the **Trace for Nexus Program** window, select an **Address** entry for which the corresponding **TCode** entry is one of these messages:
   - `IndirectBranch`
   - `DirectBranch`
   - `RepeatBranch`
   - `IndirectBranchSync`

- `DirectBranchSync`

Observe how the IDE highlights instructions that will execute between the current branch and the next branch.

The trace highlight starts with the first `IndirectBranchSync` or `DirectBranchSync` message. For example, you can view highlighting for subsequent `IndirectBranch`, `DirectBranch`, `IndirectBranchSync`, `DirectBranchSync`, or `RepeatBranch` messages.

NOTE Trace highlighting is not available for the very last message, unless that message is a `RepeatBranch`.

Table 4.11 explains how to interpret the value in the **Count** column of the **Trace for Nexus Program** window.

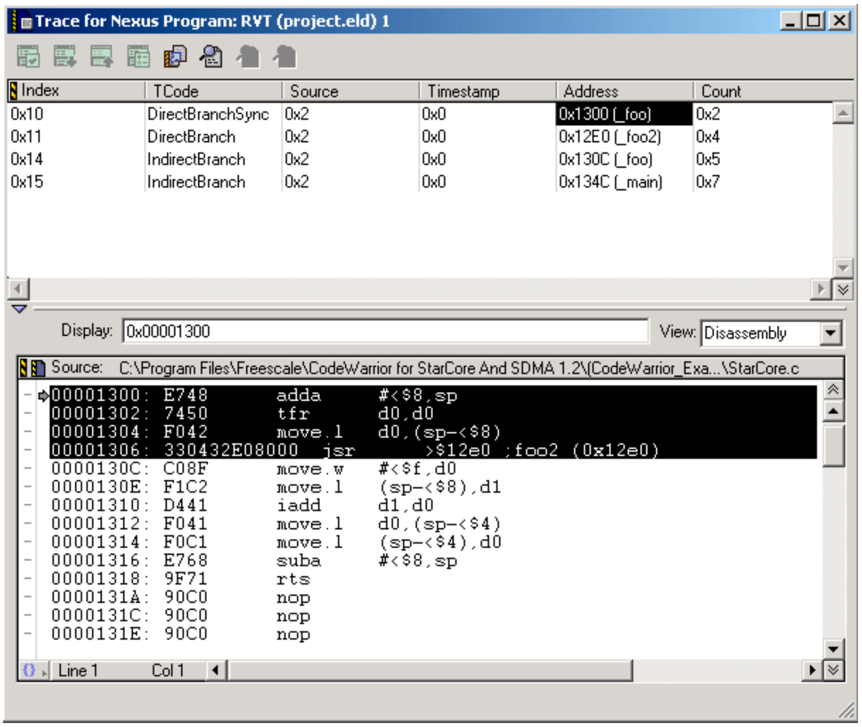
Table 4.11 Interpreting the Count value

If the TCode on the current line is:	And the TCode on the previous line is:	And the TCode on the next line is:	The trace highlight uses the Count value shown in the line for:
`IndirectBranch` or `DirectBranch` or `DirectBranchSync` or `IndirectBranchSync`	any message	`IndirectBranch` or `DirectBranch` or `DirectBranchSync` or `IndirectBranchSync`	the next TCode
`IndirectBranch` or `DirectBranch` or `DirectBranchSync` or `IndirectBranchSync`	any message	`RepeatBranch`	the current TCode

Table 4.11 Interpreting the Count value

If the TCode on the current line is:	And the TCode on the previous line is:	And the TCode on the next line is:	The trace highlight uses the Count value shown in the line for:
`RepeatBranch`	`IndirectBranch` or `DirectBranch` or `DirectBranchSync` or `IndirectBranchSync`	any message	the previous TCode
`RepeatBranch`	`RepeatBranch`	any message	the first TCode that: • appears prior to the `Repeat Branch` message in the previous line • is not itself a `Repeat Branch` message

Figure 4.37 Trace for Nexus Program Window — Trace Highlighting



Index	TCode	Source	Timestamp	Address	Count
0x10	DirectBranchSync	0x2	0x0	0x1300 (_foo)	0x2
0x11	DirectBranch	0x2	0x0	0x12E0 (_foo2)	0x4
0x14	IndirectBranch	0x2	0x0	0x130C (_foo)	0x5
0x15	IndirectBranch	0x2	0x0	0x134C (_main)	0x7

Display: 0x00001300 View: Disassembly

Source: C:\Program Files\Freescale\CodeWarrior for StarCore And SDMA 1.2\(CodeWarrior_Exa...\StarCore.c

```
00001300: E748         adda      #<$8,sp
00001302: 7450         tfr       d0,d0
00001304: F042         move.l    d0,(sp-<$8)
00001306: 330432E08000  jsr          >$12e0 ;foo2 (0x12e0)
0000130C: C08F         move.w    #<$f,d0
0000130E: F1C2         move.l    (sp-<$8),d1
00001310: D441         iadd      d1,d0
00001312: F041         move.l    d0,(sp-<$4)
00001314: F0C1         move.l    (sp-<$4),d0
00001316: E768         suba      #<$8,sp
00001318: 9F71         rts
0000131A: 90C0         nop
0000131C: 90C0         nop
0000131E: 90C0         nop
```

Register Details Window

You use the Register Details window to view descriptions and values of StarCore registers and their bitfields. XML files contain the register descriptions.

The XML register-description files reside in this path, where `CodeWarrior` is the directory in which you installed the CodeWarrior product:

- Windows:
  `CodeWarrior\bin\Plugins\support\Registers`
- Solaris:
  `CodeWarrior/CodeWarrior_Plugins/support/Registers`

The CodeWarrior IDE ‘s default setting is to search all folders in the preceding directory when searching for a register-description file. Register-description files must end with the filename extension `.xml`.

The minimum resolution of bitfield descriptions is 2 bits. Consequently, the Register Details window cannot display single-bit overflow registers.

The maximum resolution of bitfield descriptions is 32 bits. Because the data registers (D0-D15) are 40 bits wide, you cannot view all the bits in a data register simultaneously. Instead, you must view groups of bits—high, low, and extended. For example, to view the bits of the D0 register, use the following XML register-description files:

- D0.E
- D0.L
- D0.H

Some registers have multiple modes (meaning that the register's bits can have multiple meanings, depending on the current mode). If the register you examine has multiple modes, you must select the register-description file for the appropriate mode.

For example, the `OR`*x* registers have multiple modes. The register-description files for these registers have an underscore followed by a group of letters that indicate the mode (where *x* is a number between 0 and 11, excluding 8 and 9):

- `OR`*x*`_GCPM`
- `OR`*x*`_UPM`
- `OR`*x*`_SDRAM`

Similarly, other multi-mode registers have description files that use an underscore followed by an appropriate suffix.

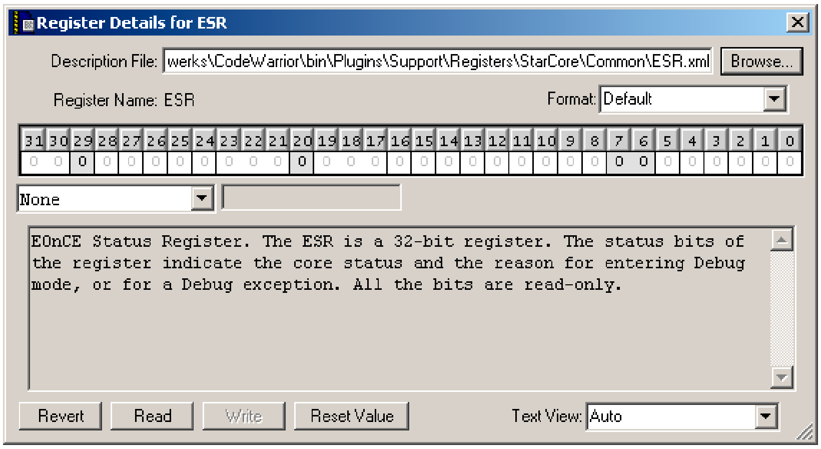
Viewing Register Descriptions

To see registers and their descriptions, follow these steps:

1. Open the Register Details window:
   - Windows: Select **View > Register Details**
   - Solaris: Select **View > Register Details Window**

   The **Register Details** window (Figure 4.38) appears.

Figure 4.38 Register Details Window


Register Details for ESR

Description File: werks\CodeWarrior\bin\Plugins\Support\Registers\StarCore\Common\ESR.xml

Register Name: ESR

Format: Default

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

None

```
EOnCE Status Register. The ESR is a 32-bit register. The status bits of
the register indicate the core status and the reason for entering Debug
mode, or for a Debug exception. All the bits are read-only.
```

Text View: Auto

2. In the **Description File** text box, enter the path to the register-description file that you want to open. Alternatively, click the **Browse** button to open a dialog box that you can use to select the register-description file.

NOTE For registers that have multiple modes, select the register-description file for the appropriate mode. For example, select the appropriate mode for OR*x* register-description files: `OR`*x*`_GCPM`, `OR`*x*`_UPM`, or `OR`*x*`_SDRAM`.

The Register Details window displays the applicable register values and descriptions.

NOTE Use the **Format** list box to specify the format of data that appears in the Register Details window. Also, use the **Text View** list box to specify the text information that appears in the window.

Register Formatter Window

The Register Formatter Window lets you define the contents and layout of the Register Windows by:

- Specifying registers to display in the **Registers** window
- Defining the order in which the **Registers** window displays the specified registers
- Creating a different **Registers** window format for each remote connection you use
- Exporting register layouts to an XML file
- Importing register layouts from an XML file.

The following sections explain how to use the Register Formatter Window.

Specifying Register Sets and Registers to Display

To specify the registers that appear in the **Registers** window, follow these steps:

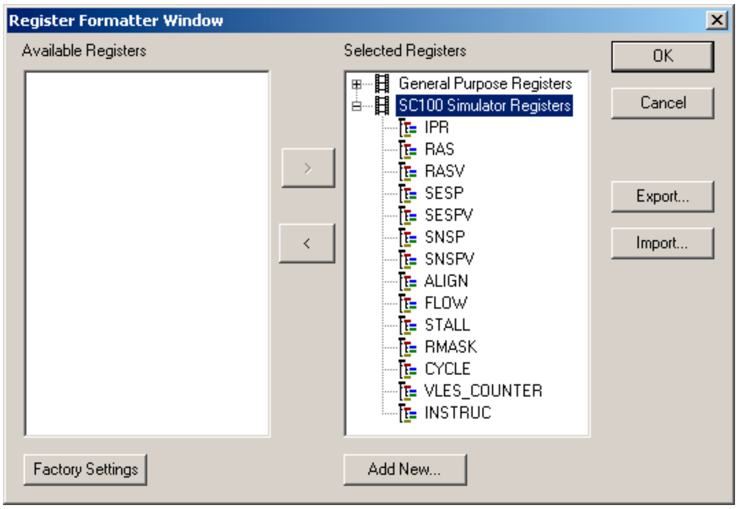
1. Select **Project > Debug**

   The debugger downloads your program to the target device or simulator using the selected remote connection.

2. Select **Debug > Register Window Formatter**

   The **Register Formatter Window** (Figure 4.39) appears.

Figure 4.39 Register Formatter Window



3. To add an entire register group:

   a. From the **Available Registers** list, select a register group that you want to appear in the **Registers** window each time you use the current remote connection.

   b. Click the > button

      The selected register group moves from the Available Registers list to the **Selected Registers** list.

4. To add one register from a register group:

a. In the Available Registers list, click the tree control next to the register group that contains the registers that you want to appear in the Registers window.

b. Select the name of the register that you want to appear in the Registers window.

c. Click the > button

The selected register (along with the name of the group this register is in) moves from the Available Registers list to the Selected Registers list.

5. Click the **OK** button.

The Register Formatter Window saves your configuration and closes.

NOTE If the Registers window is already open, you must close and re-open it for your Register Formatter Window changes to take effect.

6. Select **View** > **Registers**

The Registers window appears and displays the selected register sets.

Adding a Register to a Register Group

To add a register to a register group, follow these steps:

1. Select **Project** > **Debug**

The debugger downloads your program to the target device or simulator using the selected remote connection.

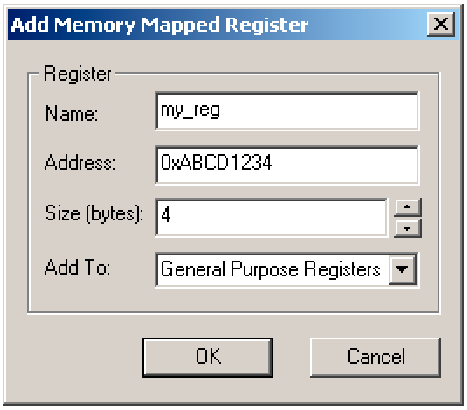
2. Select **Debug** > **Register Window Formatter**

The **Register Formatter Window** window (Figure 4.39) appears.

3. Click the **Add New** button.

The **Add Memory Mapped Register** dialog box (Figure 4.40) appears.

Figure 4.40 Add Memory Mapped Register Dialog Box



Name: my_reg
Address: 0xABCD1234
Size (bytes): 4
Add To: General Purpose Registers

4. In the **Name** text box, enter the name of the new register.
5. In the **Address** text box, enter the absolute address of the new register (in hexadecimal notation).
6. In the **Size (bytes)** text box, enter the width of the register (in decimal notation).
7. Use the **Add To** list box to specify the register group to which you want to add the register.

NOTE To create a new register group, enter a new group name in the **Add To** list box.

8. Click the **OK** button.

   The Add Memory Mapped Register dialog box closes; the Register Formatter Window adds the new register to the specified register group.

Register Windows

The **Registers** window shows the register sets and registers of the target device you that you debug. Using this window, you can see and modify the value of any register.

Using the Registers Window

To use the **Registers** window, follow these steps:

1. Open the CodeWarrior project that you want to debug.
2. Select the project's build target that you want to debug.

3. Select **Project** > **Debug**

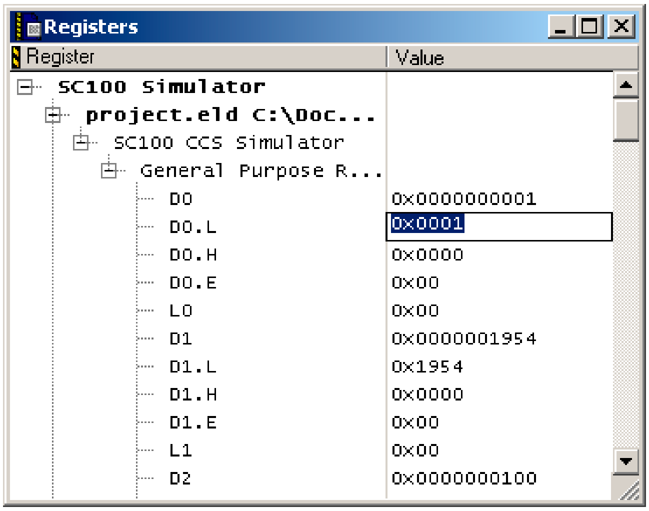
   The debugger downloads your program to the target device and halts execution at the program's entry point.

4. Debug your program as needed.
5. Select **View** > **Registers**

   The **Registers** window (Figure 4.41) appears. The window shows a tree of register groups and register contents.

NOTE Before you can open the **Registers** window, you must halt your program.

Figure 4.41 Registers Window



Registers

Register	Value
SC100 Simulator	
project.eld C:\Doc...	
SC100 CCS Simulator	
General Purpose R...	
D0	0x0000000001
D0.L	0x0001
D0.H	0x0000
D0.E	0x00
L0	0x00
D1	0x0000001954
D1.L	0x1954
D1.H	0x0000
D1.E	0x00
L1	0x00
D2	0x0000000100

Changing a Register's Value

To change a register's value, follow these steps:

1. Double-click the register's value.

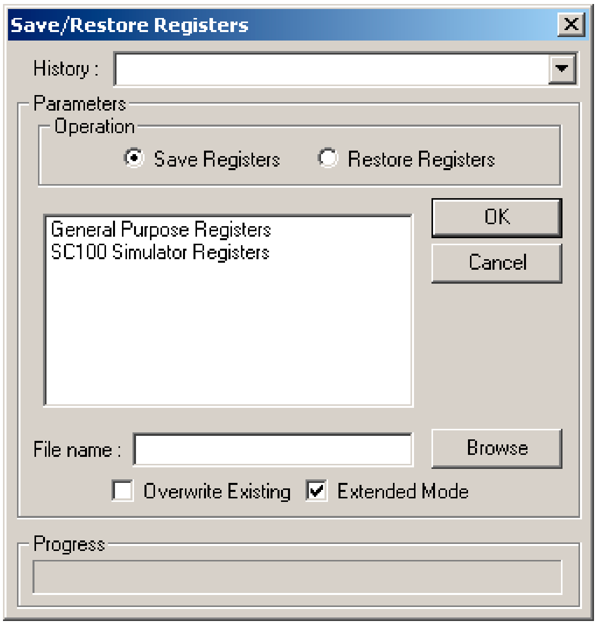
   The value becomes editable.

2. Type the new value (in hexadecimal notation) that you want to assign to the register.
3. Press the Enter key.

Save Restore Registers

To save or restore the values of register banks while you debug your source code, select **Debug > SaveRestoreRegs**. The **Save/Restore Registers** dialog box appears. Figure 4.42 shows the dialog box. Table 4.12 explains each dialog-box option.

Figure 4.42 Save/Restore Registers Dialog Box



[✓ Save Registers] Restore Registers

General Purpose Registers
SC100 Simulator Registers

Overwrite Existing [✓ Extended Mode]

Table 4.12 Save/Restore Registers Dialog Box Options

Option	Explanation
History	This list box lists all the previous save and restore operations. Select a previous save or restore operation to repeat that action.
Operation	These option buttons let you select between load operations and save operations.
• Save Registers	Click this option button to select a save-registers operation.
• Restore Registers	Click this option button to select a restore-registers operation.

Table 4.12 Save/Restore Registers Dialog Box Options (*continued*)

Option	Explanation
Register List	Select from this list the register banks that you want to save. You can select more than one register bank from the list. The list is available just for Save operations.
Filename	Enter in this text box the path to the file that you want to use for the save or restore operation. Alternatively, click the **Browse** button to open a dialog box that you can use to select the file.
Overwrite Existing	Check this checkbox to overwrite the contents of the file that you specify in the **Filename** text box. Clear this checkbox to append to the contents of the file that you specify in the **Filename** text box. Note that this option is available only when you perform Save operations.
Extended Mode	Check this checkbox to include the names of register groups in the contents of the file that you specify in the **Filename** text box. Checking this checkbox is useful when you want to discern registers that are in different cores but have the same name. For example, checking this checkbox makes it easier to discern the D0 register in different cores. Clear this checkbox to exclude the names of register groups in the contents of the file that you specify in the **Filename** text box. For example, clear the checkbox when you examine registers from a single core, because in this case you do not need to discern registers from different cores. Note that this option is available only when you perform Save operations.

Stack Crawl Depth

The maximum depth of the stack crawl is 26 stack frames. For example, in the debugger window of the IDE, you can see no more than 26 lines of stack data. This data shows the current stack frame and the last 25 stack frames up the call chain.

System-Level Connect

You can use the CodeWarrior debugger to perform a system-level connect to a target board, either before or after downloading a program to the board. After you connect to the target board, you can examine system registers and memory.

Performing a System-Level Connect

You can perform a system-level connect any time you have a project window open and your target board is correctly connected.

The following steps explain how to connect in the context of developing and debugging executable code on a target board.

To perform a system-level connect, follow these steps:

1. Select **Project** > **Debug**.

   The debugger downloads your program to the target board. The program starts running and then halts at the entry point of its `main` function.

NOTE The default debugger behavior is to set a temporary breakpoint at the entry point of `main` at program launch.

2. Select **Debug** > **Kill**.

   The debugger stops running.

3. Make sure that you select (or bring forward) the project window for the program that you downloaded.
4. Select **Debug** > **Connect**.

   The debugger connects to the target board.

Now, you can examine registers and memory contents on the board.

Tips for Debugging Assembly Code

If you set a breakpoint in assembly-language code, the source pane of the Thread window does not show the source code preceding the last breakpoint reached. You must change the value of the program counter (which changes the location that the IDE displays) to view that source code. Alternatively, you can view assembly sources in the memory window.

NOTE When you change the program-counter value, make sure that the address value you enter is less than that of the current location.

Changing the Program Counter Value

To change the program-counter value, follow these steps:

1. Select **Debug > Change Program Counter**.

   The **Change Program Counter** dialog box appears.

2. Enter an address (in hexadecimal notation).
3. Click the **OK** button.

   The dialog box closes. The Source pane of the Thread window updates to show the program counter at the specified location.

NOTE The gray arrow in the Memory window is *not* a program counter. Only the blue arrow in the Thread window represents the program counter.

5

Multi-Core Debugging

This chapter explains how to use the CodeWarrior™ debugger's multi-core debugging capabilities. Multi-core debugging lets you debug multiple cores that are connected in a JTAG chain. To use this feature, you create a separate project for each core and debug each executable image using a separate debugger window.

This chapter has these sections:

- Setting Up to Debug Multiple Cores
- Debugging Multiple Cores
- Using Multi-Core Debugging Commands

Setting Up to Debug Multiple Cores

To set up for multi-core debugging, follow these steps:

1. Set up and connect your JTAG chain.

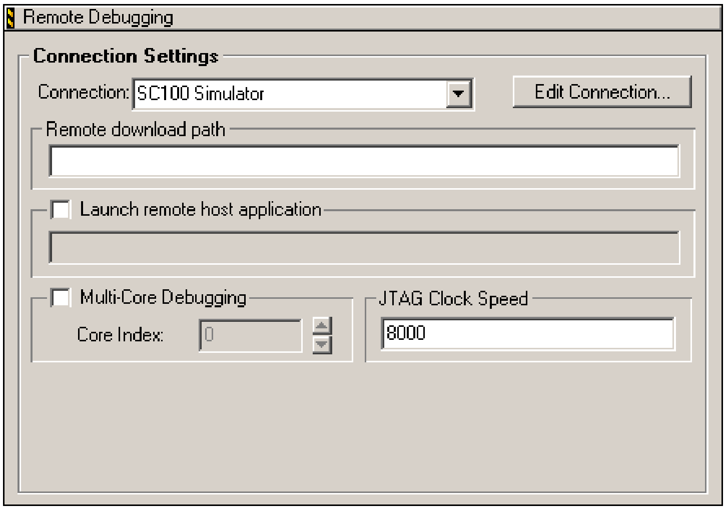
NOTE This chain can have multiple boards or multiple chips on the same board.

2. Open the CodeWarrior project that you want to debug.

NOTE If you debug more than one core, each core must have its own project.

3. In the **Remote Debugging** panel (Figure 5.1), check the **Multi-Core Debugging** checkbox and specify the **Core Index**.

Figure 5.1 Remote Debugging Panel


Remote Debugging

Connection Settings

Connection: SC100 Simulator

Remote download path

Launch remote host application

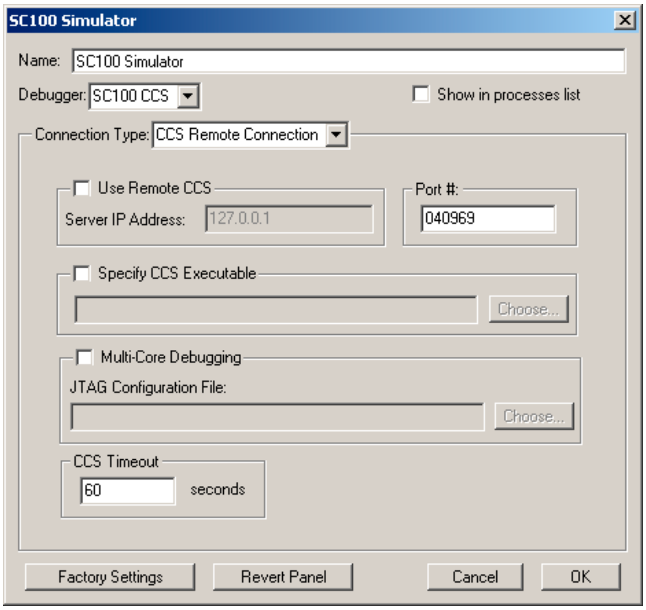
Multi-Core Debugging

Core Index: 0

JTAG Clock Speed: 8000

4. Click the **Edit Connection** button to open the remote-connection configuration dialog box (Figure 5.2).

Figure 5.2 Remote Connection Configuration


SC100 Simulator

Name: SC100 Simulator

Debugger: SC100 CCS

Show in processes list

Connection Type: CCS Remote Connection

Use Remote CCS

Server IP Address: 127.0.0.1

Port #: 040969

Specify CCS Executable

Multi-Core Debugging

JTAG Configuration File:

CCS Timeout

60 seconds

5. Check the **Multi-Core Debugging** checkbox.

   The **JTAG Configuration File** text box becomes editable.

6. Enter in the JTAG Configuration File text box the path to the JTAG initialization file that you created in step 2.

NOTE Depending on the project and the stationery that created it, you may need to change additional target settings. This section explains just those target settings related to multi-core debugging.

7. Select **Project** > **Run**.

   The IDE downloads the program to the specified core. Now you can debug your program.

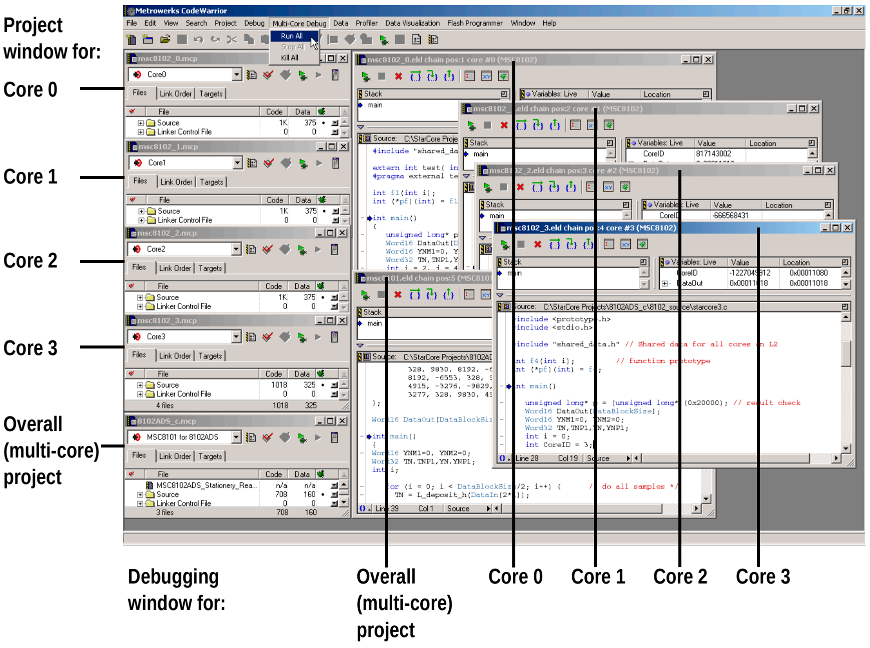
Debugging Multiple Cores

When you start debugging a multi-core project, the CodeWarrior debugger downloads each build target to the appropriate core (with correct settings for multi-core debugging).

Multi-core debugging for StarCore and SDMA is for heterogeneous core debugging. That is, the StarCore and SDMA debuggers each handle their corresponding core debugging operations. This configuration is different from homogeneous core debugging, where one debugger handles debugging operations for more than one core.

Figure 5.3 shows an initial download of a multi-core project created from multi-core stationery. The IDE displays a separate debugging window for each project in the multi-core project.

Figure 5.3 Debugger Windows After Initial Download of Multi-Core Project


Project window for:
Core 0
Core 1
Core 2
Core 3
Overall (multi-core) project
Metrowerks CodeWarrior
File Edit View Search Project Debug Multi-Core Debug Data Profiler Data Visualization Flash Programmer Window Help
Run All
Stop All
Kill All
Debugging window for:
Overall (multi-core) project
Core 0
Core 1
Core 2
Core 3

```
msc8102_3.eld chain pos:4 core #3 (MSC8102)
Source:  C:\StarCore Projects\8102ADS_c\8102_source\starcore3.c
#include <prototype.h>
#include <stdio.h>

#include "shared_data.h" // Shared data for all cores on L2

int f4(int i);           // function prototype
int (*pf)(int) = f4;

int main()

   unsigned long* p = (unsigned long*)(0x20000); // result check
   Word16 DataOut[DataBlockSize];
   Word16 YNM1=0, YNM2=0;
   Word32 TN,TNP1,YN,YNP1;
   int i = 0;
   int CoreID = 3;
Line 28    Col 19
```

To debug multiple cores, follow these steps:

1. Use the simulator projects to create your own applications to debug (adding and deleting files and source code to the various projects as needed).

NOTE To kill all debug sessions and close all debugger windows, select **Multi-Core Debug > Kill All**

2. If required, modify the target settings of any or all of your multi-core projects.

NOTE The target settings related to multi-core debugging should work correctly without modification.

3. When you are ready to debug multiple cores, choose **Project > Debug**

   The debugger downloads your multi-core projects to the simulator.

4. Debug the cores using single-core and multi-core debugging commands.

Using Multi-Core Debugging Commands

If you are debugging a multi-core project, you can use multi-core debugging commands. You also can use the standard single-core debugging commands to debug parts of each core project.

The multi-core debugging commands are in the IDE's Multi-Core Debug menu.

Table 5.1 lists and defines the affect of each multi-core debugging command available in the Multi-Core Debug menu.

Table 5.1 Multi-Core Debugging Commands

Select this command...	To perform this action...
Multi-Core Debug > Run All	Start a multi-core run. This command starts all cores executing as close to the same time as possible. (This action also is known as a synchronous run.)
Multi-Core Debug > Stop All	Perform a multi-core stop. This command stops execution on all cores as close to the same time as possible. (This action also is known as a synchronous stop.)
Multi-Core Debug > Kill All	Kill all multi-core debugging sessions as close to the same time as possible.

6

Embedded Cross Trigger

This chapter explains how to use the CodeWarrior™ IDE to work with the Embedded Cross Trigger (ECT) of the MXC 05, i.300-30, MXC275-30, MXC300-10, and MXC300-30 chipsets. With ECT, a signal sent to one core can trigger signals to other cores.

The ECT interface is a data-driven feature that lets you map core signals to channels. To create this mapping, you do not need to know about the actual registers involved. You can use the IDE to specify the signals sent to other cores based on the events or signals at a specific core.

This chapter has these sections:

- Trigger Signals
- ECT Configuration Window

Trigger Signals

ECT works with two types of trigger signals:

- *Trigger In*—a signal that gets put on a particular channel when an event occurs
- *Trigger Out*—a signal that, reacting to activity on a mapped channel, sends a signal back to a related core

By carefully mapping Trigger Ins and Trigger Outs, you can pass messages between the different cores of a chipset. This mapping is called an *ECT matrix*. For example, you can use an ECT matrix to specify how to pass signals between the StarCore™ and SDMA cores of the MXC275-30 chipset.

ECT Configuration Window

The **ECT Configuration** window lets you set up and implement an ECT matrix. The matrix defines trigger signals and the channels on which those signals occur. Figure 6.1 shows the window. Table 6.1 explains the options in this window.

After you specify a matrix in the ECT Configuration window, the IDE properly configures bits in registers that control signal messaging. For example, the IDE:

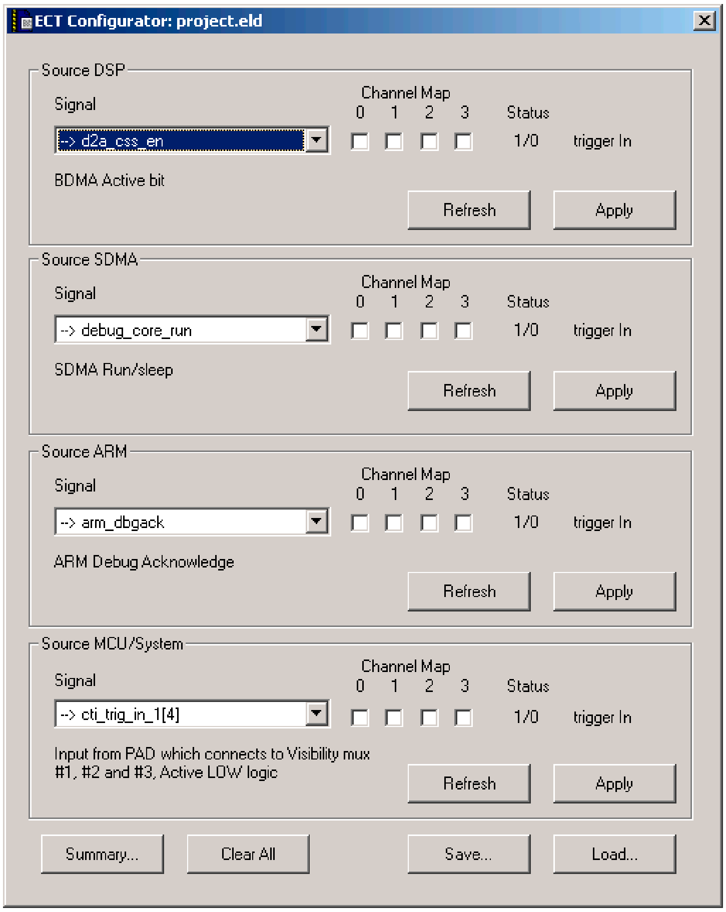
- automatically turns on debugging mode for the CTI device it programs
- ensures that the registers are enabled (not locked)

- makes the registers accessible at the user (and not supervisor) level
- ensures that the proper access key is in the appropriate access register
- sets the appropriate bits in the registers to implement the ECT matrix you specify

NOTE The IDE keeps a list of signals and their associated registers in the `signal.xml` file at this location, where *`CodeWarrior`* is the path to your CodeWarrior installation:

`CodeWarrior``\Plugins\support\ECTPlugin`

Figure 6.1 ECT Configuration Window



ECT Configurator: project.eld

Source DSP
Signal: --> d2a_css_en
Channel Map 0 1 2 3
Status 1/0 trigger In
BDMA Active bit

Source SDMA
Signal: --> debug_core_run
Channel Map 0 1 2 3
Status 1/0 trigger In
SDMA Run/sleep

Source ARM
Signal: --> arm_dbgack
Channel Map 0 1 2 3
Status 1/0 trigger In
ARM Debug Acknowledge

Source MCU/System
Signal: --> cti_trig_in_1[4]
Channel Map 0 1 2 3
Status 1/0 trigger In
Input from PAD which connects to Visibility mux #1, #2 and #3, Active LOW logic

Table 6.1 ECT Configuration Window Options

Option	Explanation
Signal	Use this list box to specify the signal on which you want the specified core to act. The list box groups signals by their related processor. Descriptive text appears below the list box.
Channel Map	Check the appropriate checkboxes to specify the channels (**0**, **1**, **2**, or **3**) on which the selected Signal occurs.
Status	Identifies the selected Signal as a **trigger In** signal or a **trigger Out** signal.
Refresh	Click to have the window show the current channel mappings for the specified core. The IDE reads the current register values and updates the window with that information. This button is useful when you have other processes (such as other toolsets) that modify channel mappings while you use the CodeWarrior IDE. Clicking this button ensures that you see the most up-to-date mapping information.
Apply	Click to have the IDE configure the specified core to reflect your Signal and Channel Map settings.
Summary	Click to open a dialog box that lists the current channel mappings (Figure 6.2).
Clear All	Click to clear all existing Signal and Channel Map settings and start over from scratch.
Save	Click, then use the resulting dialog box to save a file that contains the specified Signal and Channel Map settings for each core.
Load	Click, then use the resulting dialog box to load a file that contains Signal and Channel Map settings for each core. Remember to load a file that you previously saved from this window.

Opening the ECT Configuration Window

You use the ECT Configuration window to create an ECT matrix.

To open the window, follow these steps:

1. Start a StarCore debugging session.
2. Select **Debug** > **ECT** > **Configure**.

   The ECT Configuration window (Figure 6.1 on page 214) appears.

Creating an ECT Matrix

You create an ECT matrix by using the ECT Configuration window to specify trigger signals and their corresponding channels.

To create an ECT matrix, follow these steps for each source processor:

1. Use the Signal list box to specify the desired signal.
2. Check the checkboxes for each channel that you want to associate with the selected signal.
3. Click the Apply button.

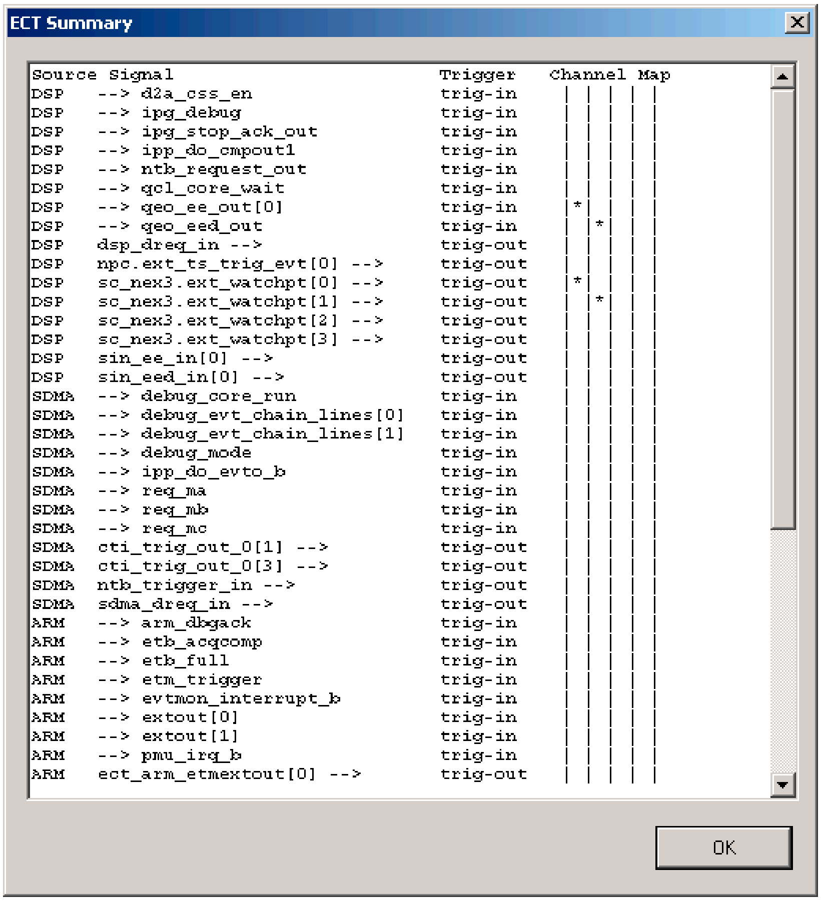
NOTE You can program some channel mappings from multiple processors. Click the **Refresh** button to ensure that the ECT Configuration window shows the channel-mapping information currently programmed into the processor.

Viewing an ECT Matrix Summary

The ECT Configuration window can display a summary view that shows all the trigger signals and associated channels that you defined. You can use this summary view to see your entire configuration at a glance.

To view an ECT matrix summary, click the **Summary** button. The Summary dialog box (Figure 6.2 on page 217) appears.

Figure 6.2 Summary Dialog Box



```
Source Signal                              Trigger   Channel Map
DSP    --> d2a_css_en                      trig-in    | | | | |
DSP    --> ipg_debug                       trig-in    | | | | |
DSP    --> ipg_stop_ack_out                trig-in    | | | | |
DSP    --> ipp_do_cmpout1                  trig-in    | | | | |
DSP    --> ntb_request_out                 trig-in    | | | | |
DSP    --> qcl_core_wait                   trig-in    | | | | |
DSP    --> qeo_ee_out[0]                   trig-in    |*| | | |
DSP    --> qeo_eed_out                     trig-in    | |*| | |
DSP    dsp_dreq_in -->                     trig-out   | | | | |
DSP    npc.ext_ts_trig_evt[0] -->          trig-out   | | | | |
DSP    sc_nex3.ext_watchpt[0] -->          trig-out   |*| | | |
DSP    sc_nex3.ext_watchpt[1] -->          trig-out   | |*| | |
DSP    sc_nex3.ext_watchpt[2] -->          trig-out   | | | | |
DSP    sc_nex3.ext_watchpt[3] -->          trig-out   | | | | |
DSP    sin_ee_in[0] -->                    trig-out   | | | | |
DSP    sin_eed_in[0] -->                   trig-out   | | | | |
SDMA   --> debug_core_run                  trig-in    | | | | |
SDMA   --> debug_evt_chain_lines[0]        trig-in    | | | | |
SDMA   --> debug_evt_chain_lines[1]        trig-in    | | | | |
SDMA   --> debug_mode                      trig-in    | | | | |
SDMA   --> ipp_do_evto_b                   trig-in    | | | | |
SDMA   --> req_ma                          trig-in    | | | | |
SDMA   --> req_mb                          trig-in    | | | | |
SDMA   --> req_mc                          trig-in    | | | | |
SDMA   cti_trig_out_0[1] -->               trig-out   | | | | |
SDMA   cti_trig_out_0[3] -->               trig-out   | | | | |
SDMA   ntb_trigger_in -->                  trig-out   | | | | |
SDMA   sdma_dreq_in -->                    trig-out   | | | | |
ARM    --> arm_dbgack                      trig-in    | | | | |
ARM    --> etb_acqcomp                     trig-in    | | | | |
ARM    --> etb_full                        trig-in    | | | | |
ARM    --> etm_trigger                     trig-in    | | | | |
ARM    --> evtmon_interrupt_b              trig-in    | | | | |
ARM    --> extout[0]                       trig-in    | | | | |
ARM    --> extout[1]                       trig-in    | | | | |
ARM    --> pmu_irq_b                       trig-in    | | | | |
ARM    ect_arm_etmextout[0] -->            trig-out   | | | | |
```

Saving an ECT Matrix

Saving an ECT matrix allows you to save the configuration that you specify in the ECT Configuration window. This way, you do not have to re-enter the entire configuration again from scratch.

To save an ECT matrix, follow these steps:

1. Click the **Save** button.

   A standard Save dialog box appears.

2. Use the Save dialog box to navigate to the location where you want to save your configuration.
3. Click **Save**.

The ECT Configuration window saves your configuration to the specified file.

Loading an ECT Matrix

You use the ECT Configuration window to load a different ECT matrix. For example, you could load an ECT matrix that you previously saved.

1. Click the **Load** button.

   A standard Open dialog box appears.
2. Use the Open dialog box to select the configuration that you previously saved.
3. Click **Open**.

The ECT Configuration window now reflects the configuration that you loaded.

7

Enhanced On-Chip Emulation

This chapter explains how to use the CodeWarrior™ Enhanced On-Chip Emulation (EOnCE™) Configurator. This tool lets you use the StarCore™ DSP's on-chip EOnCE module from within the CodeWarrior IDE. The EOnCE module allows non-intrusive interaction with the StarCore chip's core. Also, you can use a special debugging environment to examine the contents of registers, memory, and on-chip peripherals.

This chapter has these sections:

EOnCE Features

With the EOnCE Configurator, you can keep a running trace of tasks and interrupts, and you can determine when events of interest occurred.

Overview

Using the StarCore chip's EOnCE module for debugging:

- reduces system intrusion.
- reduces the use of general-purpose peripherals when debugging input and output.
- standardizes system-level debugging across multiple platforms.
- provides a rich set of breakpoint features.

  One key difference between regular software breakpoints and EOnCE breakpoints is the time at which the program halts. With a regular software breakpoint, the program halts immediately *before* the breakpoint instruction. With an EOnCE breakpoint, the program halts immediately *after* the breakpoint instruction.
- provides the ability to non-intrusively read from and write to peripheral registers while debugging.

- provides a trace buffer for program flow and data tracing.
- uses a programming model that either your software or the CodeWarrior debugger can access directly.
- does not require the debugger to halt peripherals.

EOnCE Trace Buffer Overview

This important information pertains to using the EOnCE trace buffer:

- The trace buffer is a circular buffer. If the buffer is full, and you continue to step through source code, the debugger overwrites the buffer from the beginning.
- You can determine whether the trace buffer is full by examining the TBFULL bit of the ESR (EOnCE Status Register). When the trace buffer is full, the TBFULL bit is set.
- You can trace up to 2048 bytes of addresses in the trace buffer.
- You must enable the trace buffer each time before getting new trace information.

EOnCE Configurator Page Explanations

This section explains each EOnCE Configurator page. You use these pages to configure on-chip EOnCE module debugging.

NOTE When you specify settings in the EOnCE Configurator, configure the tabbed pages in left-to-right order. For example, configure the **Address Event Detection Channel 0** page before configuring the **Event Counter** page. Also, within a page, configure settings from the top-left position to the bottom-right position.

To save EOnCE Configurator settings for your current debugging session, you must click the **OK** button in the **EOnCE Configurator** window.

To save EOnCE Configurator settings in a file for later use, select **Debug** > **EOnCE** > **Save EOnCE Configuration** and specify the filename.

To use a saved configuration, select **Debug** > **EOnCE** > **Open EOnCE Configuration** and select the configuration file using the dialog box that appears.

- EE Pins Controller Page
- Address Event Detection Channel Pages
- Data Event Detection Channel Page
- Event Counter Page
- Event Selector Page

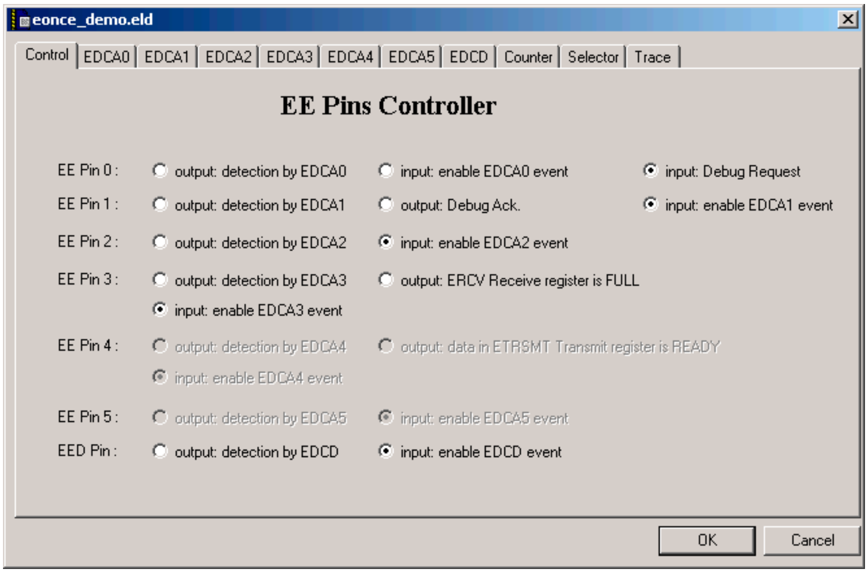
- Trace Unit Page

EE Pins Controller Page

Use this page to configure the EOnCE controller, specifically the EE pins. EE pins are general-purpose pins that can serve as input or output EOnCE pins.

Figure 7.1 shows the **EE Pins Controller** page. Table 7.1 explains each option.

Figure 7.1 EE Pins Controller Page



EE Pins Controller

EE Pin 0 :	output: detection by EDCA0	input: enable EDCA0 event	[✓ input: Debug Request]
EE Pin 1 :	output: detection by EDCA1	output: Debug Ack.	[✓ input: enable EDCA1 event]
EE Pin 2 :	output: detection by EDCA2	[✓ input: enable EDCA2 event]	
EE Pin 3 :	output: detection by EDCA3	output: ERCV Receive register is FULL	
	[✓ input: enable EDCA3 event]		
EE Pin 4 :	output: detection by EDCA4	output: data in ETRSMT Transmit register is READY	
	[✓ input: enable EDCA4 event]		
EE Pin 5 :	output: detection by EDCA5	[✓ input: enable EDCA5 event]	
EED Pin :	output: detection by EDCD	[✓ input: enable EDCD event]	

Table 7.1 EE Pins Controller Page

Page Item	Setting	Explanation
EE Pin 0	output: detection by EDCA0	Toggle the signal on EE pin 0 after detecting an event on EDCA0 (event detection channel 0).
	input: enable EDCA0 event	An input signal from EE pin 0 enables an event on EDCA0 (event detection channel 0).
	input: Debug Request	A signal on EE pin 0 during and after reset causes the core to enter debug mode. A signal on EE pin 0 also causes an exit from stop or wait processing states of the core.
EE Pin 1	output: detection by EDCA1	Toggle the signal on EE pin 1 after detecting an event on EDCA1 (event detection channel 1).
	output: Debug Ack.	Turn on a signal on EE pin 1 after the core enters debug mode. After the core exits from debug mode, turn off the signal on EE pin 1.
	input: enable EDCA1 event	An input signal from EE pin 1 enables an event on EDCA1 (event detection channel 1).
EE Pin 2	output: detection by EDCA2	Toggle the signal on EE pin 2 after detecting an event on EDCA2 (event detection channel 2).
	input: enable EDCA2 event	An input signal from EE pin 2 enables an event on EDCA2 (event detection channel 2) and ECNT.
EE Pin 3	output: detection by EDCA3	Toggle the signal on EE pin 3 after detecting an event on EDCA3 (event detection channel 3).
	output: ERCV Receive register is full	Turn on a signal on EE pin 3 after the host finishes writing to the ERCV register.After the host finishes reading the ETRSMT register, turn off the signal on EE pin 3.
	input: enable EDCA3 event	An input signal from EE pin 3 enables an event on EDCA3 (event detection channel 3).
EE Pin 4	Not applicable.	

Table 7.1 EE Pins Controller Page (*continued*)

Page Item	Setting	Explanation
EE Pin 5	Not applicable.	
EED Pin	output: detection by EDCD	Toggle the signal on the EED pin after detecting an event on EDCD (data event detection channel).
	input: enable EDCD event	An input signal from the EED pin enables an event on EDCD (data event detection channel).

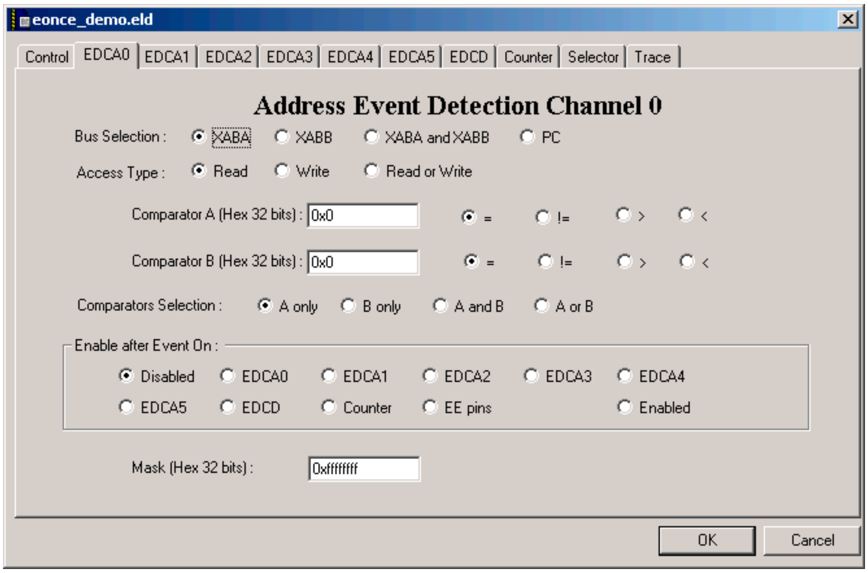
Address Event Detection Channel Pages

The EOnCE module has several address-event detection channels that can, according to your settings, detect address values from an address bus. Each address-event detection channel has a corresponding EOnCE Configurator page:

- **Address Event Detection Channel 0** page - EDCA0
- **Address Event Detection Channel 1** page - EDCA1
- **Address Event Detection Channel 2** page - EDCA2
- **Address Event Detection Channel 3** page - EDCA3
- **Address Event Detection Channel 4** page - EDCA4
- **Address Event Detection Channel 5** page - EDCA5

Figure 7.2 shows the EDCA0 page as a sample address-event detection channel page. Table 7.2 explains each option.

Figure 7.2 Address Event Detection Channel 0 Page



eonce_demo.eld

Control | EDCA0 | EDCA1 | EDCA2 | EDCA3 | EDCA4 | EDCA5 | EDCD | Counter | Selector | Trace

Address Event Detection Channel 0

Bus Selection : [✓ XABA] XABB XABA and XABB PC

Access Type : [✓ Read] Write Read or Write

Comparator A (Hex 32 bits) : 0x0 [✓ =] != > <

Comparator B (Hex 32 bits) : 0x0 [✓ =] != > <

Comparators Selection : [✓ A only] B only A and B A or B

Enable after Event On :
[✓ Disabled] EDCA0 EDCA1 EDCA2 EDCA3 EDCA4
EDCA5 EDCD Counter EE pins Enabled

Mask (Hex 32 bits) : 0xffffffff

OK Cancel

Table 7.2 Address Channel Page

Page Item	Explanation
Bus Selection	The bus on which to detect an address value: • XABA • XABB • XABA and XABB • PC For example, to set a breakpoint on an instruction, specify PC, which represents the value of the program counter.
Access Type	The type of access performed on the specified address: • Read • Write • Read or write

Table 7.2 Address Channel Page (*continued*)

Page Item	Explanation
Comparator A	Specify a value (in hexadecimal, with a maximum length of 32 bits) with which to compare the detected address value: • = (equal) • != (not equal) • > (greater than) • < (less than)
Comparator B	Specify a value (in hexadecimal, with a maximum length of 32 bits) with which to compare the detected address value: • = (equal) • != (not equal) • > (greater than) • < (less than)
Comparators Selection	Choose the values with which to compare the detected address value: • A only • B only • A and B • A or B

Table 7.2 Address Channel Page (*continued*)

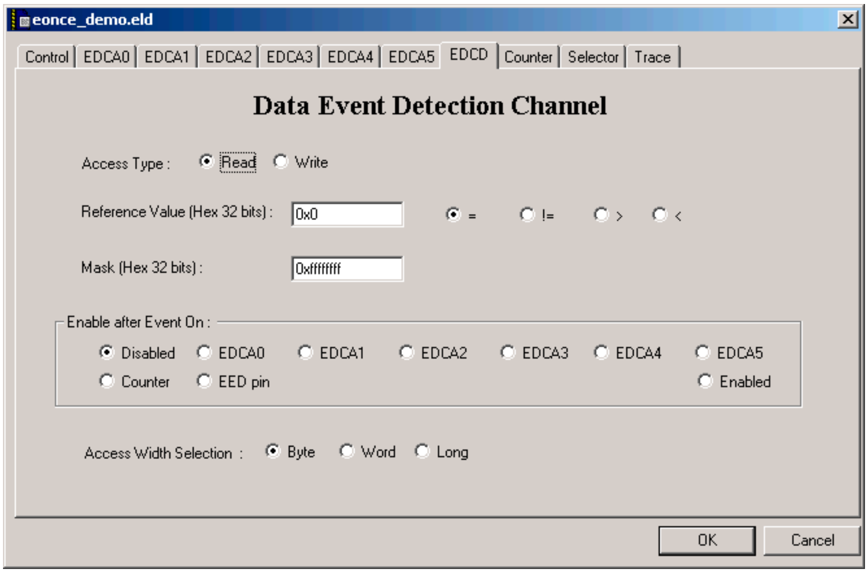
Page Item	Explanation
Enable after Event On	Enable the comparison specified by this page after an event on the specified item. You can specify one of the following: • Disabled—the EOnCE event detector never performs the specified comparison and does not trigger an event • EDCA0, EDCA1, EDCA2, EDCA3—the EOnCE event detector performs the specified comparison after an event on the specified address occurs • EDCD—the EOnCE event detector performs the specified comparison after an event on EDCD occurs • Counter—the EOnCE event detector performs the specified comparison after an event on the Counter occurs • EE pins—the EOnCE event detector performs the specified comparison after an event on any of the EE pins occurs • Enabled—the EOnCE event detector always performs the specified comparison
Mask (Hex 32 bits)	Use this field to set the value of the EDCA mask register. The EDCA mask register allows masking of any of the bits in the detected address before the address is compared with a value that you specified in the Comparator A or Comparator B fields. (All the bits of this register are set to 1 during core reset.) The CodeWarrior IDE performs an AND operation on the bits of the detected address and the mask value, which has the following results: • An address bit that corresponds to a mask bit with a value of 1 keeps its original value (0 or 1) before being compared. • An address bit with a value of 0 that corresponds to a mask bit with a value of 0 keeps its original value before being compared. • An address bit with a value of 1 that corresponds to a mask bit with a value of 0 changes to a value of 0 before being compared. After applying the mask to the address, the CodeWarrior IDE performs any comparisons that you previously defined.

Data Event Detection Channel Page

You can use the **Data Event Detection Channel** page to detect a particular data value.

Figure 7.3 shows the **Data Event Detection Channel** page.

Figure 7.3 Data Event Detection Channel Page



Data Event Detection Channel

Access Type : [✓ Read] Write

Reference Value (Hex 32 bits) : 0x0 [✓ =] != > <

Mask (Hex 32 bits) : 0xffffffff

Enable after Event On : [✓ Disabled] EDCA0 EDCA1 EDCA2 EDCA3 EDCA4 EDCA5 Counter EED pin Enabled

Access Width Selection : [✓ Byte] Word Long

Table 7.3 lists and defines the settings you can make on the **Data Event Detection Channel** page of the EOnCE Configurator.

Table 7.3 Data Event Detection Channel Page

Page Item	Explanation
Access Type	Indicates whether the data value to detect is being read or written.
Reference Value	Specify a value (in hexadecimal, with a maximum length of 32 bits) with which to compare the detected address value. If you specify a byte or a word, use least significant bit (LSB) alignment. You can specify these comparison types: • = (equal) • != (not equal) • > (greater than) • < (less than)

Table 7.3 Data Event Detection Channel Page (*continued*)

Page Item	Explanation
Mask	A 32-bit value that you can use to mask any bits in the sampled data value before the CodeWarrior IDE compares it to the specified reference value. Bits with a value of 0 in the mask cause the corresponding bit in the sampled data value to be set to 0. (A bitwise AND operation is performed on the mask value and sampled data value.) All the mask bits are set to 1 during reset.
Enable After Event On	Enable the comparison specified by this page after an event on the specified item. You can specify one of the following: • Disabled • EDCA0, EDCA1, EDCA2, EDCA3 • Counter • EED pins • Enabled If you select disabled, the IDE does not perform a comparison on the sampled data value. If you select enabled, the IDE performs the specified comparison if an event occurs on any of the items in the list.
Access Width Selection	Indicates the width of the data access to watch. The CodeWarrior IDE compares the masked data and the reference value as follows, based on whether you specify byte, word, or long: • If you specify *byte*, the CodeWarrior IDE compares only the 8 least-significant bits of each value. • If you specify *word*, the CodeWarrior IDE compares only the 16 least-significant bits of each value. • If you specify *long*, the CodeWarrior IDE compares all 32 bits of each value.

Event Counter Page

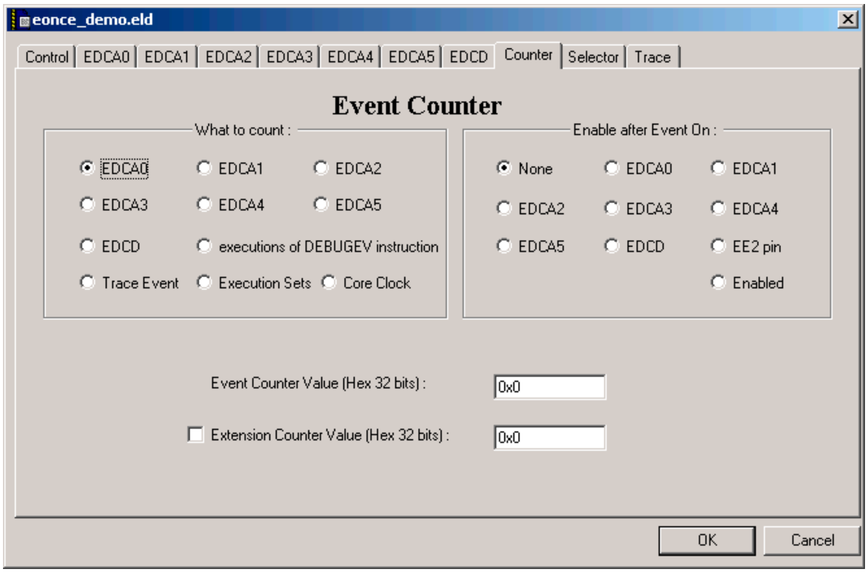
The EOnCE has a 64-bit event counter that can count events related to these items:

- The address event detection channels
- The data event detection channel
- DEBUGEV instructions
- Trace buffer tracing

- Instruction execution
- The core clock

Figure 7.4 shows the **Event Counter** page of the EOnCE Configurator.

Figure 7.4 Event Counter Page



Event Counter

What to count : [✓ EDCA0] EDCA1 EDCA2 EDCA3 EDCA4 EDCA5 EDCD executions of DEBUGEV instruction Trace Event Execution Sets Core Clock

Enable after Event On : [✓ None] EDCA0 EDCA1 EDCA2 EDCA3 EDCA4 EDCA5 EDCD EE2 pin Enabled

Event Counter Value (Hex 32 bits) : 0x0

☐ Extension Counter Value (Hex 32 bits) : 0x0

Table 7.4 lists and defines the settings you can make on the **Event Counter** page of the EOnCE Configurator.

Table 7.4 Event Counter Page

Page Item	Explanation
What to count	Tell the CodeWarrior IDE to count events on the following items: • EDCA0, EDCA1, EDCA2, EDCA3 • EDCD • Execution Set in DEBUGEV • Trace Event • Execution Sets • Core Clock
Enable after Event On	Enable a count on an event for the item specified in the **What to count** group after an event on the specified item. You can specify one of the following: • Disabled • EDCA0, EDCA1, EDCA2, EDCA3 • EDCD • EE2 pin • Enabled If you select Disabled, the IDE does not perform a count. If you select Enabled, the IDE performs the count after an event occurs on any of the items in the list.
Event Counter Value	Specify the first 32 bits of the counter value (the maximum value to which to count).
Extension Counter Value	Specify the second 32 bits of the counter value (the maximum value to which to count). To use a 64-bit counter value, you must enable the checkbox next to this field.

Event Selector Page

The **Event Selector** page specifies which events cause a particular debugging action to occur. The debugging actions follow:

- Place the EOnCE module in debug mode
- Generate a debugging exception
- Enable the trace buffer
- Disable the trace buffer

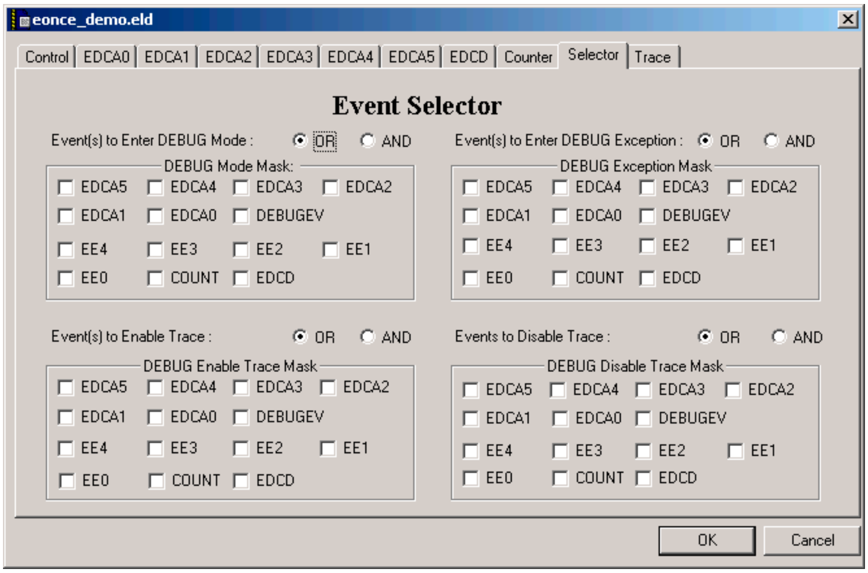
On the **Event Selector** page, you can specify that after an event occurs on one of the following items, the corresponding debugging action occurs:

- Address event detection channels
- Data event detection channels
- Event counter
- EE pins
- DEBUGEV instructions

You also can specify that multiple events must occur to trigger a particular debug event.

Figure 7.5 shows the **Event Selector** page of the EOnCE Configurator.

Figure 7.5 Event Selector Page



Event Selector

Event(s) to Enter DEBUG Mode : [✓ OR] AND

DEBUG Mode Mask:
☐ EDCA5 ☐ EDCA4 ☐ EDCA3 ☐ EDCA2
☐ EDCA1 ☐ EDCA0 ☐ DEBUGEV
☐ EE4 ☐ EE3 ☐ EE2 ☐ EE1
☐ EE0 ☐ COUNT ☐ EDCD

Event(s) to Enter DEBUG Exception : [✓ OR] AND

DEBUG Exception Mask
☐ EDCA5 ☐ EDCA4 ☐ EDCA3 ☐ EDCA2
☐ EDCA1 ☐ EDCA0 ☐ DEBUGEV
☐ EE4 ☐ EE3 ☐ EE2 ☐ EE1
☐ EE0 ☐ COUNT ☐ EDCD

Event(s) to Enable Trace : [✓ OR] AND

DEBUG Enable Trace Mask
☐ EDCA5 ☐ EDCA4 ☐ EDCA3 ☐ EDCA2
☐ EDCA1 ☐ EDCA0 ☐ DEBUGEV
☐ EE4 ☐ EE3 ☐ EE2 ☐ EE1
☐ EE0 ☐ COUNT ☐ EDCD

Events to Disable Trace : [✓ OR] AND

DEBUG Disable Trace Mask
☐ EDCA5 ☐ EDCA4 ☐ EDCA3 ☐ EDCA2
☐ EDCA1 ☐ EDCA0 ☐ DEBUGEV
☐ EE4 ☐ EE3 ☐ EE2 ☐ EE1
☐ EE0 ☐ COUNT ☐ EDCD

Table 7.5 lists and defines the settings you can make on the **Event Selector** page of the EOnCE Configurator.

Table 7.5 Event Selector Page

Page Item	Explanation
Event(s) to Enter DEBUG Mode	Select OR to indicate that any of the events chosen in DEBUG Mode Mask place the EOnCE module in debug mode. Select AND to indicate that all the events chosen in DEBUG Mode Mask must occur to place the EOnCE module in debug mode.
DEBUG Mode Mask	Place the EOnCE module in debug mode after an event on one or more specified items. You can specify the following items: • EDCA0, EDCA1, EDCA2, EDCA3 • DEBUGEV • EE4, EE3, EE2, EE1, EE0 • Counter • EDCD Click **Any** to specify one item or **All** to specify multiple items.
Event(s) to Enter DEBUG Exception	Select OR to indicate that any of the events chosen in DEBUG Exception Mask generate a debugging exception. Select AND to indicate that all the events chosen in DEBUG Exception Mask must occur to generate a debugging exception.
DEBUG Exception Mask	Generate a debug exception after an event on one or more specified items. You can specify the following items: • EDCA0, EDCA1, EDCA2, EDCA3 • DEBUGEV • EE4, EE3, EE2, EE1, EE0 • Counter • EDCD Click **Any** to specify one item or **All** to specify multiple items.
Event(s) to Enable Trace	Select OR to indicate that any of the events chosen in DEBUG Enable Trace Mask enable the trace buffer. Select AND to indicate that all the events chosen in DEBUG Enable Trace Mask must occur to enable the trace buffer.

Table 7.5 Event Selector Page (*continued*)

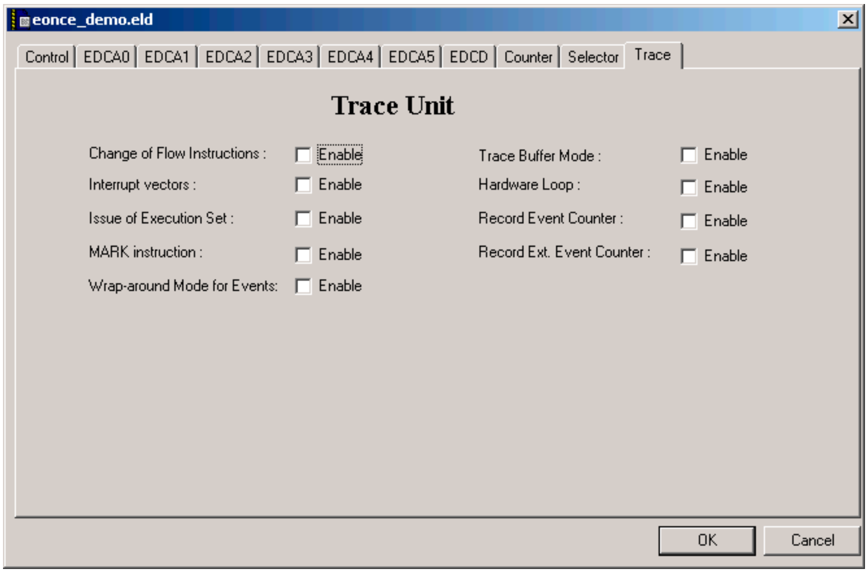
Page Item	Explanation
DEBUG Enable Trace Mask	Enable tracing after an event on one or more specified items. You can specify the following items: • EDCA0, EDCA1, EDCA2, EDCA3 • DEBUGEV • EE4, EE3, EE2, EE1, EE0 • Counter • EDCD Click **Any** to specify one item or **All** to specify multiple items.
Events to Disable Trace	Select OR to indicate that any of the events chosen in DEBUG Disable Trace Mask disable the trace buffer. Select AND to indicate that all the events chosen in DEBUG Disable Trace Mask must occur to disable the trace buffer.
DEBUG Disable Trace Mask	Disable tracing after an event on one or more specified items. You can specify the following items: • EDCA0, EDCA1, EDCA2, EDCA3 • DEBUGEV • EE4, EE3, EE2, EE1, EE0 • Counter • EDCD Click **Any** to specify one item or **All** to specify multiple items.

Trace Unit Page

With EOnCE, you can collect data in a trace buffer as you debug a program. You can use the **Trace Unit** page to choose the trace buffer settings.

Figure 7.6 shows the **Trace Unit** page of the EOnCE Configurator.

Figure 7.6 EOnCE Configurator Trace Unit Page



Change of Flow Instructions : [ ] Enable
Interrupt vectors : [ ] Enable
Issue of Execution Set : [ ] Enable
MARK instruction : [ ] Enable
Wrap-around Mode for Events: [ ] Enable
Trace Buffer Mode : [ ] Enable
Hardware Loop : [ ] Enable
Record Event Counter : [ ] Enable
Record Ext. Event Counter : [ ] Enable

Table 7.6 lists and defines the settings you can make on the **Trace Unit** page of the EOnCE Configurator.

Table 7.6 Trace Unit Page

Page Item	Explanation
Change of Flow Instructions	Enables a tracing mode that traces the addresses of execution sets containing change of flow instructions (for example, a jump, branch, or return to subroutine instruction). The CodeWarrior IDE places the address of the first instruction of such an execution set in the trace buffer.
Interrupt Vectors	Enables a tracing mode that traces the address of interrupt vectors. When enabled, each service of an interrupt places the following items in the trace buffer: • The address of the last executed execution set (before the interrupt) • The address of the interrupt vector
Issue of Execution Set	Enables a tracing mode that traces the addresses of every issued execution set. The only entry written to the trace buffer while tracing in this mode is the first address of each execution set.

Table 7.6 Trace Unit Page (*continued*)

Page Item	Explanation
MARK Instruction	Enables the EOnCE MARK instruction, which writes the PC (program counter) to the trace buffer if the trace buffer is enabled.
Wrap-around Mode for Events	Select this option to prevent the core from going into debug mode when the trace buffer is full. If this option is selected, trace events wrap-around in the trace buffer, overwriting oldest events first. If this option is *not* selected, all trace events are captured by the debugger. When the trace buffer is full, the core enters debug mode. The debugger detects this condition, retrieves the trace events, and resumes core execution. While this captures all trace events, it can slow core execution dramatically if the trace buffer is fills up rapidly. **NOTE:** Each time the core enters debug mode, the debugger retrieves all information currently in the trace buffer. If wrap-around mode is enabled and the trace buffer happens to be full when the debugger retrieves a trace, there will be gaps of missing trace information displayed in the trace view window. Therefore, to avoid confusion, it is suggested that you flush the trace buffer before continuing execution when wrap-around mode is enabled.
Trace Buffer Mode	Enables the trace buffer so that it collects data.
Hardware Loop	Enables a tracing mode that traces the addresses of hardware loops. Every change of flow resulting from a loop puts the address of the last address into the trace buffer.
Record Event Counter	Enables a tracing mode that causes each destination address placed in the trace buffer to be followed immediately by the value of the event counter register. If you enable Buffer Counter and Buffer Extension Counter at the same time, the value of the event counter register precedes the value of the extension counter register in the trace buffer.
Record Ext. Event Counter	Enables a tracing mode that causes each destination address placed in the trace buffer to be followed immediately by the value of the extension counter register.

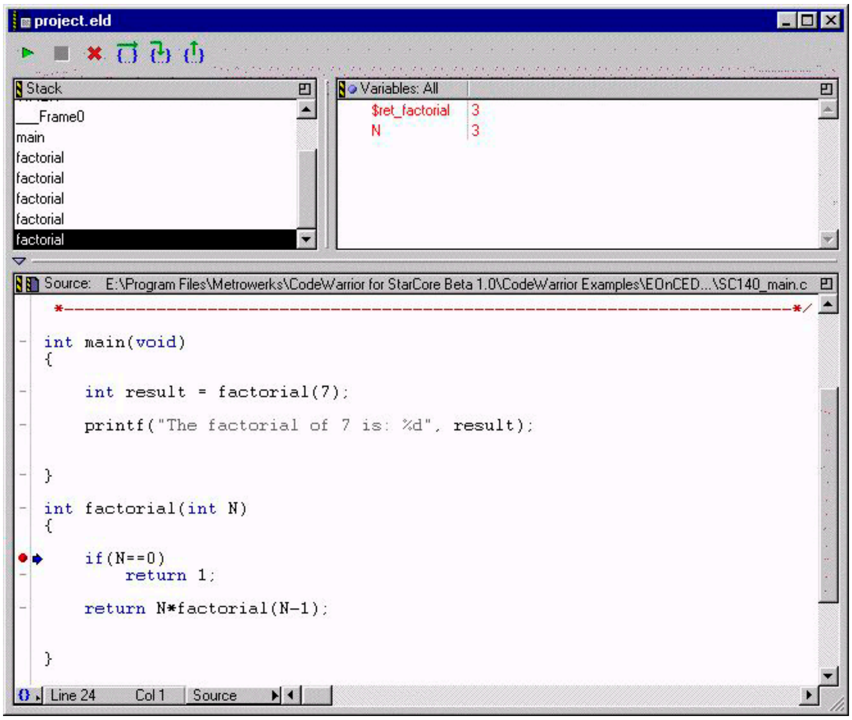
EOnCE Example: Counting Factorial Function Calls

This example shows how to count calls to a factorial function in a recursive factorial program.

To run the factorial program with an input of 7 and count the calls to the `factorial` function five times using a regular software breakpoint, you would set a breakpoint on the first line of the factorial function. Each time the IDE reaches the breakpoint, it stops and you must click the debug button to continue execution. This is a time-consuming process.

Figure 7.7 shows the debugger window after counting the call to `factorial` five times using a regular software breakpoint.

Figure 7.7 Debugger Window: Counting with a Regular Software Breakpoint


project.eld
Stack
Variables: All
Source: E:\Program Files\Metrowerks\CodeWarrior for StarCore Beta 1.0\CodeWarrior Examples\EOnCED...\SC140_main.c
Line 24
Col 1
Source

```
___Frame0
main
factorial
factorial
factorial
factorial
factorial
```

$ret_factorial	3
N	3

```
*------------------------------------------------------------------------*/

int main(void)
{

    int result = factorial(7);

    printf("The factorial of 7 is: %d", result);

}

int factorial(int N)
{

    if(N==0)
        return 1;

    return N*factorial(N-1);

}
```

However, when you use EOnCE, you can pre-set the condition (count the call to the function five times) and location at which you want the EOnCE module to count the call to the function. The count occurs automatically each time execution reaches that location.

After the fifth call, the program stops executing and the IDE enters debug mode. The example in this section discusses how to set up this condition using EOnCE. After you set up the condition, you can execute much faster than by starting the program running again each time it stops on the breakpoint.

This example covers these topics, which you must perform in the listed order:

1. Open the EOnCEDemo project.
2. Download the EOnCEDemo project.
3. Get the address of the instruction.
4. Open the EOnCE Configurator.
5. Configure the **Address Event Detection Channel 0** page.
6. Configure the **Event Counter** page.
7. Configure the **Event Selector** page.
8. Save EOnCE Configurator settings.
9. Run the EOnCE factorial count debugging example.

Open the EOnCEDemo Project

To open the EOnCEDemo project:

1. If needed, open the CodeWarrior software.
2. Choose **File** > **Open**.
3. Navigate to the following directory:

Windows

`installDir\Examples\StarCore\EOnCEDemo`

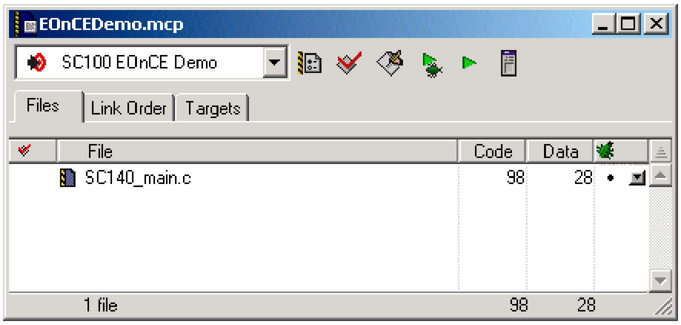
Solaris

`installDir/CodeWarrior_ver_dir/`
`CodeWarrior_Examples/EOnCEDemo`

4. Select the project file (`EOnCEDemo.mcp`).
5. Click **Open**

   When you open the project, the IDE displays a project window. (See Figure 7.8.)

Figure 7.8 EOnCEDemo Project Window



File	Code	Data
SC140_main.c	98	28
1 file	98	28

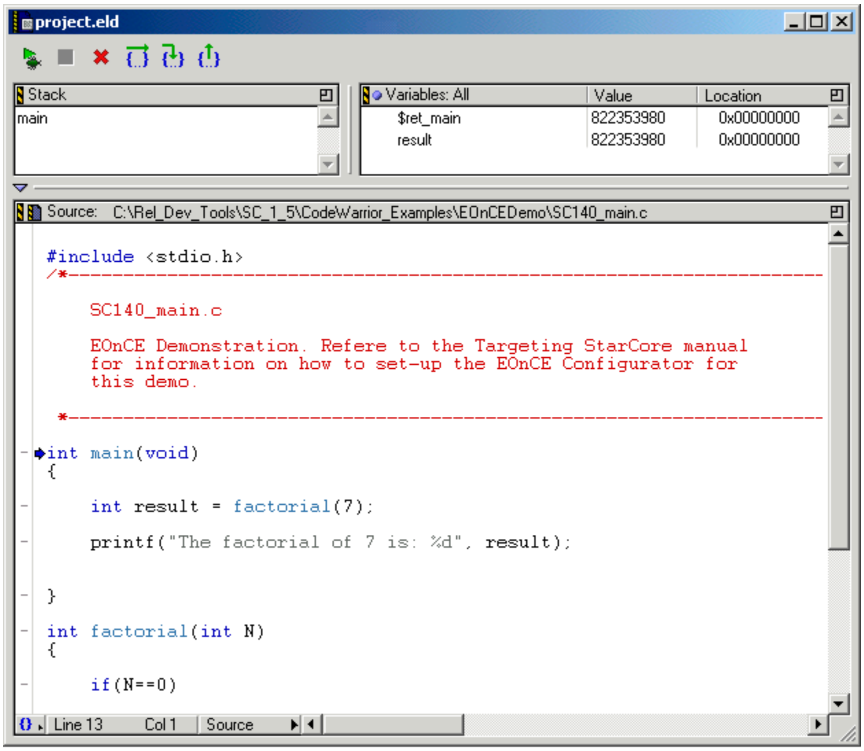
Download the EOnCEDemo Project

To download the EOnCEDemo project, choose **Project > Debug**.

The IDE downloads the project to the target board, and the debugger window appears as shown in Figure 7.9.

NOTE You must download your program to the target board before configuring EOnCE debugging conditions.

Figure 7.9 Debugger Window for the EOnCEDemo Project


project.eld
Stack
Variables: All
Value
Location
Source: C:\Rel_Dev_Tools\SC_1_5\CodeWarrior_Examples\EOnCEDemo\SC140_main.c
Line 13
Col 1
Source

```
main
```

Variables: All	Value	Location
$ret_main	822353980	0x00000000
result	822353980	0x00000000

```
#include <stdio.h>
/*------------------------------------------------------------------------

    SC140_main.c

    EOnCE Demonstration. Refere to the Targeting StarCore manual
    for information on how to set-up the EOnCE Configurator for
    this demo.

 *------------------------------------------------------------------------

int main(void)
{

    int result = factorial(7);

    printf("The factorial of 7 is: %d", result);


}
int factorial(int N)
{

    if(N==0)
```

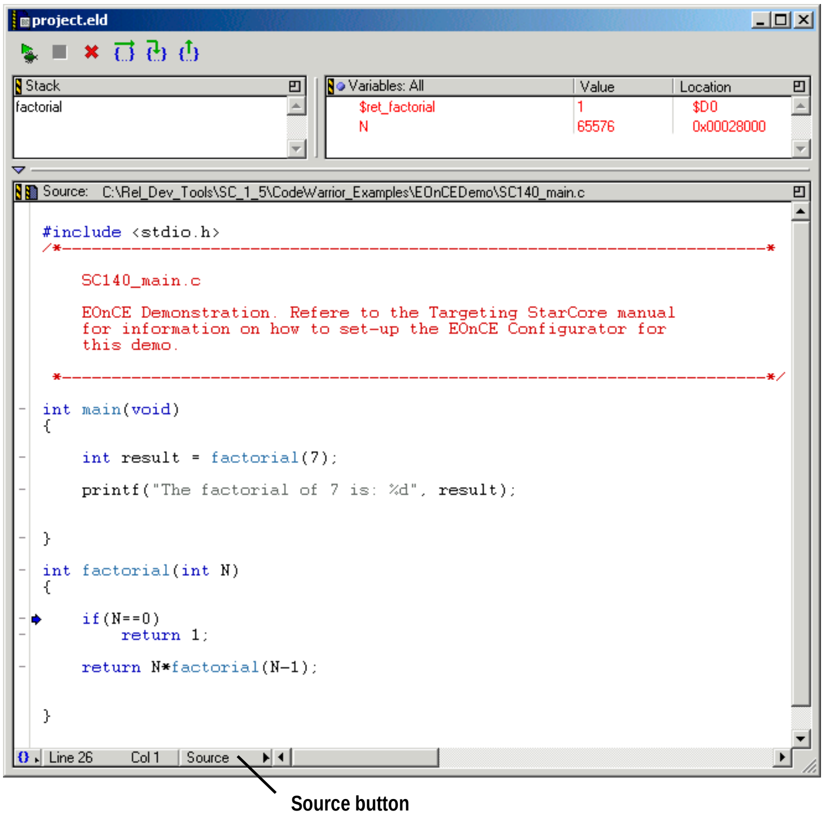
Get the Address of the Instruction

To get the address of the instruction to specify as the location where EOnCE counts the call to the factorial function:

1. In the debugger window, move the Current Statement arrow to the first line of the factorial function (the instruction where you want to set the breakpoint).

   Figure 7.10 shows the debugger window after you move the arrow.

Figure 7.10 Current Statement Arrow: First Line of the Factorial Function



Stack
factorial

Variables: All	Value	Location
$ret_factorial	1	$D0
N	65576	0x00028000

Source: C:\Rel_Dev_Tools\SC_1_5\CodeWarrior_Examples\EOnCEDemo\SC140_main.c

```
#include <stdio.h>
/*-----------------------------------------------------------------------*

    SC140_main.c

    EOnCE Demonstration. Refere to the Targeting StarCore manual
    for information on how to set-up the EOnCE Configurator for
    this demo.

 *-----------------------------------------------------------------------*/

int main(void)
{

    int result = factorial(7);

    printf("The factorial of 7 is: %d", result);

}

int factorial(int N)
{

    if(N==0)
        return 1;

    return N*factorial(N-1);

}
```

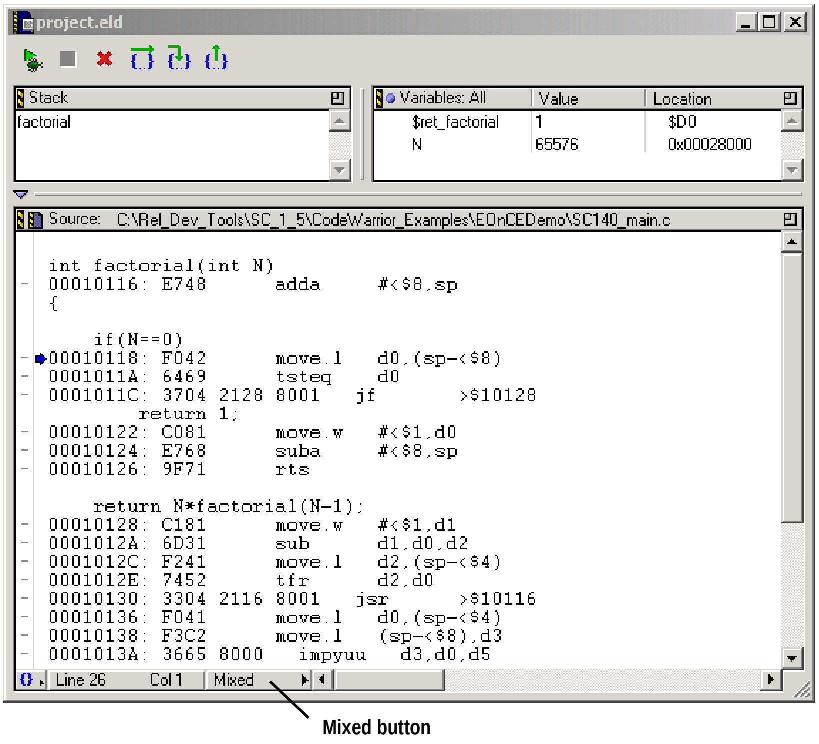
Line 26 Col 1 Source

2. At the bottom of the debugger window, click the **Source** button, then choose **Mixed** from the menu that appears.

   The caption of the button changes to **Mixed** and the debugger switches to a mixed language view. In this view, the debugger displays each C language statement of your program followed by the assembly language instructions the compiler generates for each statement.

   The Current Statement arrow now points to the address of the assembly-language instruction on which to set a breakpoint. (See Figure 7.11.)

Figure 7.11 Debugger Window Displaying a Mixed Code View



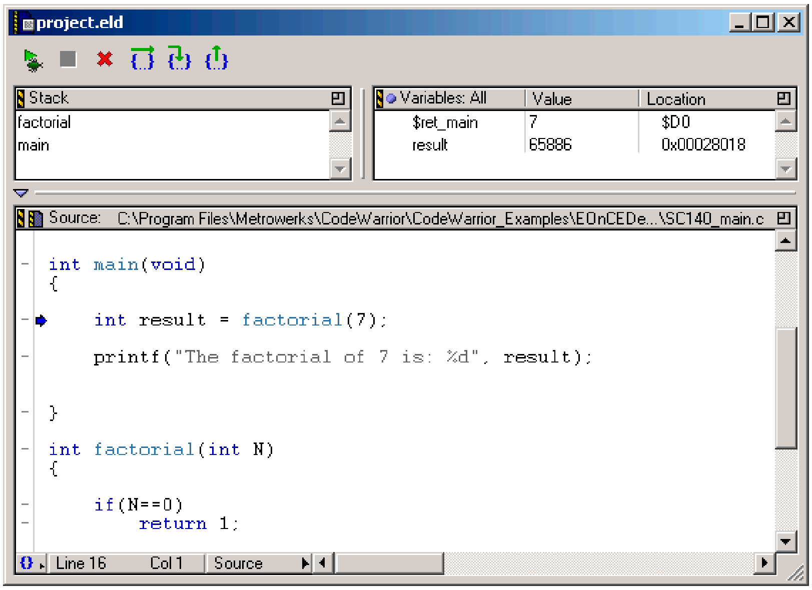
```
int factorial(int N)
00010116: E748       adda     #<$8,sp
{

    if(N==0)
00010118: F042       move.l   d0,(sp-<$8)
0001011A: 6469       tsteq    d0
0001011C: 3704 2128 8001   jf       >$10128
        return 1;
00010122: C081       move.w   #<$1,d0
00010124: E768       suba     #<$8,sp
00010126: 9F71       rts

    return N*factorial(N-1);
00010128: C181       move.w   #<$1,d1
0001012A: 6D31       sub      d1,d0,d2
0001012C: F241       move.l   d2,(sp-<$4)
0001012E: 7452       tfr      d2,d0
00010130: 3304 2116 8001   jsr      >$10116
00010136: F041       move.l   d0,(sp-<$4)
00010138: F3C2       move.l   (sp-<$8),d3
0001013A: 3665 8000   impyuu   d3,d0,d5
```

3. Write down the address value of the instruction immediately preceding the location to which the Current Statement arrow now points.
4. At the bottom of the debugger window, click the **Mixed** button, then choose **Source** from the menu that appears.

   The caption of the button changes to **Source** and the debugger switches to a source language view. In this view, just the C language statement of your program are displayed.
5. Move the Current Statement arrow to the first statement in the `main` function.

   Figure 7.12 shows the debugger window after you perform these actions.

Figure 7.12 Current Statement Arrow on the First Statement in main()



project.eld
Stack
Variables: All
Value
Location
Source:
C:\Program Files\Metrowerks\CodeWarrior\CodeWarrior_Examples\EOnCEDe...\SC140_main.c
Line 16
Col 1
Source

Stack
factorial
main

Variables: All	Value	Location
$ret_main	7	$D0
result	65886	0x00028018

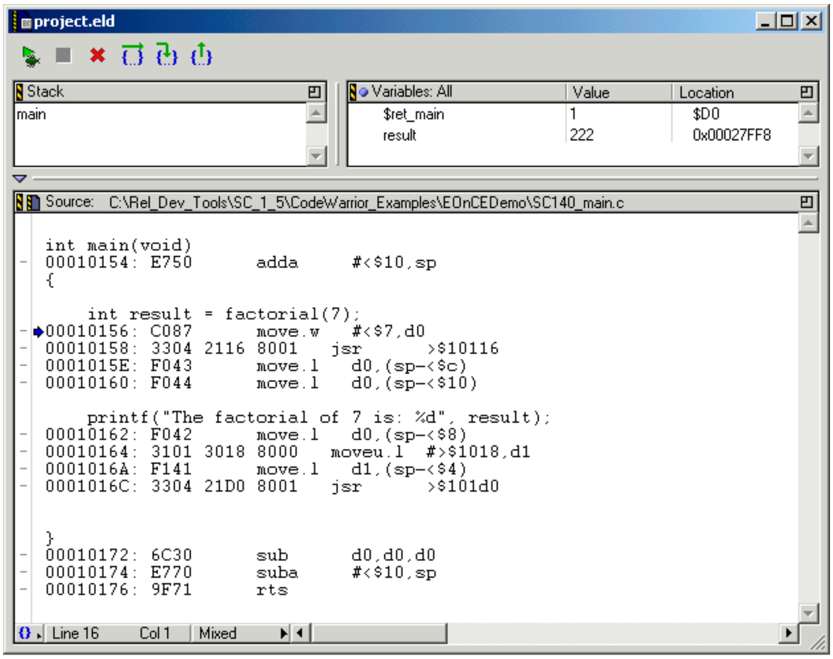
```
int main(void)
{
    int result = factorial(7);

    printf("The factorial of 7 is: %d", result);


}
int factorial(int N)
{
    if(N==0)
        return 1;
```

6. At the bottom of the debugger window, click the **Source** button, then choose **Mixed** from the menu that appears.

   The caption of the button changes to **Mixed** and the debugger again switches to mixed language view.

   Figure 7.13 shows the debugger window after you perform these actions.

Figure 7.13 Debugger Window—Mixed Code View



Stack
main

Variables: All	Value	Location
$ret_main	1	$D0
result	222	0x00027FF8

Source: C:\Rel_Dev_Tools\SC_1_5\CodeWarrior_Examples\EOnCEDemo\SC140_main.c

```
int main(void)
00010154: E750       adda     #<$10,sp
{

    int result = factorial(7);
00010156: C087       move.w   #<$7,d0
00010158: 3304 2116 8001   jsr      >$10116
0001015E: F043       move.l   d0,(sp-<$c)
00010160: F044       move.l   d0,(sp-<$10)

    printf("The factorial of 7 is: %d", result);
00010162: F042       move.l   d0,(sp-<$8)
00010164: 3101 3018 8000   moveu.l  #>$1018,d1
0001016A: F141       move.l   d1,(sp-<$4)
0001016C: 3304 21D0 8001   jsr      >$101d0


}
00010172: 6C30       sub      d0,d0,d0
00010174: E770       suba     #<$10,sp
00010176: 9F71       rts
```

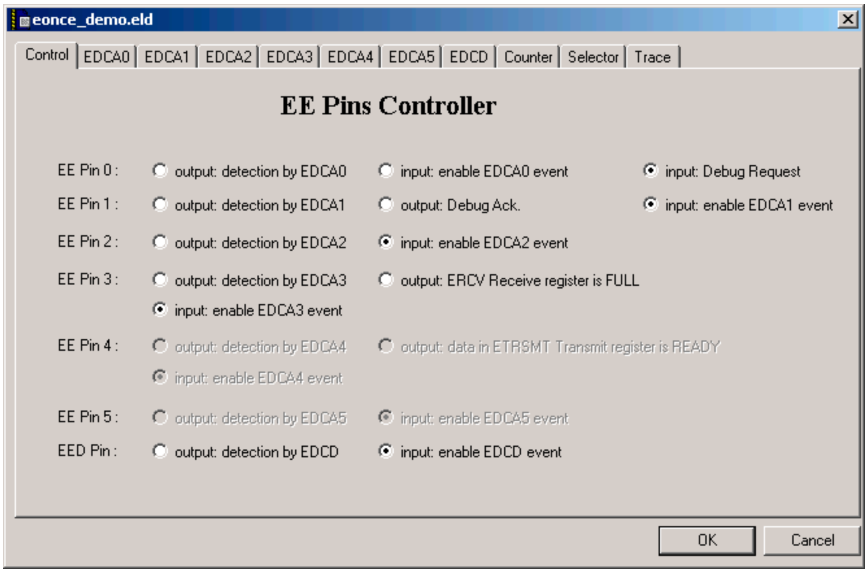
Open the EOnCE Configurator

You can use the EOnCE Configurator to set various conditions to perform EOnCE debugging.

To open the EOnCE Configurator, choose **Debug** > **EOnCE** > **EOnCE Configurator**.

The IDE displays the **EOnCE Configurator** window. (See Figure 7.14.)

Figure 7.14 EE Pins Controller Page



eonce_demo.eld

Control | EDCA0 | EDCA1 | EDCA2 | EDCA3 | EDCA4 | EDCA5 | EDCD | Counter | Selector | Trace

EE Pins Controller

EE Pin 0 : output: detection by EDCA0 | input: enable EDCA0 event | [✓ input: Debug Request]

EE Pin 1 : output: detection by EDCA1 | output: Debug Ack. | [✓ input: enable EDCA1 event]

EE Pin 2 : output: detection by EDCA2 | [✓ input: enable EDCA2 event]

EE Pin 3 : output: detection by EDCA3 | output: ERCV Receive register is FULL | [✓ input: enable EDCA3 event]

EE Pin 4 : output: detection by EDCA4 | output: data in ETRSMT Transmit register is READY | [✓ input: enable EDCA4 event]

EE Pin 5 : output: detection by EDCA5 | [✓ input: enable EDCA5 event]

EED Pin : output: detection by EDCD | [✓ input: enable EDCD event]

The window initially displays the **EE Pins Controller** page, which you can use to configure the EOnCE controller, specifically the EE pins. EE pins are general-purpose pins that can serve as input or output pins to the EOnCE.

NOTE For this example, the settings on the **EE Pins Controller** page are not relevant; do not change them.

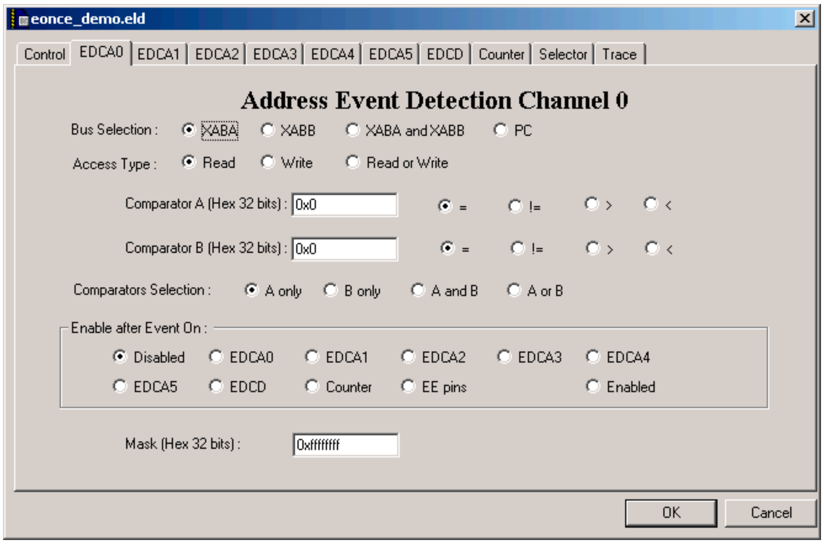
Configure the Address Event Detection Channel 0 Page

To choose settings for the address event detection channel 0 by configuring the **Address Event Detection Channel 0** page:

1. Click the **EDCA0** tab.

   The IDE displays the chosen page. (See Figure 7.15).

Figure 7.15 Address Event Detection Channel 0 Page


eonce_demo.eld
Control | EDCA0 | EDCA1 | EDCA2 | EDCA3 | EDCA4 | EDCA5 | EDCD | Counter | Selector | Trace
OK
Cancel

Address Event Detection Channel 0

Bus Selection : [✓ XABA] XABB XABA and XABB PC

Access Type : [✓ Read] Write Read or Write

Comparator A (Hex 32 bits) : 0x0 [✓ =] != > <

Comparator B (Hex 32 bits) : 0x0 [✓ =] != > <

Comparators Selection : [✓ A only] B only A and B A or B

Enable after Event On : [✓ Disabled] EDCA0 EDCA1 EDCA2 EDCA3 EDCA4 EDCA5 EDCD Counter EE pins Enabled

Mask (Hex 32 bits) : 0xffffffff

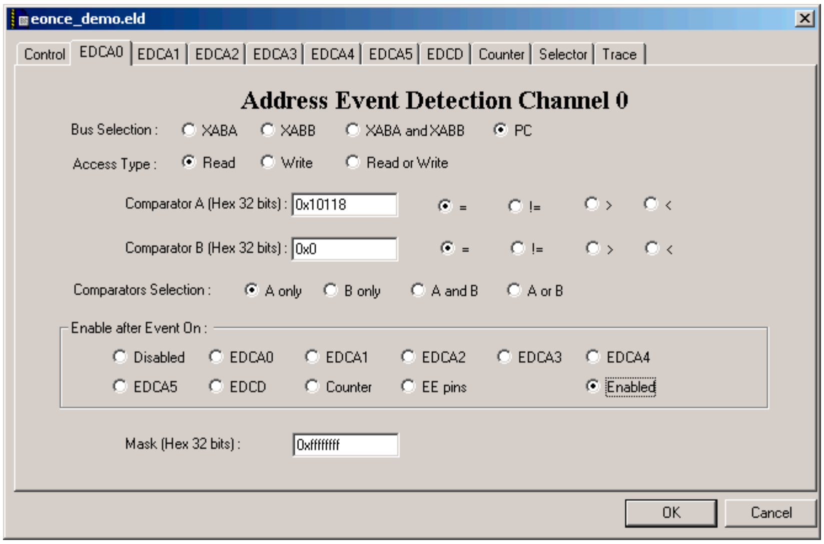
2. To set a breakpoint on an instruction, click **PC** as the **Bus Selection**.

NOTE When selecting settings in the EOnCE Configurator, configure the tabbed pages in the left-to-right order of the tabs. For example, configure the **Address Event Detection Channel 0** page before configuring the **Event Counter** page. Also, within a page, configure your selected settings from the left-top to right-bottom position.

3. Type the address value of the instruction you previously noted in the **Comparator A** field (in hexadecimal).
4. For **Enable after Event On**, select **Enabled**.

   For all other settings, use the defaults. The page now appears as shown in Figure 7.16.

Figure 7.16 Address Event Detection Channel 0 after Changing Settings


eonce_demo.eld
Control | EDCA0 | EDCA1 | EDCA2 | EDCA3 | EDCA4 | EDCA5 | EDCD | Counter | Selector | Trace
OK
Cancel

Address Event Detection Channel 0

Bus Selection : XABA XABB XABA and XABB [✓ PC]

Access Type : [✓ Read] Write Read or Write

Comparator A (Hex 32 bits) : 0x10118 [✓ =] != > <

Comparator B (Hex 32 bits) : 0x0 [✓ =] != > <

Comparators Selection : [✓ A only] B only A and B A or B

Enable after Event On : Disabled EDCA0 EDCA1 EDCA2 EDCA3 EDCA4 EDCA5 EDCD Counter EE pins [✓ Enabled]

Mask (Hex 32 bits) : 0xffffffff

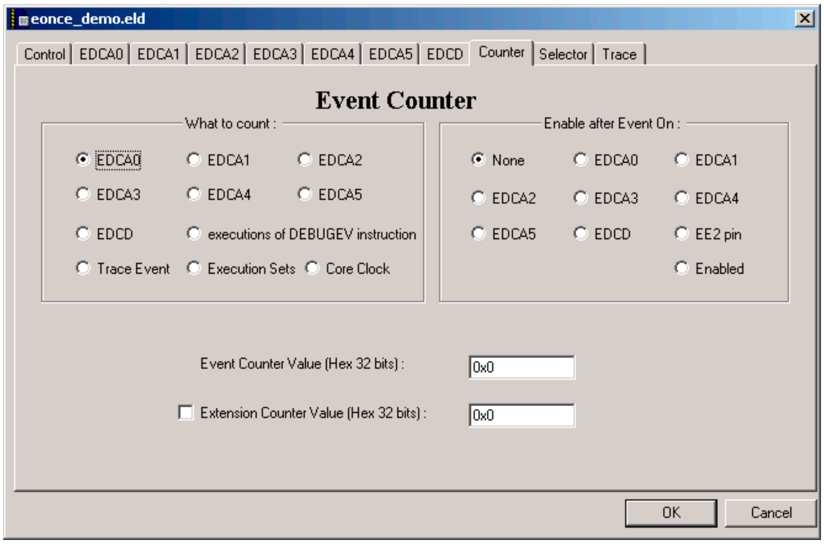
Configure the Event Counter Page

To configure the **Event Counter** page to count a particular event on address channel 0:

1. Click the **Counter** tab.

   The IDE displays the **Event Counter** page. (See Figure 7.17.)

Figure 7.17 Event Counter Page



Event Counter

What to count : [✓ EDCA0] EDCA1 EDCA2 EDCA3 EDCA4 EDCA5 EDCD executions of DEBUGEV instruction Trace Event Execution Sets Core Clock

Enable after Event On : [✓ None] EDCA0 EDCA1 EDCA2 EDCA3 EDCA4 EDCA5 EDCD EE2 pin Enabled

Event Counter Value (Hex 32 bits) : 0x0

☐ Extension Counter Value (Hex 32 bits) : 0x0

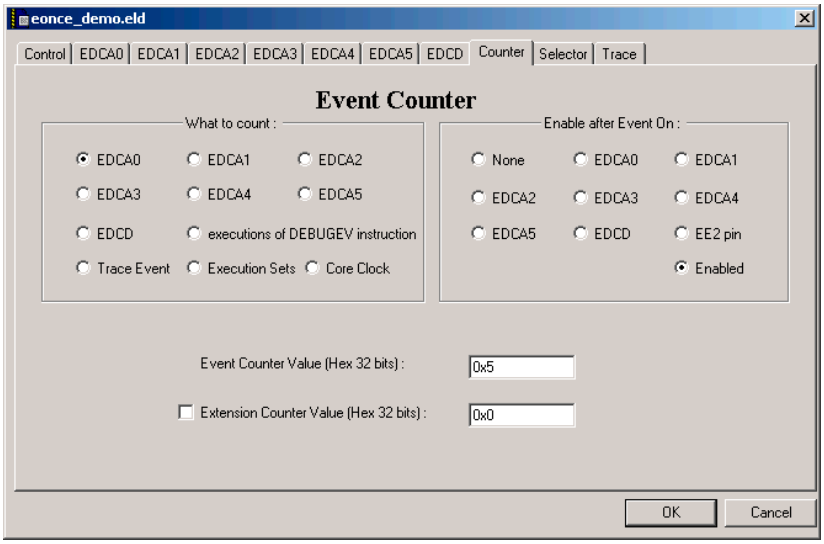
In its default state, the **Event Counter** page specifies to count EDC0 (address channel 0, which corresponds with the **Address Event Detection Channel 0** page that you just configured and is correct for this example).

2. For **Enable after Event On**, select **Enabled**.
3. Type the value (in hexadecimal) for how many times you want to count in the **Event Counter Value (Hex 32 bits)** text box.

   For this example, type `0x5` in the text box.

   Figure 7.18 shows the **Event Counter** page after your changes.

Figure 7.18 Event Counter Page after Changing Settings



Event Counter

What to count :
[✓ EDCA0] EDCA1 EDCA2
EDCA3 EDCA4 EDCA5
EDCD executions of DEBUGEV instruction
Trace Event Execution Sets Core Clock

Enable after Event On :
None EDCA0 EDCA1
EDCA2 EDCA3 EDCA4
EDCA5 EDCD EE2 pin
[✓ Enabled]

Event Counter Value (Hex 32 bits) : 0x5

☐ Extension Counter Value (Hex 32 bits) : 0x0

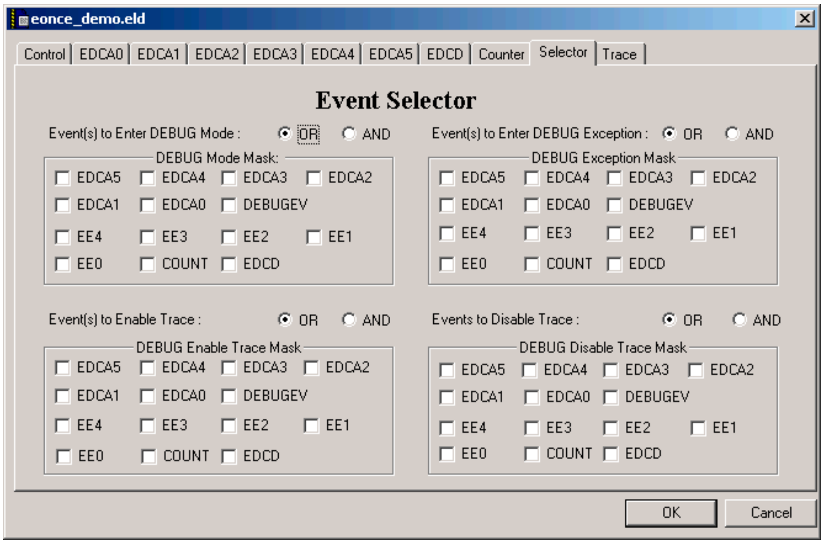
Configure the Event Selector Page

To configure the **Event Selector** page:

1. Click the **Selector** tab.

   The IDE displays the **Event Selector** page. (See Figure 7.19.)

Figure 7.19 Event Selector Page



eonce_demo.eld

Control | EDCA0 | EDCA1 | EDCA2 | EDCA3 | EDCA4 | EDCA5 | EDCD | Counter | Selector | Trace

Event Selector

Event(s) to Enter DEBUG Mode : [✓ OR] AND

DEBUG Mode Mask:

☐ EDCA5	☐ EDCA4	☐ EDCA3	☐ EDCA2
☐ EDCA1	☐ EDCA0	☐ DEBUGEV	
☐ EE4	☐ EE3	☐ EE2	☐ EE1
☐ EE0	☐ COUNT	☐ EDCD	

Event(s) to Enter DEBUG Exception : [✓ OR] AND

DEBUG Exception Mask

☐ EDCA5	☐ EDCA4	☐ EDCA3	☐ EDCA2
☐ EDCA1	☐ EDCA0	☐ DEBUGEV	
☐ EE4	☐ EE3	☐ EE2	☐ EE1
☐ EE0	☐ COUNT	☐ EDCD	

Event(s) to Enable Trace : [✓ OR] AND

DEBUG Enable Trace Mask

☐ EDCA5	☐ EDCA4	☐ EDCA3	☐ EDCA2
☐ EDCA1	☐ EDCA0	☐ DEBUGEV	
☐ EE4	☐ EE3	☐ EE2	☐ EE1
☐ EE0	☐ COUNT	☐ EDCD	

Events to Disable Trace : [✓ OR] AND

DEBUG Disable Trace Mask

☐ EDCA5	☐ EDCA4	☐ EDCA3	☐ EDCA2
☐ EDCA1	☐ EDCA0	☐ DEBUGEV	
☐ EE4	☐ EE3	☐ EE2	☐ EE1
☐ EE0	☐ COUNT	☐ EDCD	

OK | Cancel

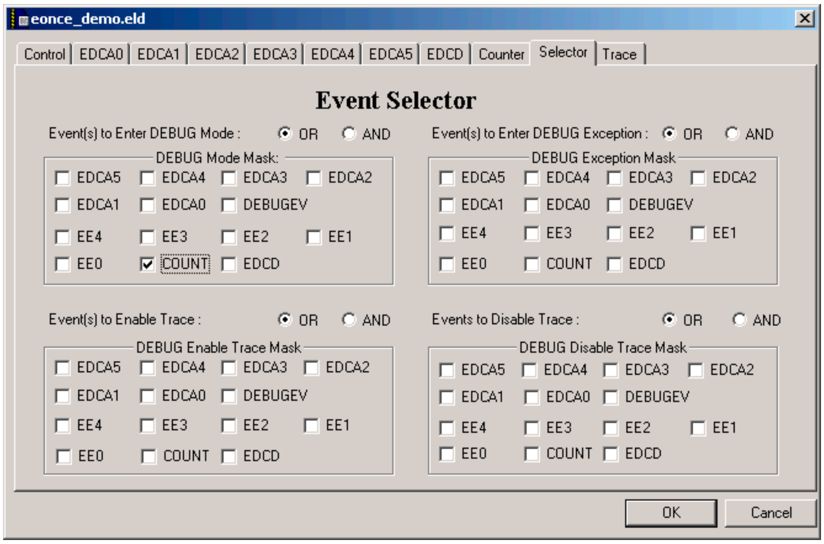
The default setting for **Event(s) to Enter DEBUG Mode** is **OR**, which is correct for this example.

2. In the **DEBUG Mode Mask** group, check the **COUNT** checkbox.

   The **Event(s) to Enter DEBUG Mode** setting and the **DEBUG Mode Mask** setting halt the CPU and cause the EOnCE module to enter debug mode when the condition or conditions that you set are met.

   Figure 7.20 shows the appearance of the **Event Selector** page after this change.

Figure 7.20 Event Selector Page after Changing Settings



Event Selector

Event(s) to Enter DEBUG Mode : [✓ OR] AND

DEBUG Mode Mask:
EDCA5, EDCA4, EDCA3, EDCA2, EDCA1, EDCA0, DEBUGEV, EE4, EE3, EE2, EE1, EE0, [✓ COUNT], EDCD

Event(s) to Enter DEBUG Exception : [✓ OR] AND

DEBUG Exception Mask
EDCA5, EDCA4, EDCA3, EDCA2, EDCA1, EDCA0, DEBUGEV, EE4, EE3, EE2, EE1, EE0, COUNT, EDCD

Event(s) to Enable Trace : [✓ OR] AND

DEBUG Enable Trace Mask
EDCA5, EDCA4, EDCA3, EDCA2, EDCA1, EDCA0, DEBUGEV, EE4, EE3, EE2, EE1, EE0, COUNT, EDCD

Events to Disable Trace : [✓ OR] AND

DEBUG Disable Trace Mask
EDCA5, EDCA4, EDCA3, EDCA2, EDCA1, EDCA0, DEBUGEV, EE4, EE3, EE2, EE1, EE0, COUNT, EDCD

Save the EOnCE Configurator Settings

To save your changes, click **OK** in the **EOnCE Configurator** window.

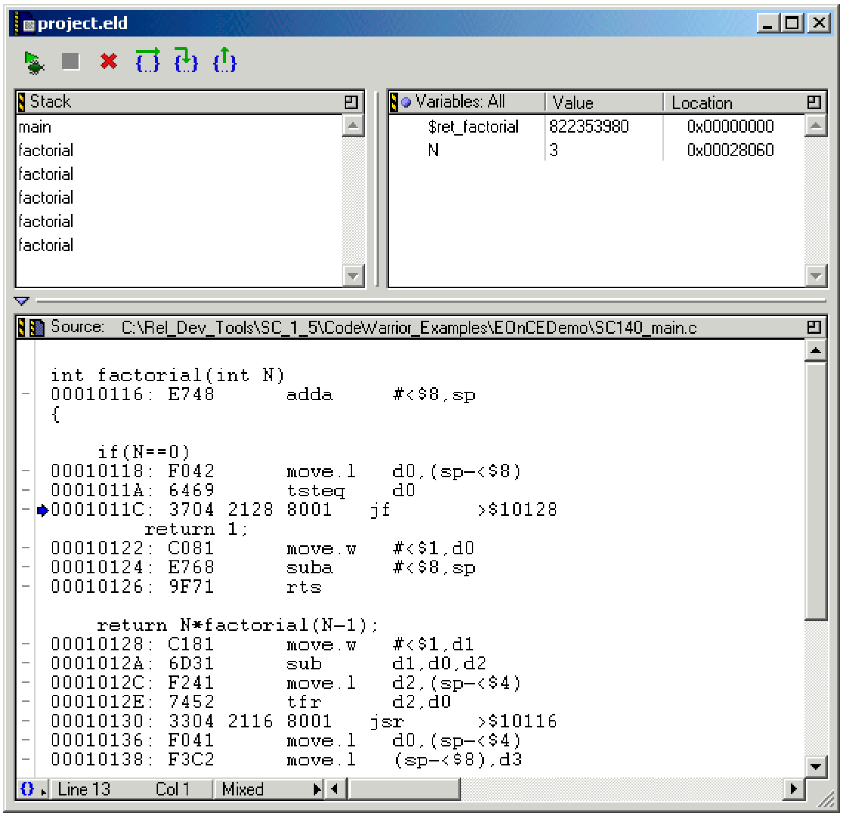
Run the EOnCE Factorial Count Debugging Example

To run the EOnCE debugging example, select **Project** > **Run**.

The debugger executes the program, and five calls to the factorial function appear in the Stack Crawl pane before the program stops running and enters debug mode.

Figure 7.21 shows the appearance of the debugger window after you run the debugging example.

Figure 7.21 Debugger Window after Running the Debugging Example


project.eld
Stack
Variables: All
Value
Location
Source: C:\Rel_Dev_Tools\SC_1_5\CodeWarrior_Examples\EOnCEDemo\SC140_main.c
Line 13
Col 1
Mixed

Stack
main
factorial
factorial
factorial
factorial
factorial

Variables: All	Value	Location
$ret_factorial	822353980	0x00000000
N	3	0x00028060

```
int factorial(int N)
00010116: E748          adda     #<$8,sp
{

    if(N==0)
00010118: F042          move.l   d0,(sp-<$8)
0001011A: 6469          tsteq    d0
0001011C: 3704 2128 8001    jf        >$10128
        return 1;
00010122: C081          move.w   #<$1,d0
00010124: E768          suba     #<$8,sp
00010126: 9F71          rts

    return N*factorial(N-1);
00010128: C181          move.w   #<$1,d1
0001012A: 6D31          sub      d1,d0,d2
0001012C: F241          move.l   d2,(sp-<$4)
0001012E: 7452          tfr      d2,d0
00010130: 3304 2116 8001    jsr       >$10116
00010136: F041          move.l   d0,(sp-<$4)
00010138: F3C2          move.l   (sp-<$8),d3
```

In this example, the program halted in debug mode; therefore, you can continue debugging from that point.

EOnCE Example: Using the Trace Buffer

This example shows how to capture data in the EOnCE trace buffer and examine it.

This section includes the following topics, which you must perform in the listed order:

1. Open the EOnCEDemo project
2. Download the EOnCEDemo project
3. Set a breakpoint
4. Run to the breakpoint

5. Open the EOnCE Configurator
6. Configure a trace
7. Save EOnCE Configurator settings
8. Run the EOnCE trace buffer debugging example

Open the EOnCEDemo Project

To open the EOnCEDemo project:

1. Start the CodeWarrior IDE.
2. Choose **File** > **Open**
3. Navigate to the following directory:

Windows

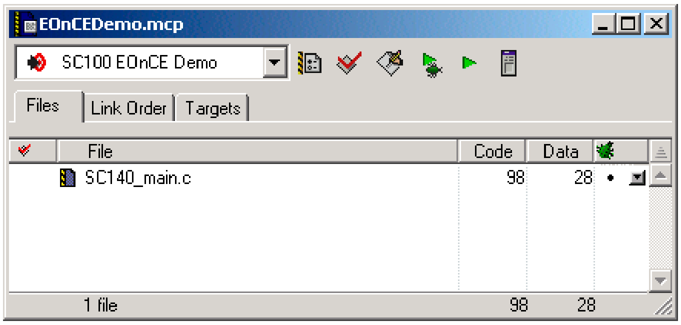
`*installDir*\Examples\StarCore\EOnCEDemo`

Solaris

```
installDir/CodeWarrior_ver_dir/
CodeWarrior_Examples/EOnCEDemo
```

4. Select the project file (`EOnCEDemo.mcp`).
5. Click **Open**
6. The IDE displays a project window. (See Figure 7.22.)

Figure 7.22 EOnCEDemo.mcp Project Window



File	Code	Data
SC140_main.c	98	28
1 file	98	28

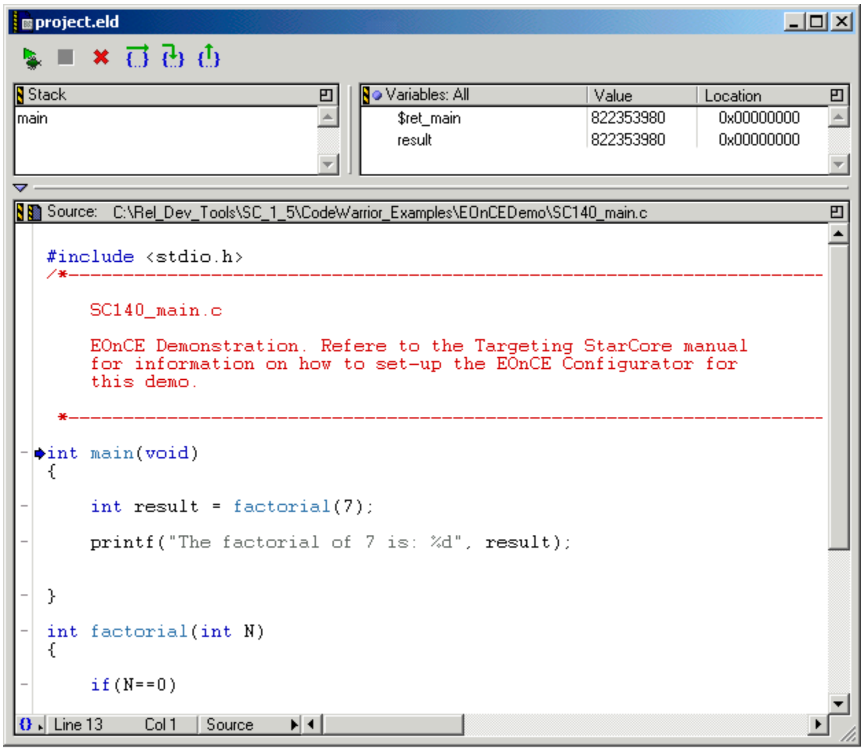
Download the EOnCEDemo Project

To download the EOnCEDemo project, choose **Project > Debug**.

The IDE downloads the project to the target board, and the debugger window appears as shown in Figure 7.23.

NOTE You must download your program to the target board before configuring EOnCE debugging conditions.

Figure 7.23 Debugger Window for the EOnCEDemo Project



```
$ret_main    822353980    0x00000000
result       822353980    0x00000000
```

Source: C:\Rel_Dev_Tools\SC_1_5\CodeWarrior_Examples\EOnCEDemo\SC140_main.c

```
#include <stdio.h>
/*-----------------------------------------------------------------------

    SC140_main.c

    EOnCE Demonstration. Refere to the Targeting StarCore manual
    for information on how to set-up the EOnCE Configurator for
    this demo.

 *-----------------------------------------------------------------------

int main(void)
{

    int result = factorial(7);

    printf("The factorial of 7 is: %d", result);

}

int factorial(int N)
{

    if(N==0)
```

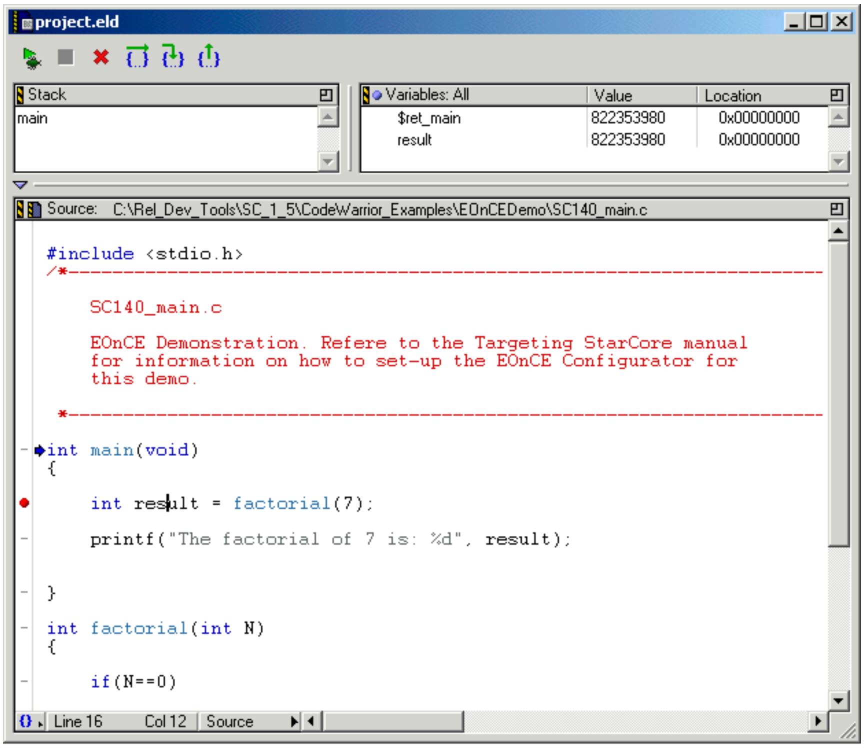
Set a Breakpoint

To set a breakpoint, click the gray dash next to the this statement in the debugger window:

```
int result = factorial (7);
```

Figure 7.24 shows the debugger window after you set the breakpoint.

Figure 7.24 Debugger Window after Setting the Breakpoint



project.eld

Stack
main

Variables: All	Value	Location
$ret_main	822353980	0x00000000
result	822353980	0x00000000

Source: C:\Rel_Dev_Tools\SC_1_5\CodeWarrior_Examples\EOnCEDemo\SC140_main.c

```
#include <stdio.h>
/*-------------------------------------------------------------------

    SC140_main.c

    EOnCE Demonstration. Refere to the Targeting StarCore manual
    for information on how to set-up the EOnCE Configurator for
    this demo.

 *-------------------------------------------------------------------
int main(void)
{

    int result = factorial(7);

    printf("The factorial of 7 is: %d", result);


}
int factorial(int N)
{

    if(N==0)
```

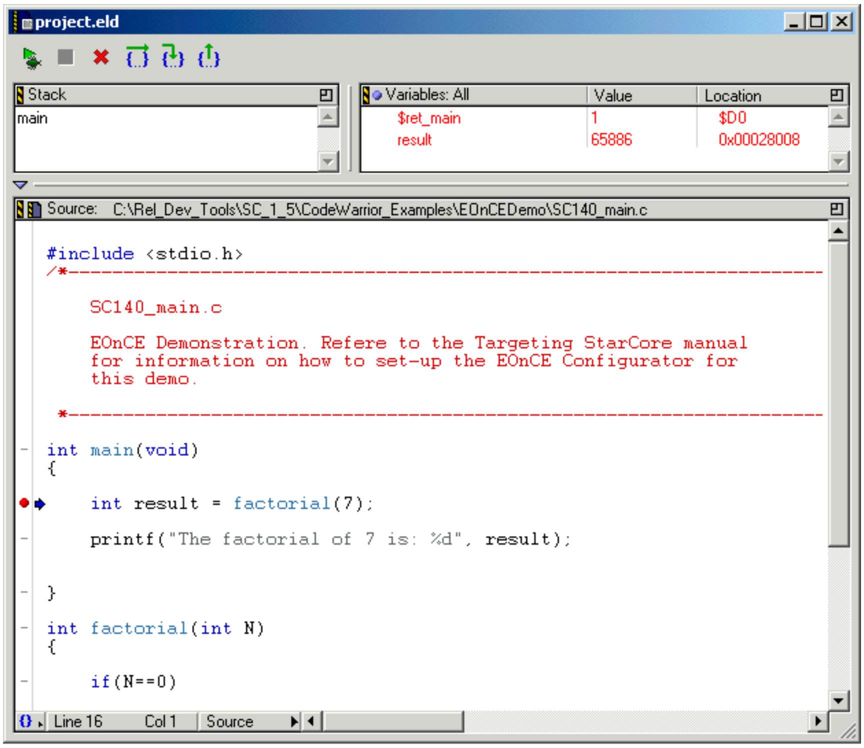
Line 16 Col 12 Source

Run to the Breakpoint

Choose **Project > Run** to run to the breakpoint you previously set.

Figure 7.25 shows the debugger window after running to the breakpoint.

Figure 7.25 Debugger Window after Running to the Breakpoint



main

Variables: All	Value	Location
$ret_main	1	$D0
result	65886	0x00028008

Source: C:\Rel_Dev_Tools\SC_1_5\CodeWarrior_Examples\EOnCEDemo\SC140_main.c

```
#include <stdio.h>
/*-----------------------------------------------------------------------

    SC140_main.c

    EOnCE Demonstration. Refere to the Targeting StarCore manual
    for information on how to set-up the EOnCE Configurator for
    this demo.

 *-----------------------------------------------------------------------
int main(void)
{

    int result = factorial(7);

    printf("The factorial of 7 is: %d", result);


}
int factorial(int N)
{

    if(N==0)
```

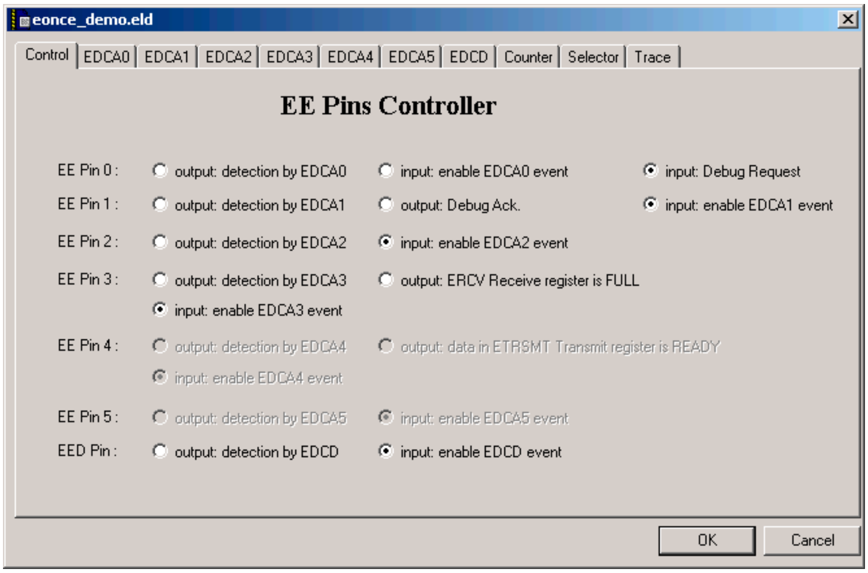
Open the EOnCE Configurator

You can use the EOnCE Configurator to set the various conditions to perform EOnCE debugging.

To open the EOnCE Configurator, choose **Debug** > **EOnCE** > **EOnCE Configurator**.

The IDE displays the **EOnCE Configurator** window. (See Figure 7.26.)

Figure 7.26 EE Pins Controller Page



EE Pins Controller

EE Pin 0 :	output: detection by EDCA0	input: enable EDCA0 event	[✓ input: Debug Request]
EE Pin 1 :	output: detection by EDCA1	output: Debug Ack.	[✓ input: enable EDCA1 event]
EE Pin 2 :	output: detection by EDCA2	[✓ input: enable EDCA2 event]	
EE Pin 3 :	output: detection by EDCA3	output: ERCV Receive register is FULL	
	[✓ input: enable EDCA3 event]		
EE Pin 4 :	output: detection by EDCA4	output: data in ETRSMT Transmit register is READY	
	[✓ input: enable EDCA4 event]		
EE Pin 5 :	output: detection by EDCA5	[✓ input: enable EDCA5 event]	
EED Pin :	output: detection by EDCD	[✓ input: enable EDCD event]	

The window initially displays the **EE Pins Controller** page, which you can use to configure the EOnCE controller, specifically the EE pins. EE pins are general-purpose pins that can serve as input or output pins to the EOnCE.

NOTE For this example, the default settings for the **EE Pins Controller** page shown in Figure 7.26 are correct; do not change them.

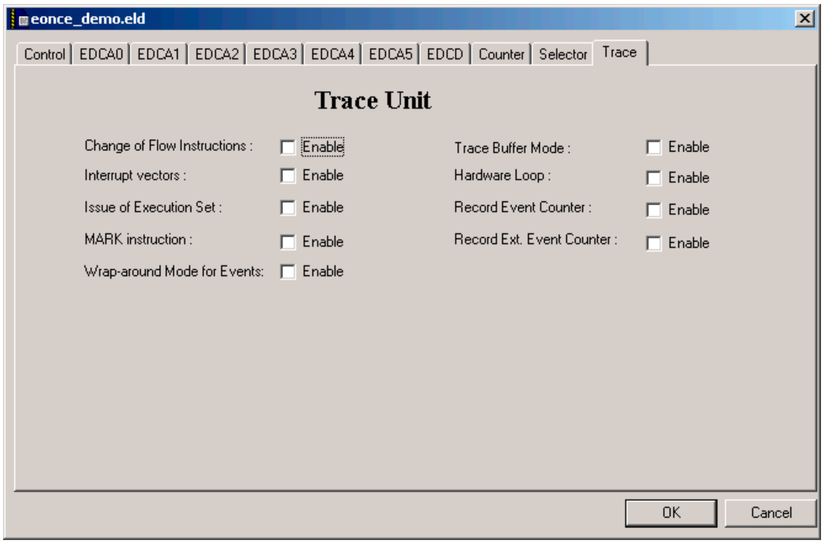
Configure a Trace

To configure the trace for this example:

1. Click the **Trace** tab.

   The IDE displays the **Trace Unit** page. (See Figure 7.27.)

Figure 7.27 EOnCE Configurator Trace Unit Page



Trace Unit

Change of Flow Instructions :	☐ Enable	Trace Buffer Mode :	☐ Enable
Interrupt vectors :	☐ Enable	Hardware Loop :	☐ Enable
Issue of Execution Set :	☐ Enable	Record Event Counter :	☐ Enable
MARK instruction :	☐ Enable	Record Ext. Event Counter :	☐ Enable
Wrap-around Mode for Events:	☐ Enable		

2. Check the **Change of Flow Instructions** checkbox.

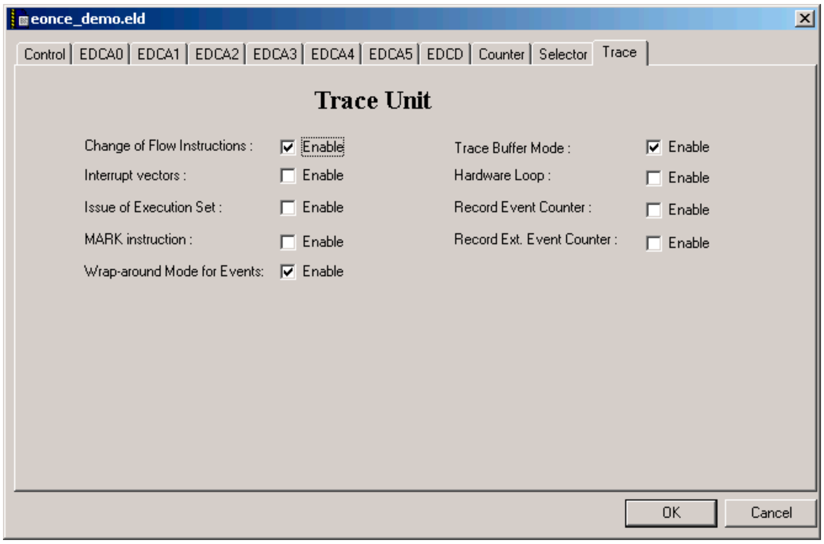
   This enables a trace on anything that changes the instruction flow (for example, a jump, branch, or return to subroutine instruction).

3. Check the **Trace Buffer Mode** checkbox.

4. Check the **Wraparound Mode for Events** checkbox.

   Figure 7.28 shows the **Trace Unit** page after configuration.

Figure 7.28 EOnCE Configurator Trace Unit Page after Configuration



Trace Unit

Change of Flow Instructions : [✓ Enable]
Interrupt vectors : Enable
Issue of Execution Set : Enable
MARK instruction : Enable
Wrap-around Mode for Events: [✓ Enable]
Trace Buffer Mode : [✓ Enable]
Hardware Loop : Enable
Record Event Counter : Enable
Record Ext. Event Counter : Enable

Save the EOnCE Configurator Settings

To save your changes, click **OK** in the **EOnCE Configurator** window.

Run the EOnCE Trace Buffer Debugging Example

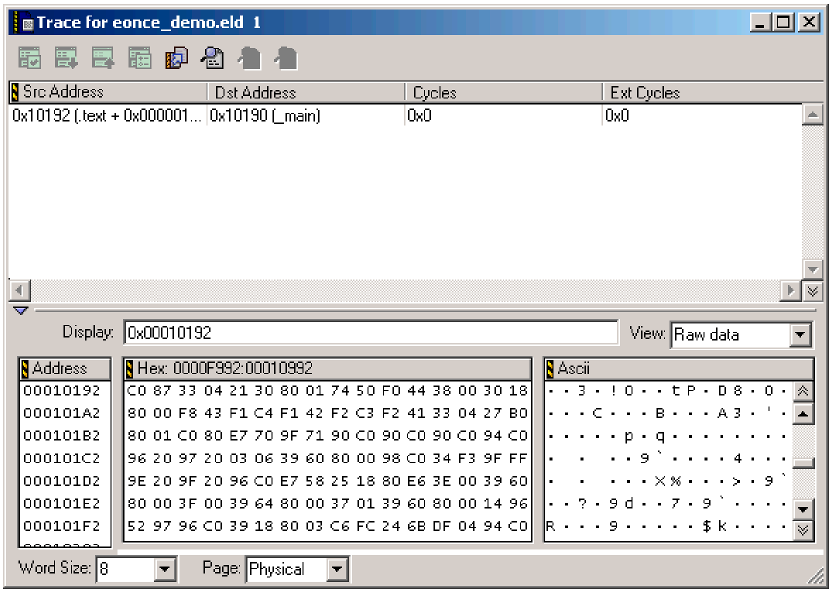
1. To run the EOnCE trace buffer debugging example, choose **Debug** > **Step Over**.

   The IDE steps over one instruction.

2. To see the results. select **Data > View Trace**.

   The debugger displays an **EOnCE Trace View** window. (See Figure 7.29.)

Figure 7.29 EOnCE Trace View Window


Trace for eonce_demo.eld 1

Src Address	Dst Address	Cycles	Ext Cycles
0x10192 (.text + 0x000001...	0x10190 (_main)	0x0	0x0

Display: 0x00010192 View: Raw data

```
Address   Hex: 0000F992:00010992                               Ascii
00010192  C0 87 33 04 21 30 80 01 74 50 F0 44 38 00 30 18    · · 3 · ! 0 · · t P · D 8 · 0 ·
000101A2  80 00 F8 43 F1 C4 F1 42 F2 C3 F2 41 33 04 27 B0    · · · C · · · B · · · A 3 · ' ·
000101B2  80 01 C0 80 E7 70 9F 71 90 C0 90 C0 90 C0 94 C0    · · · · · p · q · · · · · · · ·
000101C2  96 20 97 20 03 06 39 60 80 00 98 C0 34 F3 9F FF    ·   ·   · · 9 ` · · · · 4 · · ·
000101D2  9E 20 9F 20 96 C0 E7 58 25 18 80 E6 3E 00 39 60    ·   ·   · · · X % · · · > · 9 `
000101E2  80 00 3F 00 39 64 80 00 37 01 39 60 80 00 14 96    · · ? · 9 d · · 7 · 9 ` · · · ·
000101F2  52 97 96 C0 39 18 80 03 C6 FC 24 6B DF 04 94 C0    R · · · 9 · · · · · $ k · · · ·
```

Word Size: 8 Page: Physical

8 Using the Profiler

This chapter explains how to use the CodeWarrior™ profiler. The CodeWarrior profiler is an analysis tool that lets you examine the run-time behavior of your StarCore™ DSP programs. Using the profiler, you can find areas of source code that do not execute or take too long to execute.

Profiling is a tuning process. As a result, you should use the profiler to find out how to make your programs run better—not to find bugs. Once your program is stable, you should profile it to find performance bottlenecks.

This chapter has these sections:

Profiler Types

The CodeWarrior profiler supports three types of profiling:

- On Host
- On Chip
- On Chip Timers

On Host

On-host profiling is the simplest type of profiling to set up. All you must do is use the **Profiler Type** list box of the Profiler target settings pane to specify **On Host**.

During on-host profiling, the profiler stops the core being profiled each time the trace buffer becomes full, transfers the data to the host, and then restarts the core. As a result, on host profiling interferes often with program execution.

On Chip

As its name implies, the executable code that performs on-chip profiling resides on the StarCore chip. During on-chip profiling, an interrupt service routine (ISR) analyses the trace buffer each time the buffer becomes full. Compared to on-host profiling, the on-chip ISR technique interferes less with program execution.

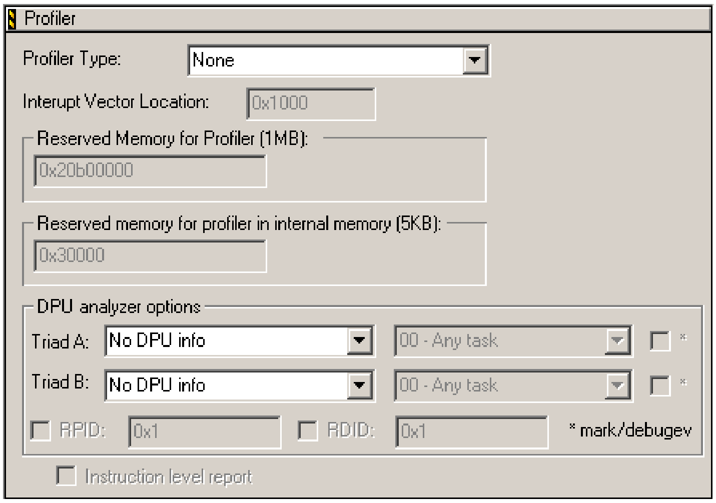
To configure a build target for on-chip profiling, follow these steps:

1. Open the project for which you want to use on-chip profiling.
2. Select **Edit > *TargetName* Settings**, where ***TargetName*** is the name of the current build target.

   The Target Settings window appears.
3. In the **Target Settings Panels** list on the left side of the window, select **Profiler**.

   The Profiler panel (Figure 8.1) appears on the right side of the window.

Figure 8.1 Profiler Target Settings Panel


Profiler

Profiler Type: None

Interupt Vector Location: 0x1000

Reserved Memory for Profiler (1MB): 0x20b00000

Reserved memory for profiler in internal memory (5KB): 0x30000

DPU analyzer options

Triad A: No DPU info | 00 - Any task | ☐ *

Triad B: No DPU info | 00 - Any task | ☐ *

☐ RPID: 0x1 ☐ RDID: 0x1 * mark/debugev

☐ Instruction level report

4. Use the Profiler Type list box to specify **On Chip**.
5. In the Interrupt Vector Location text box, enter the address (in hexadecimal) of the vector to the interrupt service routine that the on-chip profiler uses.
6. In the Reserved Memory for Profiler text box, enter the base address of the 1-megabyte external-memory buffer that the on-chip profiler requires.
7. In your linker command file, include a `.reserve` directive for the external-memory buffer defined above (so that the linker does not use this space).

   For example `.reserve 0x20b00000, 0x20c0000` reserves the area between `0x20b00000` and `0x20c0000` for the on-chip profiler.
8. In the Reserved memory for profiler in internal memory text box, enter the base address of the 5-kilobyte internal memory buffer that the on chip-profiler requires.
9. Again, in your linker command file, include a `.reserve` directive for the internal-memory buffer defined above (so that the linker does not use this space).
10. In your linker command file, define the entry point (`.entry`) as the address of the `___crt0_start` function.

    To determine this address, follow these steps:

    a. Turn off the profiler.
    b. Start debugging.
    c. Halt the debugger at your program’s entry point.
    d. Switch the debugger window to assembler view.

       The entry point address is the address to which the `jmp` instruction points.

11. Add an assembly-language file to your project that declares the area `VBA - VBA + 0x100` as code.

    For example:

    ```
    org p:$0
        dup $100
        nop
        endm
    ```

12. Allocate a buffer for the on-chip profiler's interrupt service routine (ISR).

    a. In your source file, add code that declares a buffer for the on-chip profiler's ISR.

       - Assembly-language source file—add the code hown in Listing 8.1
       - C-language source file—add the code shown in Listing 8.2

Listing 8.1 Assembly Language Declaration of a Buffer for the On Chip Profiler's ISR

```
org p:$C00
    dup $40
    nop
    endm
```

Listing 8.2 C Language Declaration of a Buffer for the On Chip Profiler's ISR

```
#pragma pgm_seg_name ".intC00"
void C00()
{
  asm("nop");
  asm("nop");
  asm("nop");
  asm("nop");
  asm("nop");
  asm("nop");
  asm("nop");
  asm("nop");
  asm("nop");
  asm("nop");
  asm("nop");
  asm("nop");
  asm("nop");
  asm("nop");
  asm("nop");
  asm("nop");
  asm("nop");
  asm("nop");
  asm("nop");
```

```
    asm("nop");
    asm("nop");
    asm("nop");
    asm("nop");
    asm("nop");
    asm("nop");
    asm("nop");
    asm("nop");
    asm("nop");
    asm("nop");
    asm("nop");
    asm("nop");
    asm("nop");
    asm("nop");
    asm("nop");
}
```

b. Add the source file (C or assembly language) to your build target.

c. In your linker command file add this directive for each code section that does *not* contain debug information:

```
.org 0xC00
.segment .p1, ".intC00"
```

13. If the vector base address (VBA) is not zero, add one more section for `0 - 0x100`:

```
org p:$0
    dup $100
    nop
    endm
```

14. Optionally, set profiler points where desired.

NOTE With on-chip profiling, a pair of Profiler Points must reside within the same function.

On Chip Timers

Like on chip profiling, on chip timers profiling takes place on the StarCore chip. The difference is that on chip timers profiling gathers just cycle count information; it does not perform function analysis. As a result, the timing information generated by on chip timers profiling is highly accurate.

To set up a build target for on chip timers profiling, select On Chip Timers from the Profiler Type listbox of the Profiler target setting panel. Then follow the same procedure as for On Chip timers.

NOTE With on chip *timer* profiling, you must set at least one pair of Profiler Points. However, these points do not have to be within the same function (as they must with on chip profiling).

Profiler Points

The on chip and on chip timers profiler types let you restrict profiling to just those parts of your program of interest. The on host profiler type does not support have this capability.

To restrict profiling to specific areas, you set pairs of profiler points—one pair for each part of your program that you want to profile. The first profiler point in a pair causes the profiler to start gathering data; the second point in a pair halts data gathering.

To set a pair of profiler points, follow these steps:

1. Open the project that you want to profile.
2. Using the Profiler target settings panel, configure a build target of this project for the type of profiling desired: host, on chip, or on chip timers.

NOTE See Profiler Types for instructions.

3. Select **Project > Make**

   The compiler and assembler translate your source files and then the linker generates an executable file.

4. Select **Project > Debug**

   The CodeWarrior debugger copies your program to the target device or simulator, halts execution at the program’s entry point, and displays the debugger window and the sessions window.

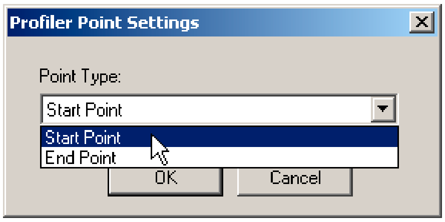
5. Set a pair of profiler points:

   To do this, follow these steps:

   a. Open the source file in which you want to set a pair of profiler points in an editor window.
   b. Put the caret in the source-code statement at which you want to start profiling.
   c. Select **Debug > Set Event Point > Set Profiler Point**

      The debugger displays the **Profiler Point Settings** dialog box. (See Figure 8.2.)

Figure 8.2 Profiler Point Settings Dialog Box

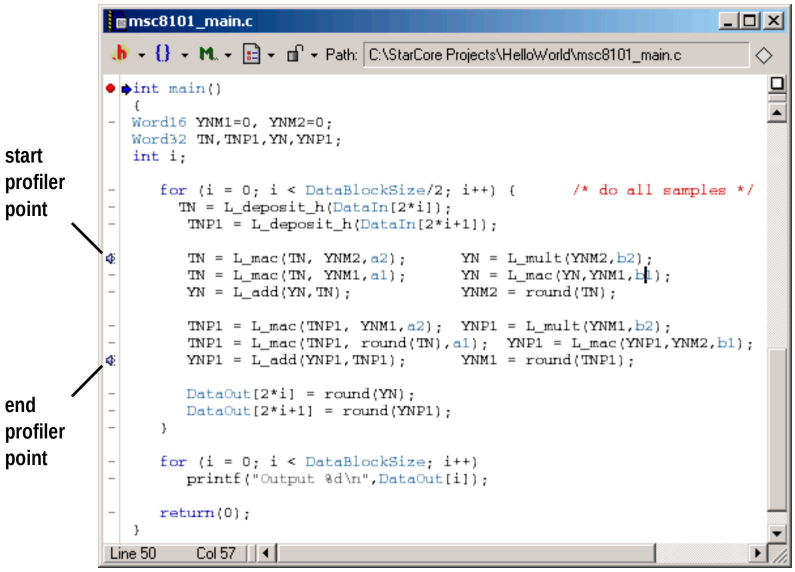


d. From the Point Type listbox, select Start Point.

e. Click **OK**

The debugger assigns a start profiler point to the selected source-code statement. (See Figure 8.3.)

Figure 8.3 Editor Window Showing Start and End Profiler Points



```
int main()
{
Word16 YNM1=0, YNM2=0;
Word32 TN,TNP1,YN,YNP1;
int i;

   for (i = 0; i < DataBlockSize/2; i++) {        /* do all samples */
      TN = L_deposit_h(DataIn[2*i]);
       TNP1 = L_deposit_h(DataIn[2*i+1]);

       TN = L_mac(TN, YNM2,a2);       YN = L_mult(YNM2,b2);
       TN = L_mac(TN, YNM1,a1);       YN = L_mac(YN,YNM1,b1);
       YN = L_add(YN,TN);             YNM2 = round(TN);

       TNP1 = L_mac(TNP1, YNM1,a2);  YNP1 = L_mult(YNM1,b2);
       TNP1 = L_mac(TNP1, round(TN),a1);  YNP1 = L_mac(YNP1,YNM2,b1);
       YNP1 = L_add(YNP1,TNP1);       YNM1 = round(TNP1);

       DataOut[2*i]  = round(YN);
       DataOut[2*i+1] = round(YNP1);
   }

   for (i = 0; i < DataBlockSize; i++)
       printf("Output %d\n",DataOut[i]);

   return(0);
}
```

f. Put the caret in the source-code statement at which you want to *end* profiling.

g. Select **Debug** > **Set Event Point** > **Set Profiler Point**

The debugger displays the **Profiler Point Settings** dialog box.

h. From the Point Type listbox, select End Point.

i. Click **OK**

The debugger assigns an end profiler point to the selected source-code statement. (See Figure 8.3.)

That’s it. Now, when you continue program execution, the profiler starts gathering data when it hits the start profiler point and stops when it hits the stop profiler point.

Profiler Examples

This directory contains example projects that you can build and use to test the various profiler types and features:

`installDir\(CodeWarrior_Examples)\StarCore_Examples\Profiler`

Launching the Profiler

Before you can use the profiler on your program, you must generate a version that includes symbolic debug information.

The CodeWarrior IDE includes debug information in each file for which the project window displays a dot in the debug column of the project. Click in the project window’s debug column next to each file that should include debug information.

If you are compiling from the command line, use the `-g` option to include debug information.

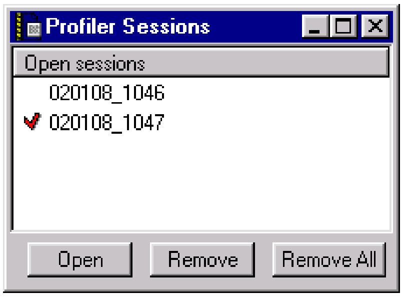
NOTE If your program makes `printf` calls to `stdout`, the output appears in the IDE’s I/O window. This can be useful for marking the progress of your profiling execution.

Once you have built a version of your program that includes debug information, you can launch the profiler. To do this, follow these steps:

1. Open the project to be profiled.
2. Select the type of profiler to use from the Profiler Type listbox of the Profiler target settings panel.
3. Choose **Project > Debug**

   The IDE displays the **Profiler Sessions** window. This window displays a list of open profiler sessions. (See Figure 8.4.)

Figure 8.4 Profiler Sessions Window

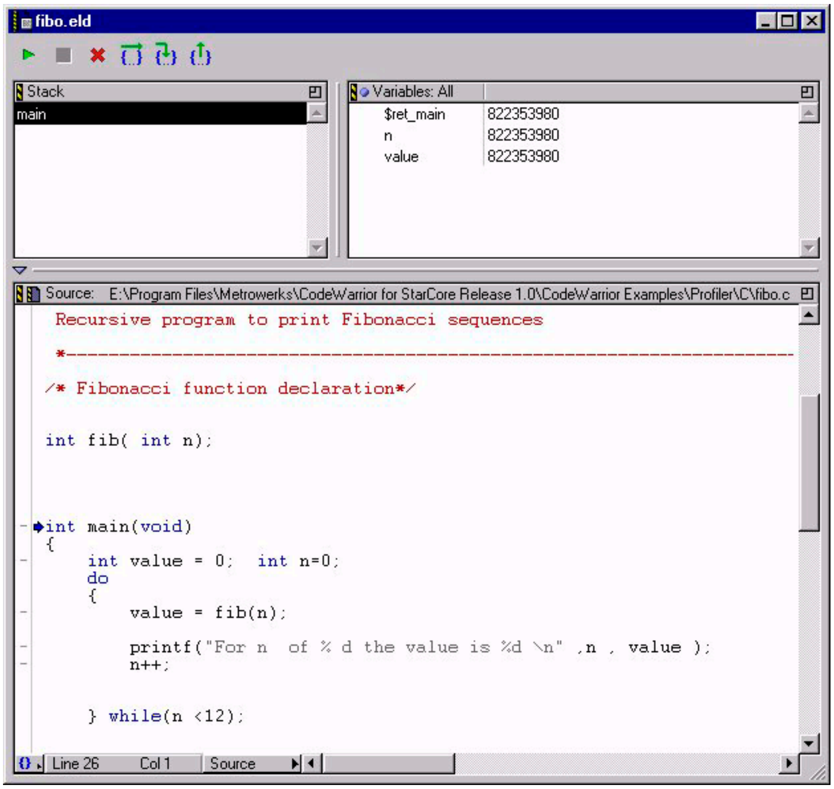


The check mark in the **Profiler Sessions** window identifies the active session.

The IDE begins executing your program and displays the debugger window. (See Figure 8.5.)

NOTE You can use any debugging commands once the debugger window appears.

Figure 8.5 Debugger Window—Upon Starting the Profiler


fibo.eld
Stack
main
Variables: All
Source: E:\Program Files\Metrowerks\CodeWarrior for StarCore Release 1.0\CodeWarrior Examples\Profiler\C\fibo.c
Line 26
Col 1
Source

```
$ret_main    822353980
n            822353980
value        822353980
```

```
  Recursive program to print Fibonacci sequences

  *---------------------------------------------------------------------

 /* Fibonacci function declaration*/

 int fib( int n);


 int main(void)
 {
     int value = 0;  int n=0;
     do
     {
         value = fib(n);

         printf("For n  of % d the value is %d \n" ,n , value );
         n++;

     } while(n <12);
```

NOTE If you debug a multi-core project, the project that specifies the location of the other projects that are part of the multi-core project (in the Other Executables target settings panel) is the master project.

While you debug a multi-core project, downloading the master project might cause the other projects in the multi-core project to download as well. In this case, if you use the profiler to download the master project, the profiler profiles all the projects in the multi-core project. Each of those projects has a separate listing in the **Profiler Sessions** window.

Opening the Profiler Sessions Window

To display the **Profiler Sessions** window, choose **Profiler** > **Sessions**.

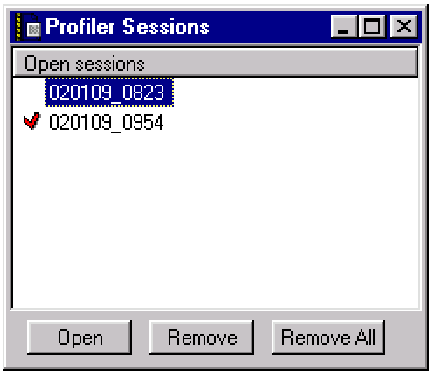
Removing a Profiler Session

To remove a profile from the **Profiler Sessions** window, perform these steps:

1. Click the name of any profiler session to highlight it.

   Figure 8.4 shows a **Profiler Sessions** window after highlighting a session.

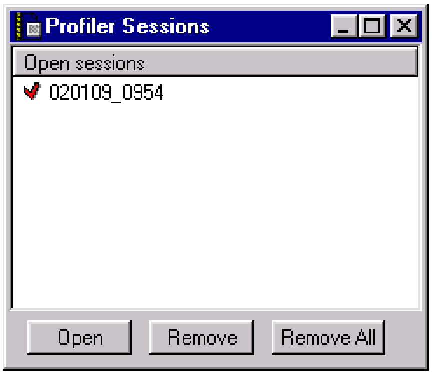
Figure 8.6 Profiler Sessions Window with Session Name Highlighted



2. Click **Remove**

   The IDE removes the session from the **Profiler Sessions** window and closes all other profiler windows related to that session. Figure 8.7 shows the **Profiler Sessions** window after you delete the chosen session.

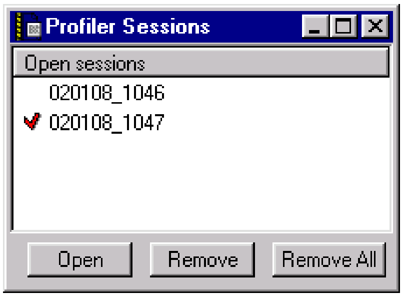
Figure 8.7 Profiler Sessions Window After Removing a Session



Removing All Profiler Sessions

To remove all profiler sessions from a **Profiler Sessions** window, click **Remove All**. (See Figure 8.8.)

Figure 8.8 Profiler Sessions Window



The IDE removes all sessions from the **Profiler Sessions** window and closes all other profiler windows related to these sessions.

View a List of Functions

To view a list of functions, perform any of these actions:

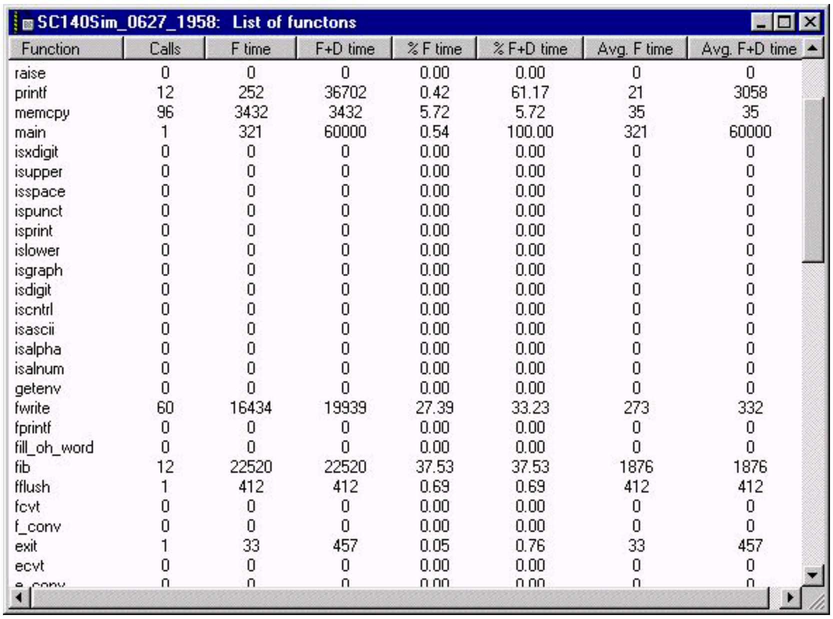
- Choose **Profiler > Functions**

NOTE The IDE applies the Functions command to the currently selected profile in the **Profiler Sessions** window (indicated by a check mark next to the name of the session).

- In the **Profiler Sessions** window, select a profiler session and click **Open**.
- In the **Profiler Sessions** window, double-click the name of a profiler session.

The CodeWarrior IDE displays the **List of Functions** window. (See Figure 8.17.)

Figure 8.9 The List of Functions Window



SC140Sim_0627_1958: List of functons

Function	Calls	F time	F+D time	% F time	% F+D time	Avg. F time	Avg. F+D time
raise	0	0	0	0.00	0.00	0	0
printf	12	252	36702	0.42	61.17	21	3058
memcpy	96	3432	3432	5.72	5.72	35	35
main	1	321	60000	0.54	100.00	321	60000
isxdigit	0	0	0	0.00	0.00	0	0
isupper	0	0	0	0.00	0.00	0	0
isspace	0	0	0	0.00	0.00	0	0
ispunct	0	0	0	0.00	0.00	0	0
isprint	0	0	0	0.00	0.00	0	0
islower	0	0	0	0.00	0.00	0	0
isgraph	0	0	0	0.00	0.00	0	0
isdigit	0	0	0	0.00	0.00	0	0
iscntrl	0	0	0	0.00	0.00	0	0
isascii	0	0	0	0.00	0.00	0	0
isalpha	0	0	0	0.00	0.00	0	0
isalnum	0	0	0	0.00	0.00	0	0
getenv	0	0	0	0.00	0.00	0	0
fwrite	60	16434	19939	27.39	33.23	273	332
fprintf	0	0	0	0.00	0.00	0	0
fill_oh_word	0	0	0	0.00	0.00	0	0
fib	12	22520	22520	37.53	37.53	1876	1876
fflush	1	412	412	0.69	0.69	412	412
fcvt	0	0	0	0.00	0.00	0	0
f_conv	0	0	0	0.00	0.00	0	0
exit	1	33	457	0.05	0.76	33	457
ecvt	0	0	0	0.00	0.00	0	0
e_conv	0	0	0	0.00	0.00	0	0

View an Instruction-Level Report

To generate and view an instruction-level report, follow these steps:

1. Open the project to be profiled.
2. Press **Alt-F7**

   The **Target Settings** window appears.
3. In the left pane of the Target Settings window, click the Profiler item.

   The Profiler target settings panel appears on the right of the **Target Settings** window.
4. In the Profiler panel, select the type of profiler to use from the Profiler Type listbox.
5. In the Profiler panel, check the Instruction level report box.
6. Click **OK**

   The IDE saves your settings and closes the **Target Settings** window.
7. Choose **Project** > **Debug**

   The debugger downloads your build target to the target device and then displays the debugger window and the **Profiler Sessions** window.

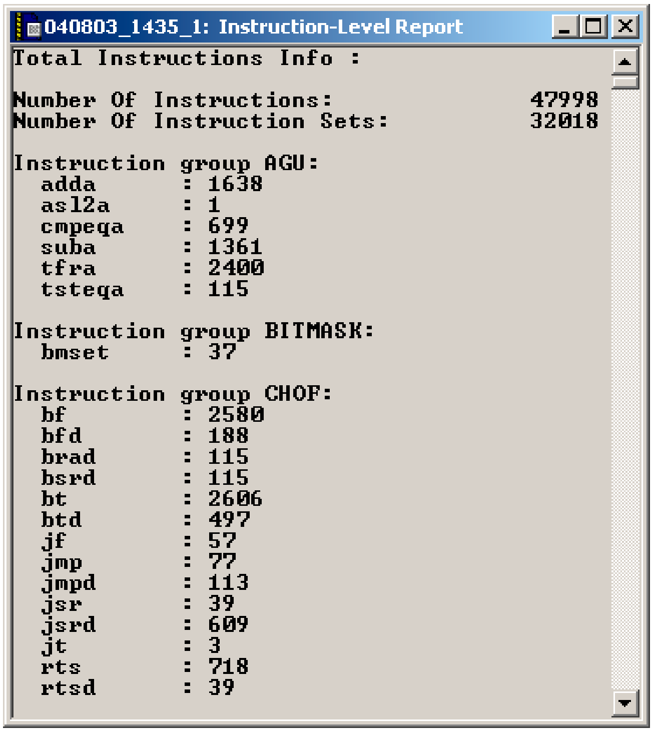
8. In the **Profiler Sessions** window, select the profiler session to use and click **Open**
9. Use the debugger window to execute your program under control of the debugger.

   The profiler gathers information as your program executes.
10. At any point during execution, select **Profiler** > **Instructions**

    The **Instruction-Level Report** window appears. (See Figure 8.10.)

NOTE The Instructions item of the Profile menu is disabled unless you check the Instruction level report checkbox of the Profiler target settings panel.

Figure 8.10 Instruction-Level Report Window



```
Total Instructions Info :

Number Of Instructions:                    47998
Number Of Instruction Sets:                32018

Instruction group AGU:
  adda       : 1638
  asl2a      : 1
  cmpeqa     : 699
  suba       : 1361
  tfra       : 2400
  tsteqa     : 115

Instruction group BITMASK:
  bmset      : 37

Instruction group CHOF:
  bf         : 2580
  bfd        : 188
  brad       : 115
  bsrd       : 115
  bt         : 2606
  btd        : 497
  jf         : 57
  jmp        : 77
  jmpd       : 113
  jsr        : 39
  jsrd       : 609
  jt         : 3
  rts        : 718
  rtsd       : 39
```

The **Instruction-Level Report** window contains these types of information:

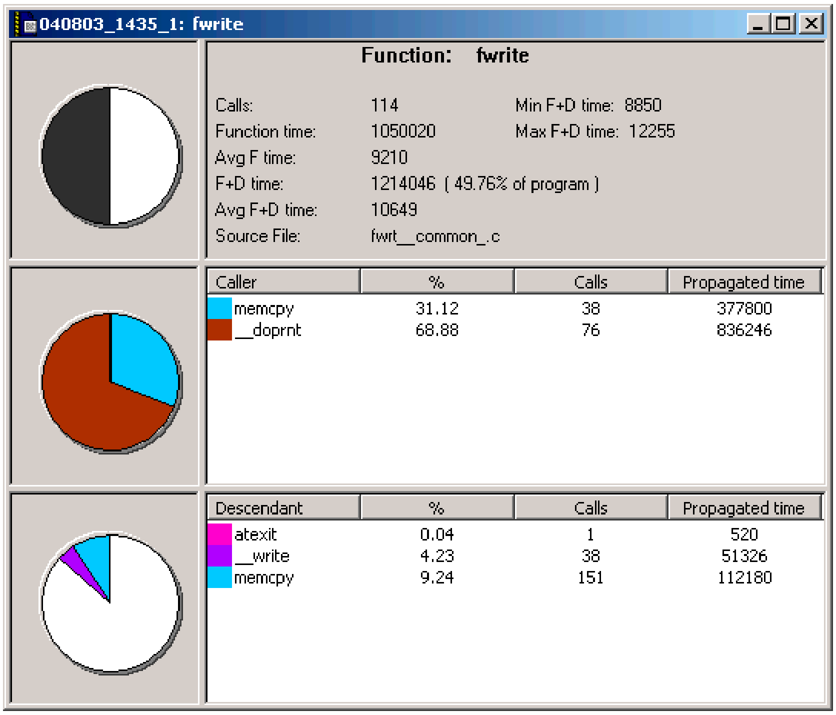
- The number of instructions executed
- The number of instruction sets executed
- A list of executed instructions grouped by category

View Function Details

To view detailed information about a function, double-click a function in a **List of Functions** window or a **Function Call Tree** window.

The **Function Details** window appears. (See Figure 8.11.)

Figure 8.11 The Function Details Window



040803_1435_1: fwrite

Function: fwrite

Calls:	114	Min F+D time:	8850
Function time:	1050020	Max F+D time:	12255
Avg F time:	9210		
F+D time:	1214046 (49.76% of program)		
Avg F+D time:	10649		
Source File:	fwrt__common_.c		

Caller	%	Calls	Propagated time
memcpy	31.12	38	377800
__doprnt	68.88	76	836246

Descendant	%	Calls	Propagated time
atexit	0.04	1	520
__write	4.23	38	51326
memcpy	9.24	151	112180

The **Function Details** window displays data (in graphical and tabular formats) about:

- A particular function
- The immediate callers of that function
- Descendants of that function

For the selected function, the **Function Details** window displays detailed performance information. The pie chart represents the percentage of total execution time used by the function and its descendants.

For caller functions, the **Function Details** window displays this information:

- A list of immediate callers

- The number of times each caller performed a call to the selected function
- Propagated time for callers: the amount of time each caller contributed to the function + descendants (F+D) time of the selected function
- The percentage of time spent in the selected function and its descendants on behalf of the caller
- A pie chart that displays the percentage of time used by each caller

For descendant functions, the **Function Details** window displays the following information:

- A list of immediate descendants
- The number of times the selected function called each descendant function
- Propagated time for descendants: the amount of time each descendant contributed to the function + descendants (F+D) time of the selected function
- The percentage of time each descendant contributed to the total F+D time
- A pie chart that displays the percentage of time used by each descendant

NOTE To open a new **Function Details** window for a descendant or caller, double-click any line in the Caller or Descendant tables.

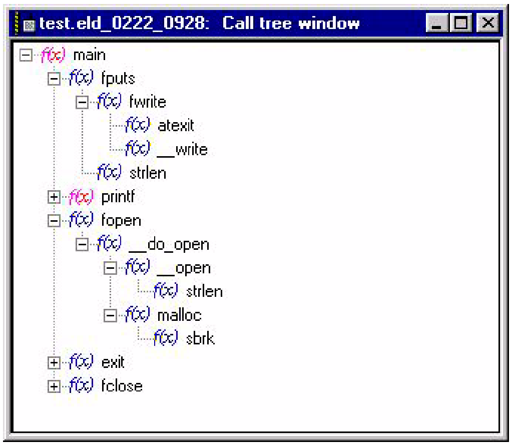
View a Function Call Tree

To view a function call tree, choose **Profiler > Function Call Tree**.

NOTE The IDE applies the Function Call Tree command to the currently selected profile in the **Profiler Sessions** window.

The **Function Call Tree** window appears. (See Figure 8.12.)

Figure 8.12 The Function Call Tree Window



The **Function Call Tree** window shows the dynamic call structure of a program. This window also highlights the path from the most expensive function. (Red entries indicate the more expensive paths.)

NOTE You can open a **Function Details** window for a function by double-clicking the function name in the **Function Call Tree** window.

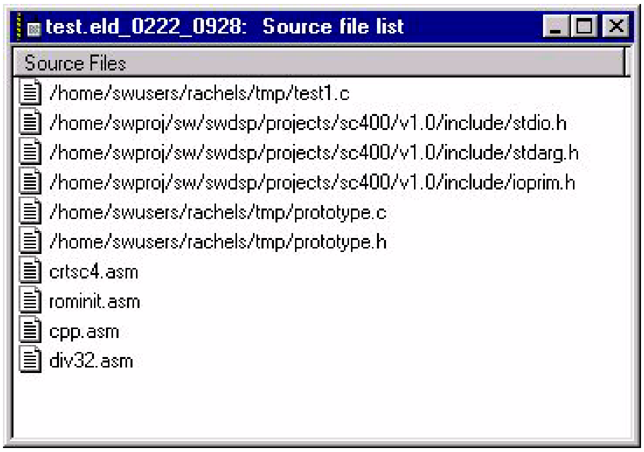
View Source Files Information

To display source files information for the current profile session, choose **Profiler > Source Files**.

NOTE The IDE applies the Source Files command to the currently selected profile in the **Profiler Sessions** window (indicated by a check mark next to the name of the session).

The **Source Files** window appears. (See Figure 8.13.)

Figure 8.13 The Source Files Window


test.eld_0222_0928: Source file list

Source Files
/home/swusers/rachels/tmp/test1.c
/home/swproj/sw/swdsp/projects/sc400/v1.0/include/stdio.h
/home/swproj/sw/swdsp/projects/sc400/v1.0/include/stdarg.h
/home/swproj/sw/swdsp/projects/sc400/v1.0/include/ioprim.h
/home/swusers/rachels/tmp/prototype.c
/home/swusers/rachels/tmp/prototype.h
crtsc4.asm
rominit.asm
cpp.asm
div32.asm

The **Source Files** window displays directory and file information for all files included in the current profile session.

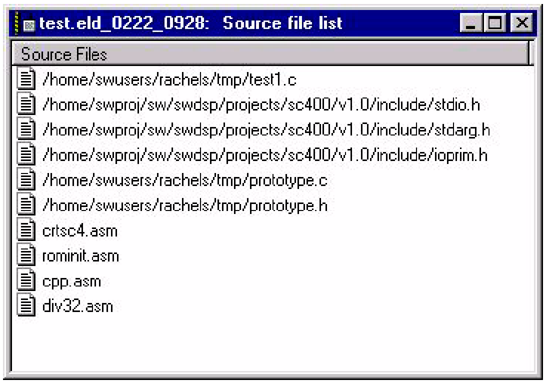
NOTE You can view profile information line-by-line by double-clicking on a source file in the **Source Files** window.

View Profile Information Line-by-Line

To view profile information line-by-line, double-click the name of a source file listed in the **Source Files** window.

Figure 8.14 shows the **Source Files** window.

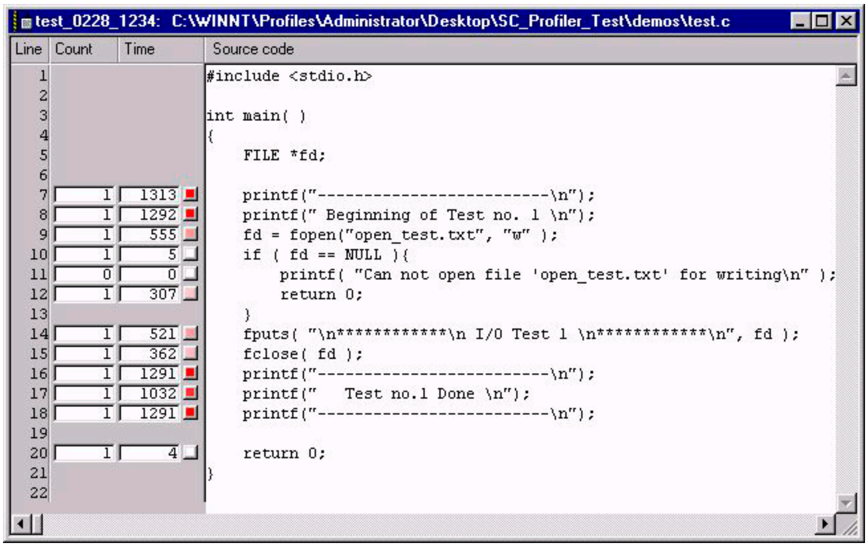
Figure 8.14 The Source Files Window


test.eld_0222_0928: Source file list

Source Files
- /home/swusers/rachels/tmp/test1.c
- /home/swproj/sw/swdsp/projects/sc400/v1.0/include/stdio.h
- /home/swproj/sw/swdsp/projects/sc400/v1.0/include/stdarg.h
- /home/swproj/sw/swdsp/projects/sc400/v1.0/include/ioprim.h
- /home/swusers/rachels/tmp/prototype.c
- /home/swusers/rachels/tmp/prototype.h
- crtsc4.asm
- rominit.asm
- cpp.asm
- div32.asm

A **Profile Line-by-Line** window appears. (See Figure 8.15.)

Figure 8.15 Profile Line-by-Line Window


test_0228_1234: C:\WINNT\Profiles\Administrator\Desktop\SC_Profiler_Test\demos\test.c

Line	Count	Time		Source code
1				`#include <stdio.h>`
2				
3				`int main( )`
4				`{`
5				`    FILE *fd;`
6				
7	1	1313	🟥	`    printf("------------------------\n");`
8	1	1292	🟥	`    printf(" Beginning of Test no. 1 \n");`
9	1	555	⬜	`    fd = fopen("open_test.txt", "w" );`
10	1	5	⬜	`    if ( fd == NULL ){`
11	0	0	⬜	`        printf( "Can not open file 'open_test.txt' for writing\n" );`
12	1	307	⬜	`        return 0;`
13				`    }`
14	1	521	⬜	`    fputs( "\n************\n I/O Test 1 \n************\n", fd );`
15	1	362	⬜	`    fclose( fd );`
16	1	1291	🟥	`    printf("------------------------\n");`
17	1	1032	🟥	`    printf("   Test no.1 Done \n");`
18	1	1291	🟥	`    printf("------------------------\n");`
19				
20	1	4	⬜	`    return 0;`
21				`}`
22				

For each source-code statement, this window displays the number of calls made (in the **Count** column) and the time in instruction cycles (in the **Time** column).

This window also uses colored marks to indicate the most heavily used lines in the Time column. The marks range in color from white to red; the most time-consuming lines use red marks.

NOTE Line-by-line profile data is not available for assembly language source files.

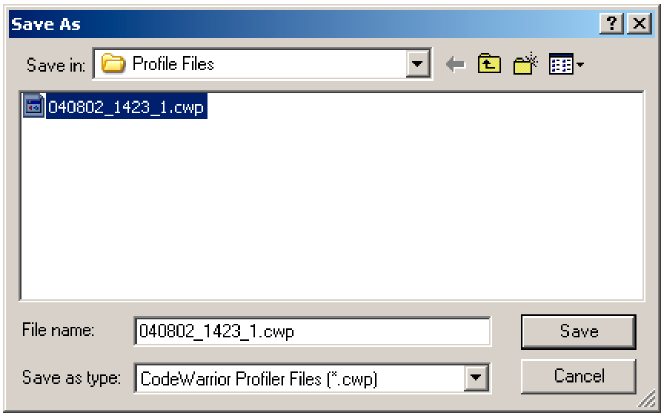
Save a Profile

To save a profile, follow these steps:

1. Choose **Profiler** > **Save**

   The **Save As** dialog box appears. (See Figure 8.16.)

Figure 8.16 Save As Dialog Box



2. Use the dialog box to save your file in the directory of your choice.

Load a Profile

To load a previously saved profile, follow these steps:

1. Choose **Profiler** > **Load**

   A standard dialog box appears.

2. If needed, use the dialog box to navigate to the directory that contains the profile.
3. Select the profile and click **Open**

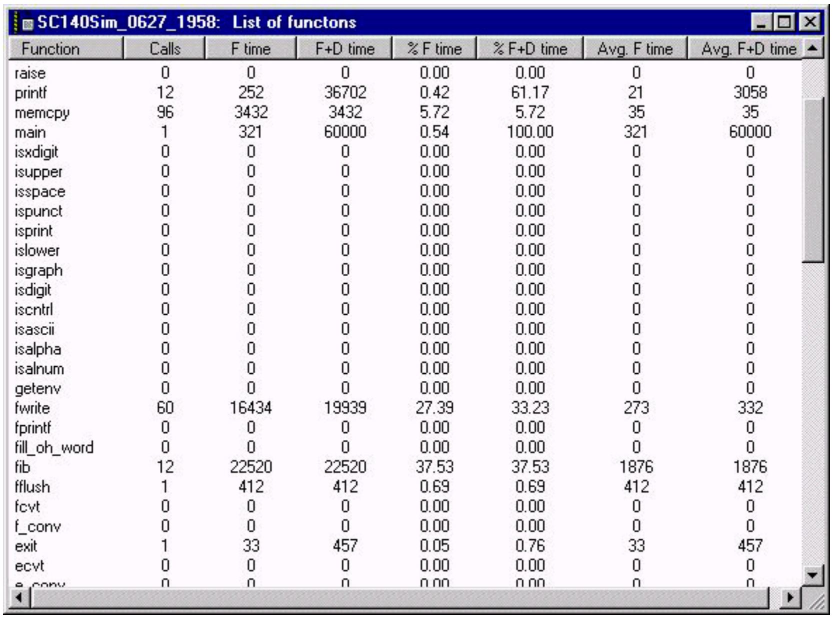
   The **Profiler Sessions** window appears. The window contains the name of the profiler session that you specified.

4. Click on the session in the **Profiler Sessions** window.

5. Click **Open**

   The CodeWarrior IDE displays a **List of Functions** window containing the information from your previously saved profile. (See Figure 8.17.)

Figure 8.17 The List of Functions Window


SC140Sim_0627_1958: List of functons

Function	Calls	F time	F+D time	% F time	% F+D time	Avg. F time	Avg. F+D time
raise	0	0	0	0.00	0.00	0	0
printf	12	252	36702	0.42	61.17	21	3058
memcpy	96	3432	3432	5.72	5.72	35	35
main	1	321	60000	0.54	100.00	321	60000
isxdigit	0	0	0	0.00	0.00	0	0
isupper	0	0	0	0.00	0.00	0	0
isspace	0	0	0	0.00	0.00	0	0
ispunct	0	0	0	0.00	0.00	0	0
isprint	0	0	0	0.00	0.00	0	0
islower	0	0	0	0.00	0.00	0	0
isgraph	0	0	0	0.00	0.00	0	0
isdigit	0	0	0	0.00	0.00	0	0
iscntrl	0	0	0	0.00	0.00	0	0
isascii	0	0	0	0.00	0.00	0	0
isalpha	0	0	0	0.00	0.00	0	0
isalnum	0	0	0	0.00	0.00	0	0
getenv	0	0	0	0.00	0.00	0	0
fwrite	60	16434	19939	27.39	33.23	273	332
fprintf	0	0	0	0.00	0.00	0	0
fill_oh_word	0	0	0	0.00	0.00	0	0
fib	12	22520	22520	37.53	37.53	1876	1876
fflush	1	412	412	0.69	0.69	412	412
fcvt	0	0	0	0.00	0.00	0	0
f_conv	0	0	0	0.00	0.00	0	0
exit	1	33	457	0.05	0.76	33	457
ecvt	0	0	0	0.00	0.00	0	0
e_conv	0	0	0	0.00	0.00	0	0

Generate a Tab-Delimited Profile Report

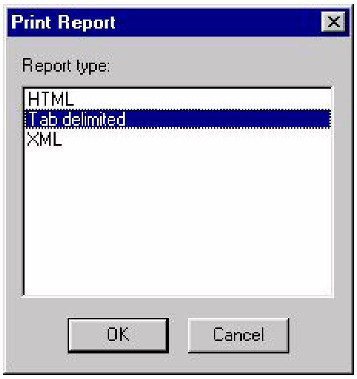
To generate a tab-delimited profile report, follow these steps:

1. Choose **Profiler > Export**

   The **Print Report** window appears.

2. Select Tab delimited in the **Print Report** window. (See Figure 8.18.)

Figure 8.18 Print Report Window with Tab delimited Selected



3. Click **OK**
4. Choose a location to save the file.

   This tells the profiler where to place the tab-delimited output file (named `report.td`). The profiler overwrites the file if it already exists.

You can open a tab-delimited report in a text editor or spreadsheet program. The report contains profiling information for functions as well as source-code profiling by line.

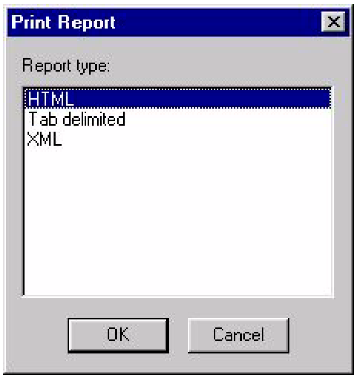
Generate an HTML Profile Report

To generate a profile report in HTML, follow these steps:

1. Choose **Profiler > Export**

   The **Print Report** window appears.
2. Select HTML in the **Print Report** window. (See Figure 8.19.)

Figure 8.19 Print Report Window with HTML Selected

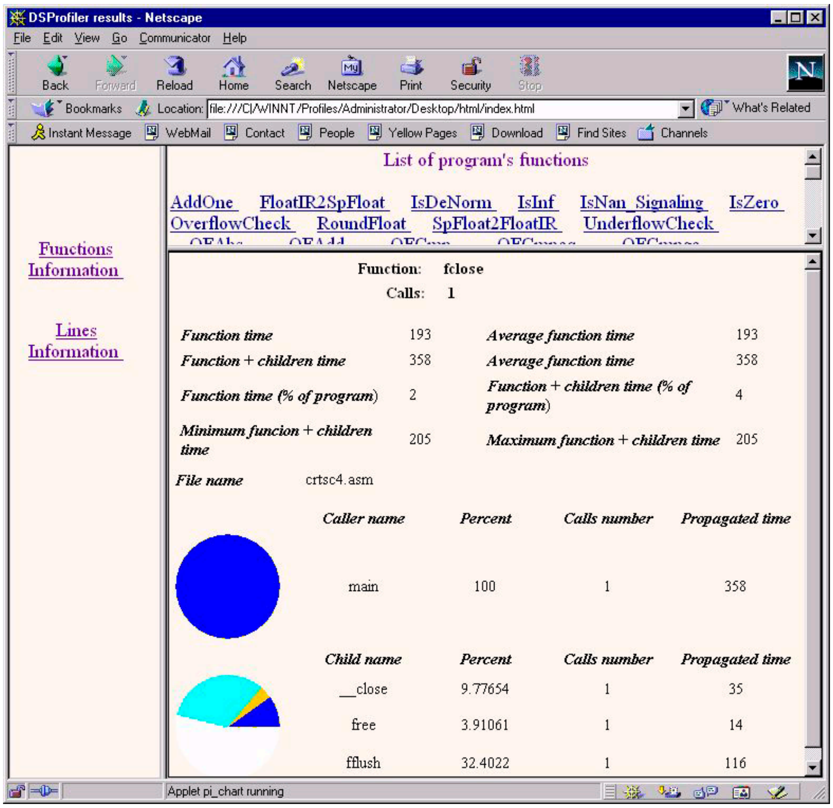


3. Click **OK**
4. Choose a location to save the file.

   This tells the profiler where to place the HTML output files and the java class file needed to draw charts. The profiler overwrites the HTML output files and java class file if they already exist.
5. To view the report, open `index.html` in the directory in which you saved the report.

   Figure 8.20 shows an example HTML report.

Figure 8.20 Viewing an HTML Profiling Report in a Web Browser


DSProfiler results - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: file:///C|/WINNT/Profiles/Administrator/Desktop/html/index.html
What's Related
Instant Message WebMail Contact People Yellow Pages Download Find Sites Channels
Functions Information
Lines Information
Applet pi_chart running

List of program's functions

AddOne FloatIR2SpFloat IsDeNorm IsInf IsNan_Signaling IsZero
OverflowCheck RoundFloat SpFloat2FloatIR UnderflowCheck

Function: fclose
Calls: 1

Function time	193	*Average function time*	193
Function + children time	358	*Average function time*	358
Function time (% of program)	2	*Function + children time (% of program)*	4
Minimum funcion + children time	205	*Maximum function + children time*	205

File name crtsc4.asm

Caller name	*Percent*	*Calls number*	*Propagated time*
main	100	1	358

Child name	*Percent*	*Calls number*	*Propagated time*
__close	9.77654	1	35
free	3.91061	1	14
fflush	32.4022	1	116

Generate an XML Profiling Report

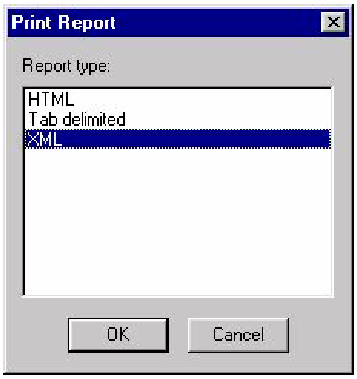
To generate a profiling report in XML:

1. Choose **Profiler > Export**

   The **Print Report** window appears.

2. Select XML in the **Print Report** window. (See Figure 8.21.)

Figure 8.21 Print Report Window with XML Selected



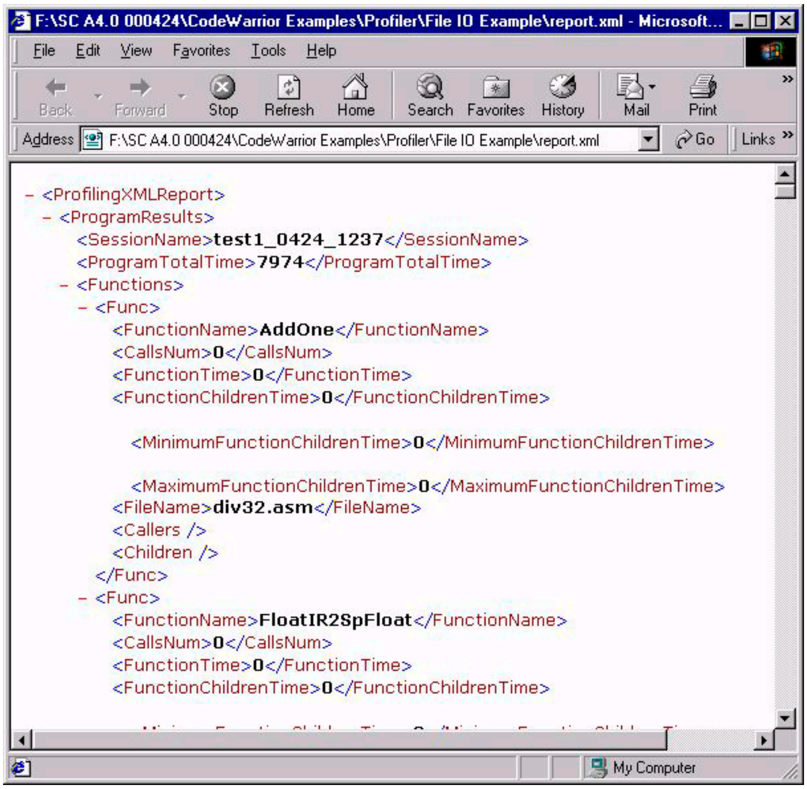
3. Click **OK**
4. Choose a location to save the file.

   This tells the profiler where to place the report (named `report.xml`). The report contains the XML output for the function call tree with number of calls made, function time, and time for child functions.

NOTE To view `report.xml`, you must use Internet Explorer version 5.0 or greater.

Figure 8.22 shows an example XML report.

Figure 8.22 Viewing an XML Profiling Report in a Web Browser


F:\SC A4.0 000424\CodeWarrior Examples\Profiler\File IO Example\report.xml - Microsoft...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print
Address F:\SC A4.0 000424\CodeWarrior Examples\Profiler\File IO Example\report.xml
Go Links
My Computer

```
- <ProfilingXMLReport>
  - <ProgramResults>
      <SessionName>test1_0424_1237</SessionName>
      <ProgramTotalTime>7974</ProgramTotalTime>
    - <Functions>
      - <Func>
          <FunctionName>AddOne</FunctionName>
          <CallsNum>0</CallsNum>
          <FunctionTime>0</FunctionTime>
          <FunctionChildrenTime>0</FunctionChildrenTime>

            <MinimumFunctionChildrenTime>0</MinimumFunctionChildrenTime>

            <MaximumFunctionChildrenTime>0</MaximumFunctionChildrenTime>
          <FileName>div32.asm</FileName>
          <Callers />
          <Children />
        </Func>
      - <Func>
          <FunctionName>FloatIR2SpFloat</FunctionName>
          <CallsNum>0</CallsNum>
          <FunctionTime>0</FunctionTime>
          <FunctionChildrenTime>0</FunctionChildrenTime>
```

Set Up to Profile Assembly Language Programs

To get profiling results from an assembly language program:

1. Use the following syntax for assembly language functions:

   ```
   global func_name
   func_name type func
     function_source_code
   func_name_end
   ```

2. Follow these rules:
   - Call or jump to the subroutine by only one change-of-flow instruction.

- Provide only one entry point for the function you want to profile (the first subroutine instruction).
- Return from the subroutine by only one change-of-flow instruction.

Set Up the VTB to Use an On-host Profiler

To set up the Virtual Trace Buffer (VTB) to use an on-host profiler, you must modify some default settings. This section explains how to make the necessary modifications, based on the type of mode you use.

Trace Mode

In order to use on-host profiling, use the **Trace Mode** list box of the VTB Configuration panel to specify **EonCE No Compression**. This setting ensures that the trace buffer contains source addresses, destination addresses, and cycle information. The profiler can process this information correctly.

These trace modes are *not* compatible with the profiler, because they cause the trace buffer to contain information that the profiler cannot process:

- EonCE with Compression
- EonCE Counter
- Sample Bit Counter
- Trace Buffer Data Register
- EDCA5 Event Counter

Write Mode

Each **Write Mode** that you can specify in the VTB Configuration panel has some important considerations. The following sections explain those considerations.

Over Write

In Over Write mode, the debugger continues execution even after the VTB becomes full. The debugger overwrites the data in the VTB, starting at the **VTB Start Address**. Use this mode during long debugging session, when you want to see the content of the VTB after program execution reaches a breakpoint.

To have more accurate information for profiling purposes, we recommend that you:

- change the **VTB Start Address** and **VTB End Address** values to create a larger buffer size

- step into each function so that you minimize the buffer load between two debug events (when the profiler processes the VTB)

One Address

This mode is not compatible with the profiler, unless you write the trace information to a peripheral device. The Debugging and Profiling Unit (DPU) always writes to the same **VTB Start Address**.

Trace Event Request

The DPU generates a debug request to the core each time it tries to write to the **Trace Event Request Address**. This address must be at least 0.25 kilobytes before the **VTB End Address**. This spacing is necessary because the content might exceed the ending address due to the information written there after entering debug mode.

In Trace Event Request mode, the debugger stops execution every time the VTB becomes full. To have your program stop execution less often, change the **VTB Start Address** and **VTB End Address** values to create a larger buffer size.

On-chip Profiler Panel Settings

To configure the Profiler panel for on-chip profiling, follow these steps:

1. Check the **Enable Virtual Trace Buffer** checkbox of the VTB Configuration panel.
2. Use the information explained in Trace Mode and Write Mode to configure the VTB for on-host profiling.

NOTE VTB configurations for on-host and on-chip profiling are very similar. For on-chip profiling, make sure that the range between the **VTB Start Address** and **VTB End Address** is not excessive.

3. Enter `0x20000` in the **Reserved Memory for Profiler** text box of the Profiler panel.
4. Enter `0x11000` in the **Reserved memory for profiler in internal memory** text box of the Profiler panel.

9 Debugger Communication Protocols

The CodeWarrior™ debugger can communicate with StarCore™ and SDMA devices in several ways.

Table 9.1 lists each target device along with the debugger-supported communication protocols for that device. The protocols are the CodeWarrior Connection Server (CCS) and the ARM® RealView® In-Circuit Emulator (RVI).

Table 9.1 Debugger Communication Protocols

Debugger Target Device	CCS	RVI
i.300-30 (physical hardware)		✓
i.300-30 (Palladium™ emulator)		✓
MXC 05 (physical hardware)		✓
MXC 05 (Palladium emulator)		✓
MXC 275-30 (physical hardware)		✓
MXC 275-30 (Palladium emulator)		✓
MXC 300-10 (physical hardware)		✓

Table 9.1 Debugger Communication Protocols (*continued*)

Debugger Target Device	CCS	RVI
MXC 300-30 (physical hardware)		✓
Platform 2002 Simulator	✓	

This chapter explains these communications protocols.

- CodeWarrior Connection Server
- ARM® RealView® ICE Connection

CodeWarrior Connection Server

The CodeWarrior Connection Server (CCS) provides a TCP/IP connection point for debugger communications. If you run a CCS instance on your computer, remote instances of the CodeWarrior debugger can access each target board connected to your system. Similarly, each instance of the CodeWarrior debugger can access the target boards connected to each remote computer that runs a CCS instance.

The P2002 simulator target uses the CCS.

Default CCS Remote Connection

Before you can use CCS to debug programs, you must define a remote connection for CCS in the CodeWarrior IDE. The CodeWarrior installer creates a default CCS remote connection. You can use this default remote connection as-is, modify it, or create new remote connections from scratch.

The **Remote Connections** panel of the **IDE Preferences** window (Figure 9.1) lists the default remote connections. Of these connections, the **SC100 Simulator** connection is the default CCS remote connection. This connection defines parameters for the CCS to communicate with a single-core StarCore simulator.

Creating a CCS Remote Connection

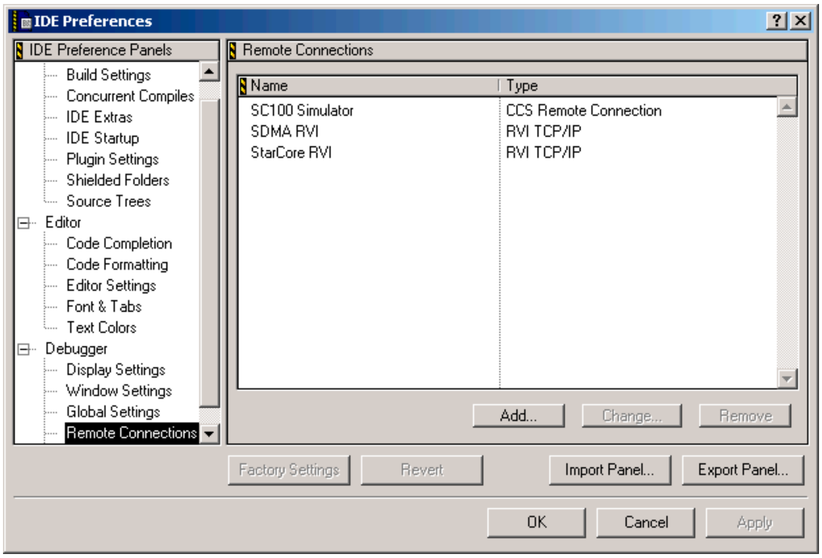
If you do not want to change the default CCS remote connection, you can create your own remote connection from scratch.

To create a new remote connection, follow these steps:

1. Select **Edit** > **Preferences**.

   The **IDE Preferences** window (Figure 9.1) appears.

Figure 9.1 IDE Preferences window



Remote Connections

Name	Type
SC100 Simulator	CCS Remote Connection
SDMA RVI	RVI TCP/IP
StarCore RVI	RVI TCP/IP

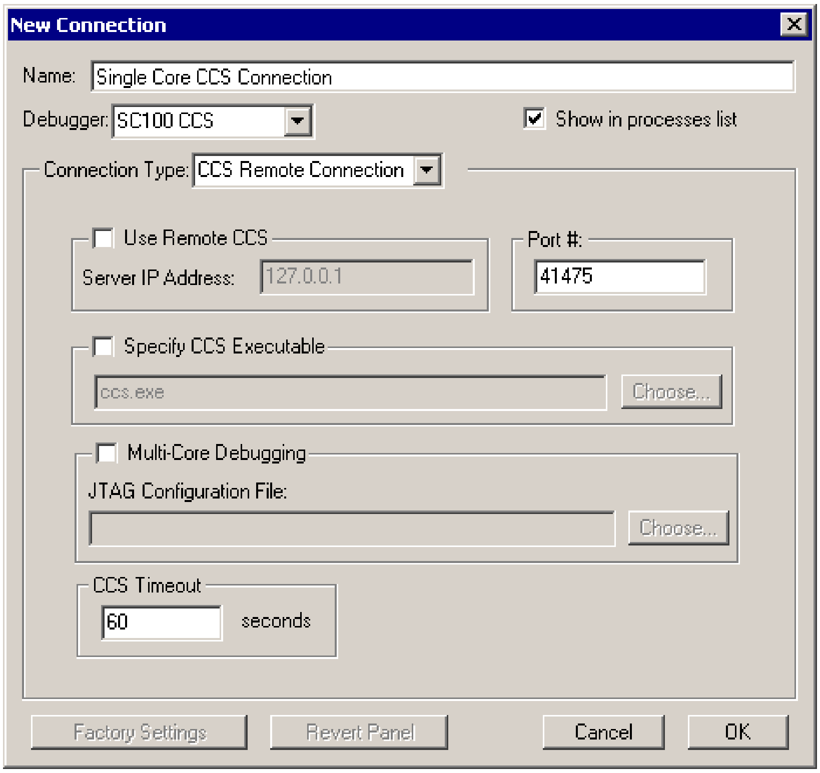
2. Select **Remote Connections** from the **IDE Preference Panels** list on the left side of the window.

   The Remote Connections preference panel appears.

3. Click the **Add** button.

   The **New Connection** dialog box appears.

Figure 9.2 New Connection Dialog Box


New Connection

Name: Single Core CCS Connection

Debugger: SC100 CCS

[✓] Show in processes list

Connection Type: CCS Remote Connection

[ ] Use Remote CCS

Server IP Address: 127.0.0.1

Port #: 41475

[ ] Specify CCS Executable

ccs.exe

[ ] Multi-Core Debugging

JTAG Configuration File:

CCS Timeout

60 seconds

4. In the **Name** text box, enter the name of the new connection.
5. Use the **Debugger** list box to specify **SC100 CCS**.
6. Use the **Connection Type** list box to specify **CCS Remote Connection**.
7. In the **Port #** text box, enter the port number on which CCS listens for messages.

NOTE The CCS protocol uses the default port 41475.

8. If you are connecting to a CCS instance on a different computer, check the **Use Remote CCS** box and specify the **Server IP Address** of the remote computer.
9. If you are using CCS with a multi-core target, check the **Multi-Core Debugging** checkbox, then specify the path to the appropriate **JTAG Configuration File**.
10. Click the **OK** button.

    The **New Connection** dialog box closes, and the remote connection you just created appears in the **Remote Connections** list.

Configuring a Project to Use a CCS Remote Connection

After you create a CCS remote connection, you must configure your project to use that connection.

NOTE If you have not created a CCS remote connection yet, follow the instructions in “Creating a CCS Remote Connection” on page 290.

To use a CCS remote connection with your project, follow these steps:

1. Bring forward the project window of the project that you want to configure to use the remote connection.
2. Select **Edit > *Targetname* Settings**, where ***Targetname*** is the name of the active build target in the project.

   The **Target Settings** window appears.
3. Select **Remote Debugging** from the **Target Settings Panels** list on the left side of the window.

   The Remote Debugging panel appears in the window.
4. Use the **Connection** list box to specify the name of the CCS remote connection.

   For example, specify the name of the remote connection that you created in “Creating a CCS Remote Connection” on page 290.
5. Click the **OK** button.

   The IDE saves your changes and closes the Target Settings window. During each subsequent debugging session, your project will use the remote connection you specified.

Running CCS

Each time you debug a project that uses a local CCS connection, the CodeWarrior IDE automatically starts CCS if it is not running already. Also, you can run CCS manually from this location, where *CodeWarrior* is the path to your CodeWarrior installation:

CodeWarrior`\ccs\bin\ccs.exe`

If CCS is running, this icon appears in the Windows taskbar:

Right-click this icon to display the CCS context menu. The menu has these commands:

- **Show Console** —Displays the CCS console
- **Hide Console**—Hides the CCS console
- **About CCS**—Displays version information

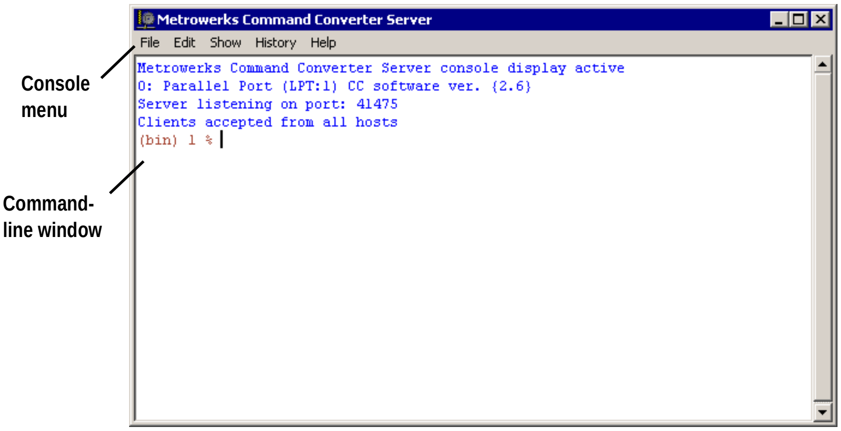
- **Quit CCS** —Terminates CCS

The CCS Console

The CCS console lets you view and change the server connection options. You can issue commands by typing them into the command-line window or by selecting options from the console menu.

Figure 9.3 shows the CCS console.

Figure 9.3 The CCS Console



```
Metrowerks Command Converter Server console display active
0: Parallel Port (LPT:1) CC software ver. {2.6}
Server listening on port: 41475
Clients accepted from all hosts
(bin) 1 %
```

Configuring the CCS

In its default configuration, CCS uses parallel port 1 to communicate with a target device and port 41475 to listen for commands. To change these default settings, follow these steps:

1. Run *CodeWarrior*`\ccs\bin\ccs.exe`, where *CodeWarrior* is the path to your CodeWarrior installation.

   CCS starts and displays this icon in the Windows taskbar:
2. Right-click the CCS icon.

   A context menu appears.
3. Select **Show console** from this menu.

   The CCS console appears.
4. At the console command prompt, issue these commands

   a. `delete all`

      This command deletes the current CCS configuration.

   b. `config cc` *`connection_type`*

      where *`connection_type`* can be:

      - `parallel:`<#>
      - `lpt` (same as `parallel`)
      - `epp:`<#>
      - `pci`
      - `hti:`<*ip_address*>
      - `bdm:`<#>
      - `usb`
      - `powertap:`<*ip_address*>
      - `wiretap:`<#>
      - `lspusb`

      This command defines the command converter that CCS uses.

   c. `config port` *port_number*

      where *port_number* is the port on which CCS listens for commands. The default CCS port number is 41475.

5. Right-click the CCS icon.

   The CCS context menu appears.

6. Select **Quit CCS** from this menu.

   CCS exits.

NOTE Any changes you make to the CCS configuration persist from one CCS session to the next.

ARM® RealView® ICE Connection

The CodeWarrior IDE provides a TCP/IP connection point for debugger communications with the ARM Ltd. RealView In-Circuit Emulator (RVI). Most of the StarCore and SDMA devices support this type of connection.

Default RVI Remote Connections

Before you can use RVI to debug programs, you must define a remote connection for RVI in the CodeWarrior IDE. The CodeWarrior installer creates default RVI remote connections. You can use these default remote connections as-is, modify them, or create new remote connections from scratch.

The **Remote Connections** panel of the **IDE Preferences** window (Figure 9.4) lists the default remote connections. Of these connections, the **SDMA RVI** and **StarCore RVI** connections are the default RVI remote connections.

Creating an RVI Remote Connection

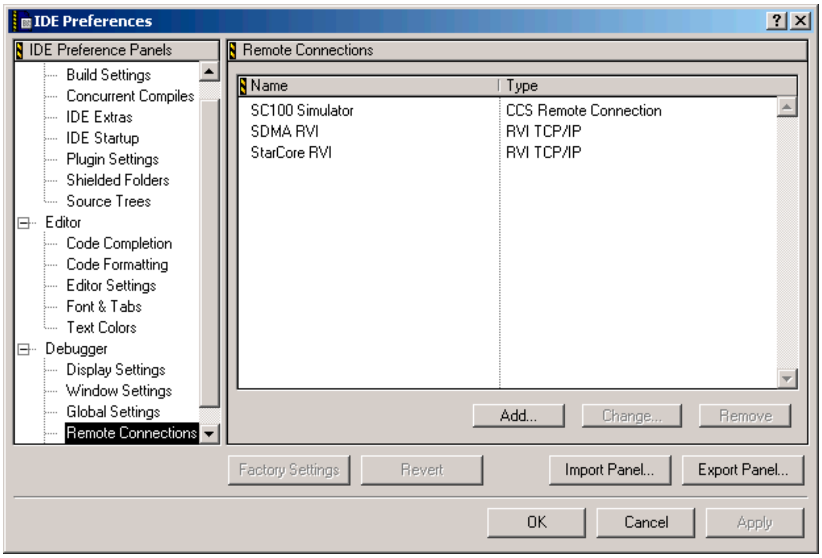
If you do not want to change the default RVI remote connections, you can create your own remote connection from scratch.

To create a new remote connection, follow these steps:

1. Select **Edit > Preferences**.

   The **IDE Preferences** window (Figure 9.4) appears.

Figure 9.4 IDE Preferences window



Name	Type
SC100 Simulator	CCS Remote Connection
SDMA RVI	RVI TCP/IP
StarCore RVI	RVI TCP/IP

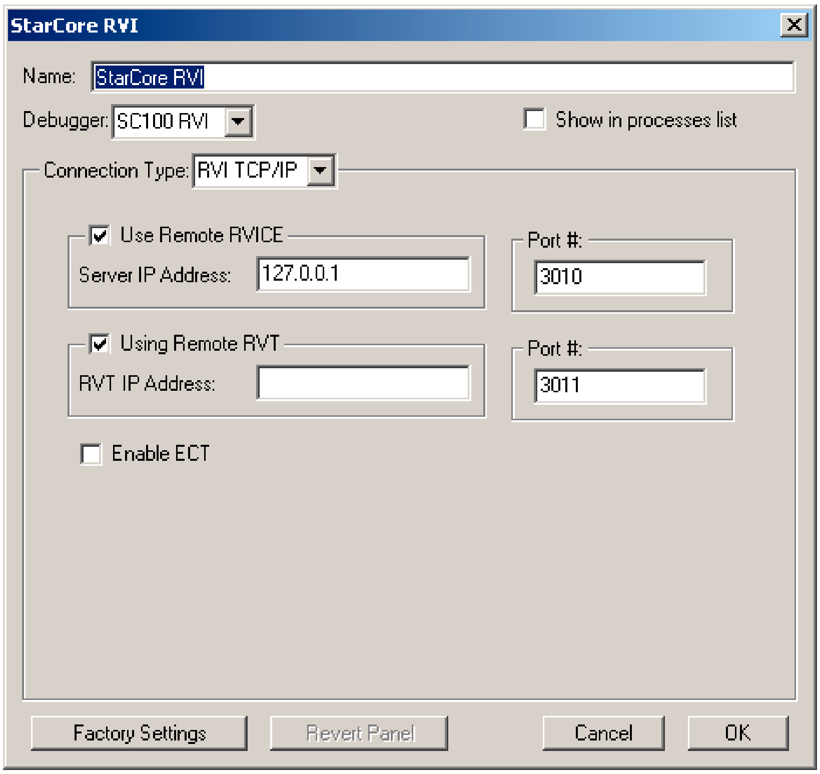
2. Select **Remote Connections** from the **IDE Preference Panels** list on the left side of the window.

   The Remote Connections preference panel appears.

3. Click the **Add** button.

   The **New Connection** dialog box appears.

Figure 9.5 New Connection Dialog Box


StarCore RVI

Name: StarCore RVI

Debugger: SC100 RVI

☐ Show in processes list

Connection Type: RVI TCP/IP

☑ Use Remote RVICE

Server IP Address: 127.0.0.1

Port #: 3010

☑ Using Remote RVT

RVT IP Address:

Port #: 3011

☐ Enable ECT

4. In the **Name** text box, enter the name of the new connection.
5. Use the **Debugger** list box to specify **SC100 RVI**.
6. Use the **Connection Type** list box to specify **RVI TCP/IP**.
7. Check the **Use Remote RVICE** checkbox, then specify the **Server IP Address** and **Port #** through which the RVI communicates with the target hardware.

NOTE The RVI protocol uses the fixed port 3010.

8. If you are connecting to ARM RealView Nexus Trace hardware, check the **Use Remote RVT** box and specify the **RVT IP Address** and **Port #** of the hardware.

9. If you are using the Embedded Cross Trigger as part of the remote connection, check the **Enable ECT** checkbox.
10. Click the **OK** button.

    The **New Connection** dialog box closes, and the remote connection you just created appears in the **Remote Connections** list.

Configuring a Project to Use an RVI Remote Connection

After you create an RVI remote connection, you must configure your project to use that connection.

NOTE If you have not created an RVI remote connection yet, follow the instructions in "Creating an RVI Remote Connection" on page 296.

To use an RVI remote connection with your project, follow these steps:

1. Bring forward the project window of the project that you want to configure to use the remote connection.
2. Select **Edit > *Targetname* Settings**, where ***Targetname*** is the name of the active build target in the project.

   The **Target Settings** window appears.
3. Select **Remote Debugging** from the **Target Settings Panels** list on the left side of the window.

   The Remote Debugging panel appears in the window.
4. Use the **Connection** list box to specify the name of the RVI remote connection.

   For example, specify the name of the remote connection that you created in "Creating an RVI Remote Connection" on page 296.
5. Click the **OK** button.

   The IDE saves your changes and closes the Target Settings window. During each subsequent debugging session, your project will use the remote connection you specified.

10

StarCore™ DSP Utilities

This chapter explains how to use the StarCore DSP utility programs included with CodeWarrior™ Development Studio for StarCore and SDMA. Refer to the CodeWarrior *Build Tools Reference for SDMA* for information about SDMA utility programs.

This chapter has these sections:

Flash Programmer

The CodeWarrior flash programmer lets you manipulate the flash memory of a StarCore DSP from within the CodeWarrior IDE.

NOTE This release of CodeWarrior Development Studio for StarCore and SDMA does *not* support the flash-programmer tool.

SC100 ELF Dump Utility

The SC100 ELF file dump utility outputs the headers of absolute and linkable object files in a human-readable form.

The utility-generated output depends on the ELF object-file type:

- Absolute (executable) object file

  The default output is the ELF header, all program headers, and all section headers.

- Linkable (relocatable) object file

  The default output is the ELF header and all section headers.

NOTE The CodeWarrior *Build Tools Reference for SDMA* has information about the SDMA ELF Dump utility.

Running the SC100 ELF Dump Utility

You can run the ELF dump utility from within the CodeWarrior IDE by selecting **SC100 ELF Dump** from the **Post-linker** list box of the **Target Settings** panel.

You can also invoke the ELF dump utility from a command-line prompt by using this syntax:

```
sc100-elfdump [option ...] file ...
```

The ELF dump utility is in this directory, where *CodeWarrior* is the path to your CodeWarrior installation:

CodeWarrior`\StarCore_Support\compiler\bin`

Parameters

`option`

One or more of the command-line options listed in Table 10.1. Without options, the utility returns the contents of the ELF Ehdr, Phdr, and Shdr structures and the symbol table. If you specify command-line options, the utility returns only the information that you specify on the command line.

NOTE `sc100-elfdump` utility command-line options are case-sensitive.

`file`

One or more filenames (including optional pathnames). Each input file must be an ELF object file (either absolute or relocatable).

Table 10.1 SC100 ELF File Dump Utility—Command-Line Options

Option	Effect
`-A`	Writes the contents of all program segments.
`-D`	Writes the contents of all PT_DYNAMIC segments. (Does not apply to the SC140 DSP core.)
`-E`	Writes ELF header information. (Default)

Table 10.1 SC100 ELF File Dump Utility—Command-Line Options (*continued*)

Option	Effect
`-I`	Writes the contents of all PT_INTERP segments. (Does not apply to the SC140 DSP core.)
`-L`	Writes the contents of all PT_LOAD segments.
`-N`	Writes the contents of all PT_NOTE segments. (Does not apply to the SC140 DSP core.)
`-P`	Writes the contents of all PT_PHDR segments.
`-R` *file*	Writes the output to the specified file, instead of to the standard output.
`-S`	Writes the contents of all PT_SHLIB segments. (Does not apply to the SC140 DSP core.)
`-U`	Writes the contents of all unknown-type segments as hex dumps.
`-V`	Displays the version of the ELF file dump utility.
`-X`	Dumps all program-segment contents as hex.
`-a`	Writes the contents of all sections.
`-b`	Writes the contents of all SHT_PROGBITS sections.
`-c`	Writes the elf file without date or other information so that a valid checksum on the elf file can be obtained.
`-d`	Writes the contents of all SHT_DYNAMIC sections. (Does not apply to the SC140 DSP core.)
`-e` *file*	Writes error messages to the specified file instead of to stderr.
`-g`	Writes the contents of all debug sections (in hexadecimal).
`-h`	Writes the contents of all SHT_HASH sections. (Does not apply to the SC140 DSP core.)
`-i`	Interprets section contents.
`-n`	Writes the contents of all SHT_NOTE sections.
`-o`	Writes the contents of overlay table sections.

Table 10.1 SC100 ELF File Dump Utility—Command-Line Options (*continued*)

Option	Effect
`-q`	Specifies quiet mode: limits header information to the specified sections and segments.
`-r`	Writes the contents of all SHT_REL and SHT_RELA sections.
`-s`	Writes the contents of all SHT_SHLIB sections. (Does not apply to the SC140 DSP core.)
`-t`	Writes the contents of all SHT_STRTAB sections.
`-u`	Writes the contents of all unknown-type sections as hex dumps.
`-x`	Dumps contents of all sections as hex.
`-y`	Writes the contents of all SHT_SYMTAB sections.
`-z`	Writes the contents of all SHT_DYNSYM sections. (Does not apply to the SC140 DSP core.

SC100 ELF File Dump Output

Listing 10.1 shows the output of the ELF file dump utility. Note these things:

- The file name is on the first line: `hello.eld`
- The ELF header extends from line `e_ident` through line `e_shstrndx`, inclusive.
- The program headers extend from `Segment 0` through `Segment 1` and their subordinate lines, inclusive.
- The section headers extend through the remaining lines.

Listing 10.1 SC100 ELF File Dump Utility—Output

```
hello.eld:
   e_ident      : 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
                  (ELF 32-bit LSB Version 1
   e_type       : 2 (Executable file)
   e_machine    : 58 (StarCore 100)
   e_version    : 1
   e_entry      : 0
   e_phoff      : 0x34
   e_shoff      : 0x2fe
   e_flags      : 0x80 (SC140 (V2))
   e_ehsize     : 52
```

```
e_phentsize : 32
e_phnum     : 2
e_shentsize : 40
e_shnum     : 8
e_shstrndx  : 7
Segment 0:
    p_type   : PT_LOAD
    p_offset : 0x100
    p_vaddr  : 0
    p_paddr  : 0
    p_filesz : 88
    p_memsz  : 88
    p_flags  : 0x5 PF_R PF_X
    p_align  : 16
Segment 1:
    p_type   : PT_LOAD
    p_offset : 0x158
    p_vaddr  : 0x58
    p_paddr  : 0x58
    p_filesz : 14
    p_memsz  : 20
    p_flags  : 0x7 PF_R PF_W PF_X
    p_align  : 2

Section 0:
    sh_name      :
    sh_type      : SHT_NULL
    sh_flags     : 0
    sh_addr      : 0
    sh_offset    : 0
    sh_size      : 0
    sh_link      : 0
    sh_info      : 0
    sh_addralign : 0
    sh_entsize   : 0
Section 1:
    sh_name      : .text
    sh_type      : SHT_PROGBITS
    sh_flags     : 0x6 SHF_ALLOC SHF_EXECINSTR
    sh_addr      : 0
    sh_offset    : 0x100
    sh_size      : 88
    sh_link      : 0
    sh_info      : 0
    sh_addralign : 16
    sh_entsize   : 0
              .
              .
```

.

ELF to LOD Utility

Use the ELF to LOD utility to write the information in an ELF file into a specially formatted ASCII file called a loadable module (LOD) file.

Running the ELF to LOD Utility

You can run the ELF to LOD utility from within the CodeWarrior IDE by selecting **SC100 ELF to LOD** from the **Post-linker** list box of the **Target Settings** panel.

Also, you can invoke the ELF to LOD utility from a command-line prompt by using this syntax:

```
elflod [option ...] file ...
```

The ELF to LOD utility is in this directory, where *CodeWarrior* is the path to your CodeWarrior installation:

CodeWarrior`\StarCore_Support\compiler\bin`

Parameters

`option`

Determines the location where the utility generates its output. If you use `option -d` *file*, the utility redirects its output to the specified *file*. Otherwise, the utility generates its output in a default location.

NOTE `elflod` utility command-line options are case-sensitive.

`file`

The filename of the ELF object file to use to produce a LOD file.

ELF To LOD Output

Listing 10.2 shows an example LOD file.

Listing 10.2 Format of a LOD File

```
_START Module_ID Version Rev# Device# Asm_Version Comment
_END Entry_point_address
_DATA Memory_space Address Code_or_Data
_BLOCKDATA Memory_space Address Count Value
```

```
_SYMBOL Memory_space Symbol_Address ...
_COMMENT Comment
```

ELF to S-Record Utility

Use the ELF to S-Record (elfsrec) utility to convert ELF format files to S-Record format files.

The S-Record format, which is a standard file format, encodes programs or data files in a printable form for exchange among computer systems.

Running the ELF to S-Record Utility

You can run the ELF to S-Record utility from within the CodeWarrior IDE by selecting SC100 ELF to S-Record from the Post-linker listbox of the **Target Settings** panel.

Also, you can invoke the ELF to S-Record utility from a command prompt using this command line:

```
elfsrec [option ...] file ...
```

The ELF to S-Record utility is in this directory, where *CodeWarrior* is the path to your CodeWarrior installation:

CodeWarrior`\StarCore_Support\compiler\bin`

Parameters

`option`

One or more of the command-line options listed in Table 10.2.

NOTE `elfsrec` utility command-line options are case-sensitive.

`file`

One or more filenames (including optional pathnames). Each input file must be an ELF object file.

Table 10.2 elfsrec Utility—Command-Line Options

Option	Description
`-b`	Causes elfsrec to create byte-addressable S-Records. By default, elfsrec uses this option. This generates S1 records.
`-w`	Causes elfsrec to create word-addressable S-Records. This generates S2 records.
`-l`	Causes elfsrec to create long-word-addressable S-Records. This generates S3 records.
`-d` [*file_name*]	Causes elfsrec to write the S-Records to the specified file. If you do not specify a file name, the output file has the same name as the input file with a `.s` extension.
-o *value*	Specifies a memory offset (in hexadecimal or decimal). (Hexadecimal numbers must be preceded by `0x`.) The elfsrec utility adds the specified value to the memory address of each line in the S-Record file.

Using StarCore-Specific elfsrec Options

You can use these `elfsrec` options with the CodeWarrior for StarCore and SDMA software:

- `-l`
- `-d`
- `-o`

NOTE The elfsrec utility's default option is `-b`.

SC100 Archiver Utility

The SC100 Archiver groups separate object files into a single file for linking or archival storage. You can add, extract, delete, and replace files in an existing archive.

The `sc100-ar` utility is in this directory, where *CodeWarrior* is the path to your CodeWarrior installation:

CodeWarrior`\StarCore_Support\compiler\bin`

NOTE The CodeWarrior *Build Tools Reference for SDMA* has information about the SDMA Archiver utility.

To invoke the SC100 archiver, use one of the command lines listed below. Table 10.3 defines the purpose and effect of each command-line option.

```
sc100-ar -d [-v] archive file ...
sc100-ar -p [-v] archive [file ...]
sc100-ar -r [-c] [-u] [-v] [-e] [-f] [-s] archive file ...
sc100-ar -t [-v] archive [file ...]
sc100-ar -x [-v] archive [file ...]
sc100-ar -V
sc100-ar @argument file
```

NOTE SC100 Archiver utility command-line options are case-sensitive.

Parameters

`archive`

The name of archive file.

`file`

Name of the file or files to add, extract, replace, or delete from the specified archive file. Separate multiple filenames with spaces. The SC100 archiver processes files in the order listed on the command line.

NOTE You specify an archive file for `file` because an archive file can contain other archive files.

`argument file`

The name of a file that contains SC100 archiver command-line options. These are the syntax rules for an argument file:

- Begin a comment line with the # character.
- End each line with the \ (backslash) character.

Table 10.3 SC100 Archiver Utility—Command-Line Options

Option	Effect
`-c`	Suppresses the default diagnostic message written to standard error upon archive creation. This option is valid only with the `-r` option.
`-d`	Deletes the listed files from the specified archive.
`-e`	Recreates the whole archive. This option is valid only with the -r option.
`-f`	Forces adding an unknown file format to the library. This option is valid only with the `-r` option.
`-p`	Writes the contents of the listed files from the specified archive to the standard output. If the command does not include any filenames, the archiver writes the contents of all files, in their order in the archive.
`-r`	Replaces current files of the specified archive, appends new files to the specified archive, or creates a new archive that contains the listed files.
`-s`	Forces extracting `.elf` files from the specified archive, adding or replacing the `.elf` files instead of the whole archive. This option is valid with just the `-r` option.
`-t`	Writes the archive table of contents, including the specified files, to the standard output. If the command does not include any file names, the table of contents includes all archive files, in their order in the archive.
`-u`	Updates archive files that have been changed since the last update. This option is valid only with the -r option.
`-v`	Produces verbose output: • With the `-d`, `-r`, or `-x` option, produces a file-by-file description of archive creation and maintenance. • With the `-p` option, writes the name of a file to the standard output before writing the file contents to the standard output. • With the `-t` option, includes a long listing of file information within the archive.

Table 10.3 SC100 Archiver Utility—Command-Line Options (*continued*)

Option	Effect
`-V`	Displays the current archiver version, then exits.
`-x`	Extracts the listed files from the specified archive. If the command does not include any file names, the archiver extracts all files of the archive. This option does not change archive contents.

disasmsc100 Disassembler

The `disasmsc100` utility disassembles both SC140 and SC140E DSP binaries. Features of the `disasmsc100` include:

- Interpretation of relocation information
- Data disassembling
- Label (symbol) address output
- Padding awareness (alignment)
- Statistics display

NOTE The `sc100-dis` utility is identical to the `disasmsc100` utility. The CodeWarrior product includes `sc100-dis` for backward compatibility.

The `disasmsc100` utility is in this directory, where *CodeWarrior* is the path to your CodeWarrior installation:

CodeWarrior`\StarCore_Support\compiler\bin`

Running the disasmsc100 Utility

Use this command line to invoke the `disasmsc100` utility:

```
disasmsc100 [-c] [-d] [-f] [-k<file>] [-i{l|b}]
[-l<hex address>>] [-h<hex address>] [-p] [-r] [-q] [-s]
[-x] [-u] [-v] [-z] <srcfile>
```

Table 10.4 lists and defines each option you can pass to the `disasmsc100` utility.

NOTE `disasmsc100` utility command-line options are case-sensitive.

Table 10.4 disasmsc100 Disassembler—Command-Line Options

Option	Effect
`-c`	Specifies compact mode: outputs all execution-set instructions on a single line.
`-d`	Treats an input file as a memory dump (`.lod`) file.
`-b<label>`	Starts disassembling at specified label.
`-e<label>`	Ends disassembling at specified label.
`-l<addr>`	Starts disassembling at specified address (in hexadecimal).
`-h<addr>`	Ends disassembling at specified address (in hexadecimal).
`-f`	Prints `loopstart` — `loopend` directives instead of `lpmarkx*`.
`-i{l|b}`	Specifies interactive mode with little endian (l) or big endian (b).
`-k<file>`	Reads options from the specified configuration file.
`-q`	Suppresses banner display.
`-u`	Ignores relocation information.
`-n`	Displays unmangled form of C++ names.
`-p`	Suppresses the Pc display.
`-r`	Rearranges instructions in packets.
`-s`	Suppresses label and header display.
`-x`	Displays mixed hexadecimal and assembly code.
`-v`	Specifies verbose mode.
`-z`	Displays statistics.

Disasmsc100 Disassembler Output

Listing 10.3 shows the `disasmsc100` utility's output if you pass the `-z` (display statistics) command-line option.

Listing 10.3 disasmsc100 Disassembler—Output Produced by the -z Option

```
;Global EQUs
X        equ      $fffd
```

```
zl      equ     $fffffffe

;Local EQUs
lab1    equ     $fffffffd
lab3    equ     $fffffffd

                section .text2
                sectype progbits
                secflags alloc
                secflags execinstr
;00000000:
_f3     type    func
F__MemAllocArea_18_00000000
F__MemAllocArea_18
                nop
;00000002:
F__MemAllocArea_18_00000002
                nop
;00000004:
F_f3_end
F__MemAllocArea_18_end
                endsec
-------------------------------------------------------
  General Statistics:
    No of instruction:                           2
    No of packets:                               2
    No of 1-word-low-prefixes:                   0
    No of 1-word-high-prefixes:                  0
    No of 2-word-prefixes:                       0
    No of DALU instructions:                     0   0%
    No of AGU instructions:                      0   0%
    No of prefixes and NOP instructions:         2 100%
-------------------------------------------------------
  DALU Statistics:
    No of VLESs with 0 DALU:                     2 100%
    No of VLESs with 1 DALU:                     0   0%
    No of VLESs with 2 DALU:                     0   0%
    No of VLESs with 3 DALU:                     0   0%
    No of VLESs with 4 DALU:                     0   0%
    DALU parallelism:                           0.00
-------------------------------------------------------
  AGU Statistics:
    No of VLESs with 0 AGU:                      2 100%
    No of VLESs with 1 AGU:                      0   0%
    No of VLESs with 2 AGU:                      0   0%
    AGU parallelism:                           0.00
-------------------------------------------------------
  DALU/AGU Usage Details:
          0 DALU | 1 DALU | 2 DALU | 3 DALU | 4 DALU
  0 AGU     100%       0%       0%       0%       0%
  1 AGU       0%       0%       0%       0%       0%
  2 AGU       0%       0%       0%       0%       0%
-------------------------------------------------------
  Used Instuctions:
    nop                                        2
-------------------------------------------------------
```

SC100 Name Utility

The SC100 Name utility displays the symbolic information in each object file and object-file library specified on the command line. If a file contains no symbolics, the utility reports this fact and exits without returning an error.

The `sc100-nm` utility is in this directory, where *CodeWarrior* is the path to your CodeWarrior installation:

CodeWarrior`\StarCore_Support\compiler\bin`

NOTE The CodeWarrior *Build Tools Reference for SDMA* has information about the SDMA Name utility.

Running the SC100 Name Utility

Use this command line to invoke the name utility:

```
sc100-nm [-option ...] file ...
```

Parameters

`option`

One or more of the options listed in Table 10.5.

NOTE SC100 Name utility command-line options are case-sensitive.

`file`

Name of the file to process.

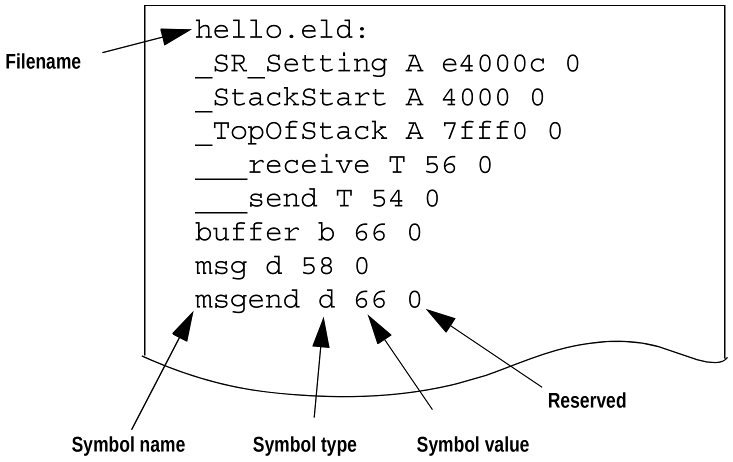
Table 10.5 SC100 Name Utility—Command-Line Options

Option	Effect
`-A`	Writes the full pathname or library name of an object on each line.
-g	Writes only external (global) symbol information. (Do not use this option with the -u option.)
-P	Writes the information in the POSIX.2 portable output format.
-s	Prints the symbol index for archives.
-t {d | o | x}	Writes each numeric value in the specified format: • d — decimal • o — octal • x — hexadecimal (the default)
-u	Writes only undefined symbols. (Do not use this option with the -g option.)
-V	Displays the version of the name utility.
-v	Sorts output by value, instead of by name.

SC100 Name Utility Output

Figure 10.1 shows the output that the SC100 Name utility generates.

Figure 10.1 SC100 Name Utility—Output



```
hello.eld:
_SR_Setting A e4000c 0
_StackStart A 4000 0
_TopOfStack A 7fff0 0
___receive T 56 0
___send T 54 0
buffer b 66 0
msg d 58 0
msgend d 66 0
```

Table 10.6 provides a key to the SC100 name utility's output. Note that uppercase letters indicate global symbols, and lowercase letters indicate local symbols.

Table 10.6 SC100 Name Utility—Output Key

Character	Symbol Type
`U`	Undefined reference
`A` or `a`	Absolute symbol
`B` or `b`	BSS symbol
`T` or `t`	Text (code) symbol
`D` or `d`	Data symbol
`R` or `r`	Read-only data symbol
`N`	Debug symbol
`?`	Unknown symbol type or binding

SC100 Size Utility

The SC100 Size utility outputs the size (in bytes) of each section of each ELF object file specified on the command line. The default output lists totals for all `.text`, `.rodata`, `.data`, and `.bss` sections.

The `sc100-size` utility is in this directory, where *CodeWarrior* is the path to your CodeWarrior installation:

CodeWarrior`\StarCore_Support\compiler\bin`

NOTE The CodeWarrior *Build Tools Reference for SDMA* has information about the SDMA Size utility.

Running the SC100 Size Utility

Use this command line to invoke the size utility:

```
sc100-size [-option ...] file ...
```

Parameters

`option`

One or more of the options listed in Table 10.7.

NOTE `sc100-size` utility command-line options are case-sensitive.

`file`

Name of an ELF file.

Table 10.7 SC100 Size Utility—Command-Line Options

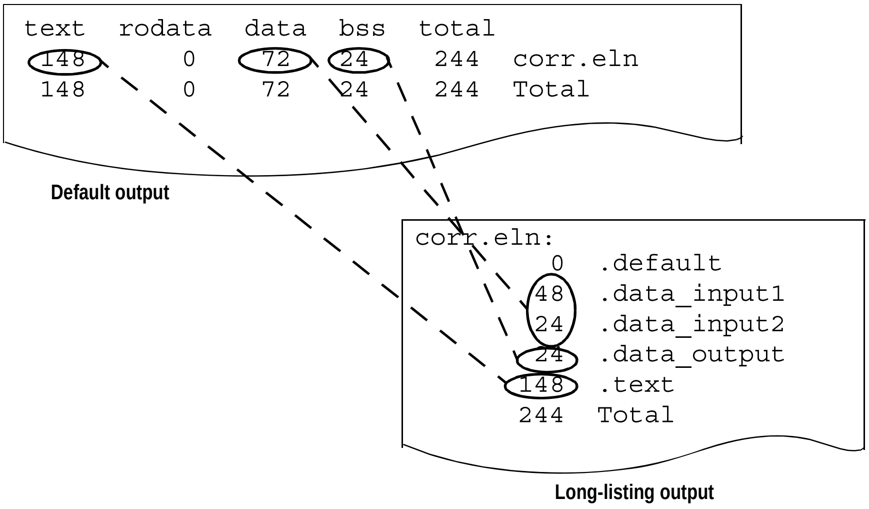
Option	Effect
-l	Specifies long listing mode: outputs names and sizes of individual sections.
-n	Outputs the sizes of individual sections that do not get loaded.
-p	Outputs the size of all loadable segments (program view).
-V	Displays the version of the size utility.

SC100 Size Utility Output

Listing 10.1 shows two examples of size utility output.

- The default output, at the upper left, lists the totals of all text, rodata, data, and bss sections of the object file. It shows 148 text bytes, 72 data bytes, and 24 bss bytes.
- The lower right output example shows the long-listing format for the same object file. It shows that the 72 data bytes are in two files of 48 and 24 bytes.

Figure 10.2 SC100 Size Utility—Output



```
text   rodata  data  bss   total
 148       0    72   24    244   corr.eln
 148       0    72   24    244   Total
```

Default output

```
corr.eln:
          0   .default
         48   .data_input1
         24   .data_input2
         24   .data_output
        148   .text
        244   Total
```

Long-listing output

Statistics Utility

The `sc100-stat` utility is a standalone statistics tool for `.eld` files.

This utility reads a `.eld` file and returns statistics about:

- the number of instructions
- the type of instructions
- the number of instruction sets
- the ratio between the number of instructions and instruction sets

The `sc100-stat` utility is in this directory, where *CodeWarrior* is the path to your CodeWarrior installation:

CodeWarrior`\StarCore_Support\compiler\bin`

The syntax for `sc100-stat` is:

```
sc100-stat .eld_filename [section_name...] [-d]
```

Table 10.8 lists and defines the options for the `sc100-stat` utility.

NOTE `sc100-stat` utility command-line options are case-sensitive.

Table 10.8 sc100-stat Utility—Syntax Diagram Key

Option	Description
.eld_filename	The name of the `.eld` file on which to run sc100-stat.
section_name...	An optional list of section names for sc100-stat to check. If no section names are listed, sc100-stat checks the .text section by default.
`-d`	Causes sc100-stat to print the disassembled code before the statistics.

11

Link Commander

This chapter explains how to use the Link Commander, a graphical utility for creating and editing linker command files.

The sections are:

- User Interface Components
- Creating a Linker Command File

User Interface Components

To start the Link Commander, select **Project > Link Commander** from the IDE's menu. The **Link Commander** window appears. (See Figure 11.1.)

Figure 11.1 Link Commander Window

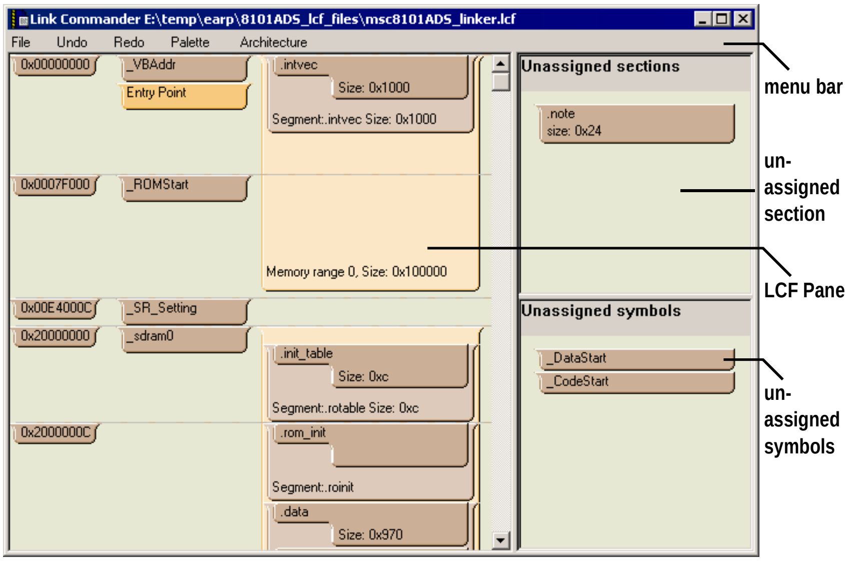


Table 11.1 lists and defines each option in the Link Commander menu bar.

Table 11.1 Link Commander—Menu Selections

Menu	Selection	Effect
File	New (blank file)	Creates a new, empty linker command file.
	New (template file)	Creates a new linker command file according to the command file you select as a template.
	Open	Opens an existing linker command file.
	Save	Saves the current linker command file.
	Save As	Saves the current linker command file under the name and path you specify.
	Save As and Update Project	Saves the current linker command file under the name and path that you specify. Adds this file to the current CodeWarrior project.
Undo	---	Undoes as many as five previous actions.
Redo	---	Redoes as many as five previously undone actions.
Palette	---	Changes the color scheme of the **Link Commander** window.
Architecture	---	Selects an architecture and core, thereby adding memory range guidelines to the LCF pane.

NOTE Selecting **File** > **Save** removes any comments that may have existed in the original linker command file. To preserve such comments, select **File** > **Save As** and specify a different name or path for the new file.

The **Link Commander** window consists of three panes:

- Unassigned Sections

  This pane contains the sections of your project that are not yet mapped to locations within the linker command file.

  – To generate the list of sections within your project, you must first compile your project.

  - Right-click a section name to bring up its placement context menu. Use this menu to select an appropriate place in the LCF.

- Unassigned Symbols

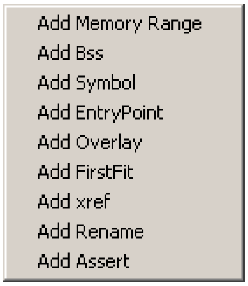
  This pane contains common symbols of your project that have not been assigned any value. To assign a value, right-click a symbol.

- LCF

  This pane depicts the linker command file graphically.

  - Right-click on any existing LCF object to edit its properties.
  - Right-click on a blank portion of the pane to bring up the Add context menu. (See Figure 11.2.)
  - Select from this menu to add a memory range, a symbol, or any of several other LCF objects.

Figure 11.2 Add Context Menu



Creating a Linker Command File

To create an LCF, follow these steps:

1. Assign Memory Addresses to Symbols
   a. Right-click in the LCF Pane.
   b. Select Add Symbol from the context menu.
2. Create Memory Ranges
   a. Right-click in the LCF pane.
   b. Select Add Memory Range from the context menu.
3. Create Segments
   a. Right-click in a memory range inside the LCF pane.
   b. Select Add Segment from the context menu.

4. Assign Sections
   a. Right-click a section in the Unassigned Sections pane.
   b. Select the destination segment for the section.
5. Create an Entry Point
   a. Right-click in the LCF pane.
   b. Select Add Entry Point from the context menu.

12

StarCore C and Assembly Language Benchmarks

Your CodeWarrior™ software includes source code for common DSP benchmarks. Use these benchmarks to:

- Evaluate the performance of the Enterprise C Compiler.
- Evaluate the performance of the StarCore™ DSP architecture.
- Model how to program for the StarCore DSP.

The benchmark package contains these items:

- C Language Benchmarks
- Assembly Language Benchmarks

C Language Benchmarks

The C language benchmark source code files are here:

installDir`\Examples\StarCore_Examples\Benchmark\C\`

Table 12.1 describes the benchmarks in this directory.

Table 12.1 C Language Benchmarks

Benchmark	Description

`efr/src/autocorr`
`efr/src/chebps`
`efr/src/cor_h`
`efr/src/lag_max`
`efr/src/norm_corr`
`efr/src/search_10i40`
`efr/src/syn_filt`
`efr/src/vq_subvec`
	Enhanced Full Rate GSM vocoder standard C reference code benchmarks. These functions represent the most MIPS-consuming functions in the complete vocoder application.
	
	The results of these benchmarks are a good indication of the compiler performance on real DSP applications like the EFR.

`msample/src/bqa1`	Bi-Quad Simulation (1 sample)
`msample/src/bqa2`	Bi-Quad Simulation (multi-sample, 2 samples)
`msample/src/bqa4`	Bi-Quad Simulation (multi-sample, 4 samples)
`msample/src/cora1`	Correlation Simulation (1 sample)
`msample/src/cora2`	Correlation Simulation (multi-sample, 2 samples)
`msample/src/cora4`	Correlation Simulation (multi-sample, 4 samples)
`msample/src/fira1`	FIR Simulation (1 sample)
`msample/src/fira2`	FIR Simulation (multi-sample, 2 samples)
`msample/src/fira4`	FIR Simulation (multi-sample, 4 samples)
`msample/src/iira1`	IIR Simulation (1 sample)
`msample/src/iira2`	IIR Simulation (multi-sample, 2 samples)
`msample/src/iira4`	IIR Simulation (multi-sample, 4 samples)

Running the C Language Benchmarks

Two sample projects include all the sources you need to build the efr and msample benchmarks.

To run the C language benchmarks:

1. Open the CodeWarrior project file for either the `efr` or `msample` benchmarks. These benchmarks reside at these locations:
   - `*installDir*\Examples\StarCore_Examples\Benchmark\C\efr\efr.mcp`
   - `*installDir*\Examples\StarCore_Examples\Benchmark\C\msample\msample.mcp`

   Each of these projects contains a build target for each of the source code samples.
2. To build a particular benchmark, select the target name of interest from the current build target listbox of the project window.
3. Select **Project** > **Make**

Additional Examples

The directories listed below contain more example programs.

- *installDir*`\Examples\StarCore\c_asm_mix`
- *installDir*`\Examples\StarCore\Command_Line_Script_Debug`
- *installDir*`\Examples\StarCore\EOnCEDemo`
- *installDir*`\Examples\StarCore\FileIO`
- *installDir*`\Examples\StarCore\Profiler`
- *installDir*`\Examples\StarCore\Sc140`
- *installDir*`\Examples\StarCore\Simulator`

Assembly Language Benchmarks

The assembly language benchmark source code files are here:

installDir`\Examples\StarCore_Examples\Benchmark\asm`

There is an absolute and a relocatable version of each benchmark. The directories that contain each benchmark reside in one of the following directories, which are located in the overall benchmark directory mentioned in the preceding paragraph:

- `Absolute ASM`
- `Relocatable ASM`

Before using the relocatable version of a benchmark, you must specify the linker command file for the benchmark in the **Enterprise Linker** or **DSP Linker** settings panel. (The linker command file is the file in each relocatable benchmark directory that has the extension `.mem`.)

Table 12.2 lists the assembly language benchmarks.

Table 12.2 Assembly Language Benchmarks

Benchmark	Description
`blkmov`	Block move
`bq4`	4 multiply biquad filter
`bq5`	5 multiply biquad filter
`cfir`	Complex FIR filter
`cmax`	Complex maximum
`corr`	Correlation or convolution
`dotsq`	Dot product and square product
`eng`	Vector energy
`fft`	256 point FFT transform, radix 4
`iir`	IIR filter
`L1_norm`	Mean absolute error
`L2_norm`	Mean square error
`lfir`	Lattice FIR filter
`liir`	Lattice IIR filter
`lmsdly`	Delayed LMS filter
`minposr`	Minimum positive ratio
`minr`	Minimum ratio
`rmin`	Real minimum
`viterbi`	Veterbi decoder
`wht`	Walsh-Hadamard transform

Index