CodeWarrior™ Development Studio for StarCore® DSP Architectures

v2.6

Targeting Manual

Revised 2004/10/01

metrowerks

Metrowerks reserves the right to make changes to any product described or referred to in this document without further notice. Metrowerks makes no warranty, representation or guarantee regarding the merchantability or fitness of its products for any particular purpose, nor does Metrowerks assume any liability arising out of the application or use of any product described herein and specifically disclaims any and all liability. **Metrowerks software is not authorized for and has not been designed, tested, manufactured, or intended for use in developing applications where the failure, malfunction, or any inaccuracy of the application carries a risk of death, serious bodily injury, or damage to tangible property, including, but not limited to, use in factory control systems, medical devices or facilities, nuclear facilities, aircraft navigation or communication, emergency systems, or other applications with a similar degree of potential hazard**.

How to Contact Metrowerks

Corporate Headquarters	Metrowerks Corporation 7700 West Parmer Lane Austin, TX 78729 U.S.A.
World Wide Web	`http://www.metrowerks.com`
Sales	United States Voice: 800-377-5416 United States Fax: 512-996-4910 International Voice: +1-512-996-5300 E-mail: `sales@metrowerks.com`
Technical Support	United States Voice: 800-377-5416 International Voice: +1-512-996-5300 E-mail: `support@metrowerks.com`

Table of Contents

1

Introduction

This manual explains how to use the CodeWarrior™ Development Tools for StarCore® DSP Architectures product.

The sections of this chapter are:

- Read the Release Notes
- Related Documentation

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 this directory:

`installDir\Release_Notes`

where `installDir` is a placeholder for the directory in which you installed your CodeWarrior product.

Related Documentation

This manual provides high-level information about the CodeWarrior Development Studio for StarCore DSP Architectures product. For more detailed information, refer to these documents:

- *CodeWarrior IDE User Guide*

  Explains how to use general features of the CodeWarrior IDE, such as the project manager, editor, and debugger.

- *Metrowerks Enterprise C Compiler User Guide*

  Documents the C compiler included with the CodeWarrior for StarCore DSPs product.

- *SC100 Assembly Language Tools User Guide*

  Documents the assembler included with the CodeWarrior for StarCore DSPs product.

- *SC100 Linker User Guide*

  Documents the linker included with the CodeWarrior for StarCore DSPs product.
- *SC140 DSP Core Reference Manual*

  Documents the instruction set architecture and programming model for the SC140 core as well as corresponding register details and programming modes.
- *SC100 Application Binary Interface Reference Manual*

  Documents the SC100 Application Binary Interface (ABI). The ABI is a set of interface standards that writers of compilers, assemblers, and debugging tools must use when creating tools for the SC100 architecture.

You can download these documents from this web site:

`http://e-www.motorola.com/webapp/sps/library/docu_lib.jsp`

2

Installing Your CodeWarrior™ Product

This chapter explains how to install the CodeWarrior Development Studio for StarCore® DSP Architectures product.

The sections are:

- System Requirements
- Installing the CodeWarrior™ Software

System Requirements

The system requirements for the Windows®-hosted and Solaris-hosted StarCore tools differ. The following sections define the requirements for each platform.

- Windows® PC
- Solaris™ Workstation

Windows® PC

Table 2.1 lists the hardware and software required to run CodeWarrior for StarCore DSPs on a Windows PC.

Table 2.1 System Requirements—Windows® PC

Hardware	266 MHz Pentium® II class processor or better 128 MB of memory (minimum), CD-ROM drive
Operating System	Windows NT® 4.0 (with service pack 6 or later), Windows® 2000 (with service pack 3 or later), or Windows® XP
Free Disk Space	350 MB (minimum)

Solaris™ Workstation

Table 2.1 lists the hardware and software required to run CodeWarrior for StarCore DSPs on a Solaris workstation.

Table 2.2 System Requirements—Solaris™ Workstation

Hardware	Sun Microsystems™ SPARC workstation 128 MB of memory (minimum) CD-ROM drive Free PCI slot (if using PCI command converter)
Operating System	Solaris 7 or Solaris 8 operating system (needed for local hardware debugging)
Free Disk Space	475 MB (minimum)

Installing the CodeWarrior™ Software

The installation procedure for the Windows-hosted and Solaris-hosted tools differ. The sections that follow explain how to install your CodeWarrior product on each machine.

- Windows® PC Instructions
- Solaris™ Workstation Instructions

Windows® PC Instructions

NOTE You must have administrator privileges to install this CodeWarrior product.

To install CodeWarrior for StarCore DSPs on a Windows PC, follow these steps:

1. Put the installation CD in CD drive.

   The installation menu appears.

NOTE If auto-install is disabled, run `Launch.exe` manually. This program is in the root directory of the installation CD.

2. In the installation menu, click **Launch the installer**

   The Install wizard starts and displays its welcome page.

3. Follow the wizard's on-screen instructions to install the software.
4. When prompted to check for CodeWarrior software updates, click **Yes**

   The **CodeWarrior Updater** window appears.

NOTE If the **CodeWarrior Updater** already has the required Internet connection settings, proceed directly to step 8.

5. Click **Settings**

   The **Internet Properties** dialog box appears.

6. Use this dialog box to modify your Internet settings, if necessary.
7. Click **OK**

   The **Internet Properties** dialog box closes.

8. In the **CodeWarrior Updater** window, click **Next**

   The updater checks for newer versions of the CodeWarrior products installed on your PC.

9. Follow the updater's on-screen instructions to download CodeWarrior product updates to your PC.
10. When the updater displays the message *Update Check Complete!*, click **Finish**

    The Wizard displays a message box that gives you the option to read the product release notes.

11. Click **Yes**

    The Wizard displays the release notes in your web browser.

12. When you are finished reading release notes, exit the web browser.

    The Wizard displays its "installation complete" page.

13. Select **Yes, I want to restart my computer now** and click **Finish**

    Your PC restarts. Software installation is complete.

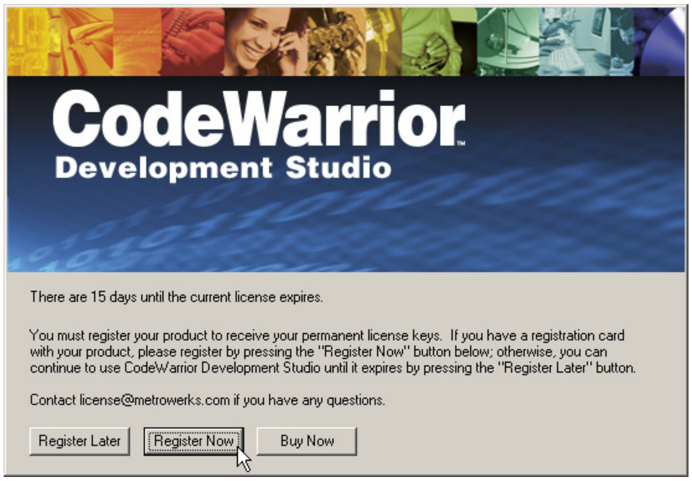
14. Register and license your software.

    To do this, follow these steps:

    a. Select **Start > Programs > Metrowerks CodeWarrior > CodeWarrior for StarCore 2.6 > CodeWarrior IDE**

       The IDE starts and displays the registration dialog box. (See Figure 2.1.)

Figure 2.1 Registration Dialog Box



There are 15 days until the current license expires.

You must register your product to receive your permanent license keys. If you have a registration card with your product, please register by pressing the "Register Now" button below; otherwise, you can continue to use CodeWarrior Development Studio until it expires by pressing the "Register Later" button.

Contact license@metrowerks.com if you have any questions.

b. Click **Register Now**

The registration dialog box closes. Your web browser starts and displays Metrowerks registration web page.

c. Use the registration web page to enter your registration information.

In response, Metrowerks e-mails you a license authorization code (typically within 30 minutes).

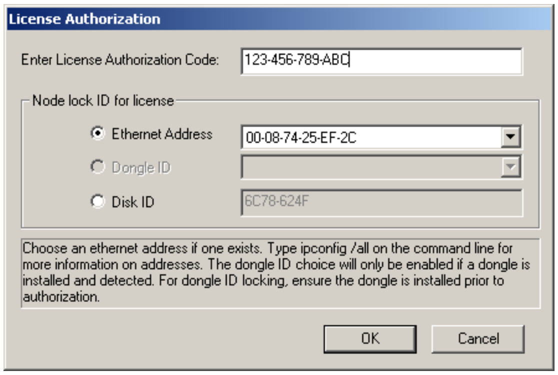
NOTE To register your CodeWarrior product, you must have a registration code. This code is printed on the registration card included with your product.

d. Open the Metrowerks e-mail message containing your license registration code.

e. From the IDE's menu bar, select **Help > License Authorization**

The **License Authorization** dialog box appears. (See Figure 2.2.)

Figure 2.2 License Authorization Dialog Box



License Authorization

Enter License Authorization Code: 123-456-789-ABC

Node lock ID for license

● Ethernet Address: 00-08-74-25-EF-2C

○ Dongle ID

○ Disk ID: 6C78-624F

Choose an ethernet address if one exists. Type ipconfig /all on the command line for more information on addresses. The dongle ID choice will only be enabled if a dongle is installed and detected. For dongle ID locking, ensure the dongle is installed prior to authorization.

f. Copy and paste your license authorization code from the e-mail message to the Enter License Authorization Code text box.

g. Follow the instructions in the dialog box to supply a license node lock ID.

h. Click **OK**

   The **License Authorization** dialog box closes.

Your CodeWarrior software is now installed, registered, and licensed.

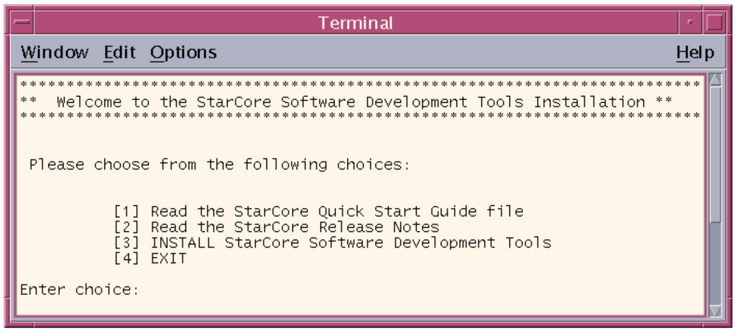
Solaris™ Workstation Instructions

To install CodeWarrior for StarCore DSPs on a Solaris workstation, follow these steps:

1. Put the installation CD in the CD-ROM drive.
2. Open a terminal window.
3. Log in as root or super user.
4. Mount the CD-ROM media on the file system.
5. Set the path to this directory:
   `/cdrom/codewarriorforsolaris2.6/`
6. Type `installCW` and press **Return**

   The Installer menu appears. (See Figure 2.3.)

Figure 2.3 Installer Menu



```
*************************************************************************
**  Welcome to the StarCore Software Development Tools Installation **
*************************************************************************


 Please choose from the following choices:


          [1] Read the StarCore Quick Start Guide file
          [2] Read the StarCore Release Notes
          [3] INSTALL StarCore Software Development Tools
          [4] EXIT

Enter choice:
```

7. Type `3` and press **Return**

   The Installer asks for the destination directory

8. Enter destination directory and press **Return**

   Product installation begins.

9. Follow the on-screen installation instructions to complete product installation.

10. When installer menu reappears, type `4` and press **Return**

    The Installer exits.

11. Register and license the software.

    To do this, follow these steps:

    a. Start your web browser and enter this location:
    `http://metrowerks.com/mw/register/`

    The Metrowerks registration web page appears.

    b. From the License Type listbox, select `New Purchase`

    c. In the Registration Code text box, enter the registration code printed on registration card included in your CodeWarrior software package.

    d. Click **Continue Registration**

    The Validation page appears.

    e. Verify that the information on this page is correct.

    f. Click **Continue Registration**

    The User Information page appears.

    g. Enter required user information on this page.

    h. Click **Complete Registration**

    The Thank You page appears. Metrowerks sends a message containing an authorization code to the specified e-mail address.

i. Follow the directions in the Metrowerks e-mail to complete product activation.

Your CodeWarrior software is now installed, registered, and licensed.

NOTE E-mail licensing questions to `license@metrowerks.com`.

3

Overview: The CodeWarrior™ for StarCore® DSP Tools

This chapter provides an overview of the StarCore-specific development tools included with your CodeWarrior™ product.

The sections are:

- Metrowerks™ Enterprise C Compiler
- SC100 Assembler
- SC100 Linker
- CodeWarrior™ Debugger
- StarCore® Utilities

Metrowerks™ Enterprise C Compiler

The Metrowerks Enterprise C Compiler:

- Conforms to the American National Standards Institute (ANSI) C standard.
- Conforms to version 1 of the StarCore Application Binary Interface (ABI) standard.
- 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 for standard C constructs for representing special addressing modes.
- Supports a wide range of runtime libraries and runtime environments.
- Optimizes for size (smaller code), speed (faster 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 code. The compiler performs many optimizations, including these:

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

For documentation of this tool, see the *Metrowerks Enterprise C Compiler User Guide*.

SC100 Assembler

The assembler translates assembly language source code to machine language object files or executable programs. Assembly language source code can be either hand written or generated by the compiler.

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

For documentation of the assembler, see the *SC100 Assembler User Guide*.

SC100 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 the LCF you require.

For documentation of the linker, see the *SC100 Linker User Guide*.

CodeWarrior™ Debugger

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

If you debug in conjunction with a simulator, you have the additional option of analyzing code performance using the iCacheViewer. See Using the Profiler for instructions.

Using the Optimized Code Debugger extension of the debugger, you can debug even highly optimized code. See Debugging Optimized Code for instructions.

StarCore® Utilities

CodeWarrior for StarCore DSPs includes these software development utilities:

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

Archiver

The 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 Archiver Utility for instructions that explain how to use the archiver.

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

See disasmsc100 Disassembler for instructions that explain how to use this utility.

ELF File Dump Utility

The ELF file dump utility outputs the headers of each ELF object file passed on the command line in a human-readable form. The information generated by the ELF dump utility depend on the type of ELF object file:

- Executable object file

  The default output is the ELF header, all program headers, and all sections headers.

- Relocatable object file

  The default output is the ELF header and all section headers.

See ELF Dump Utility for instructions that explain how to use this utility.

Name Utility

The name utility displays the symbolic information of each object file and library passed on the command line. If a file contains no symbolics, the utility reports this fact.

See Name Utility for instructions that explain how to use this utility.

Size Utility

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

See Size Utility for instructions that explain how to use this utility.

4

Tutorial: Using the CodeWarrior™ for StarCore® DSP Tools

This chapter consists of a tutorial that shows you how to create, build, and debug a StarCore® DSP project using the CodeWarrior for StarCore DSP development tools.

The sections are:

- Using Stationery
- Creating a Project
- Debugging a Project

Using Stationery

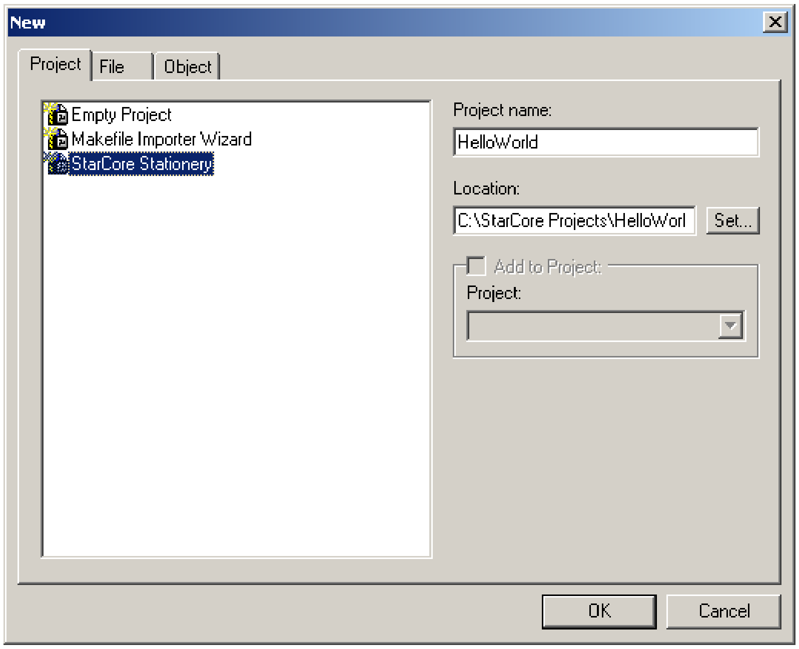
You create most new projects using *project stationery*. Project stationery is a collection of projects for the various StarCore debug targets.You can use these prebuilt projects as templates for constructing your own new projects.

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

1. Select **File > New**

   The **New** window appears. (See Figure 4.1.)

Figure 4.1 The New Window



Project name: HelloWorld

Location: C:\StarCore Projects\HelloWorl

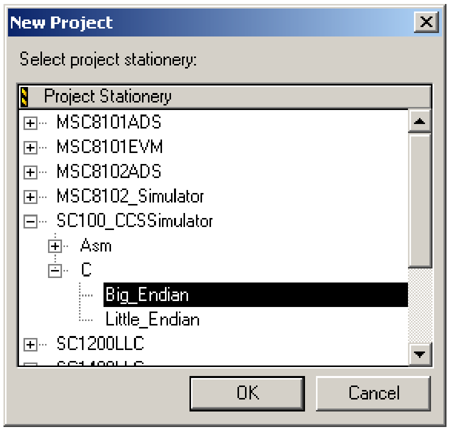
□ Add to Project:

Project:

2. In the Project name text box, type the name of the new project.
3. In the Location text box, enter the path in which to create the project.
4. Click **OK**

   The **New Project** window appears. (See Figure 4.2.)

Figure 4.2 New Project Window



In this dialog box, there is project stationery for these devices and simulators:

- MSC8101ADS
- MSC8101EVM
- MSC8102ADS
- MSC8102 Simulator
- SC100_CCSSimulator
- SC1200LLC
- SC1400LLC
- SC140 SDP
- SCLLC_FPGA_Eval
- SCLLC_Simulator
- SmartDSP_OS
- StarCore Librarian

In addition, for some of these devices, there is stationery for both C language projects assembly language projects. Further, for some devices, there is stationery for both big endian and little endian memory organization.

5. From the stationery list, select the stationery for the project type you want to create.
6. Click **OK**

   The IDE creates a CodeWarrior project using the selected stationery and displays the project in a project window.

Creating a Project

In this tutorial, you create a project using the SC100_CCSSimulator project stationery, add some source code, make some target settings, and build the project.

The sections are:

- Create a Project
- Add a New Source File
- View Target Settings
- Build the Project

Create a Project

Create a StarCore project using the **SC100_CCSSimulator > C > Big Endian** project stationery. To do this, follow the steps provided in "Using Stationery" on page 29.

Add a New Source File

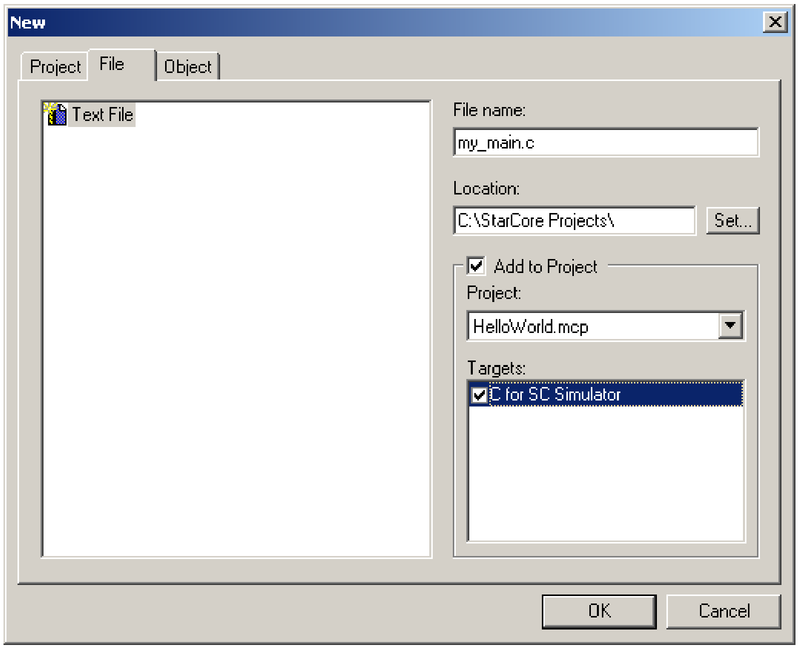
1. Choose **File > New**

   The **New** window appears.

2. Click the **File** tab of the **New** window.
3. Type this file name in the File name field
   `my_main.c`
4. Check the Add to Project checkbox.
5. Ensure that the Project listbox displays the name of your project.
6. In the Targets listbox, check the box next to the target to which to add the file.

   Figure 4.3 shows the **New** window.

Figure 4.3 New Window—Set up for Adding a File to a Project



7. Click **Set** to navigate to a different directory and save the file or click **OK** in the **New** window to accept the default location.

   An editor window appears with the name you specified and the IDE adds the file to the specified project.

8. In the editor window, type (or copy and paste) the source code shown in Listing 4.1.

Listing 4.1 Example Source Code

```
#include <stdio.h>

int a = 5;
int b = 10;
int c = 0;

void main(void)
{

  printf("Hello StarCore!\n");
  do {
    a++;
    b++;
    c = a + b;

    printf("The current value of a is: %d \n", a);
    printf("The current value of b is: %d \n", b);
    printf("The current value of c is: %d \n", c);
  } while (c < 100);
}
```

9. Choose **File > Save** and close the file.

10. Remove the placeholder source file.

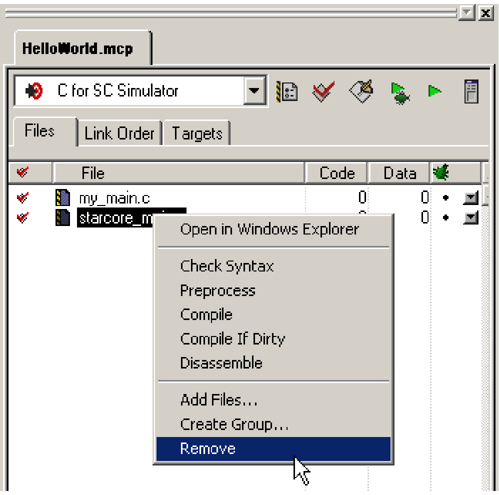
    (The file `starcore_main.c` is a placeholder for your own main source file.)

    a. In the project window, select `starcore_main.c`.

    b. Right-click on `starcore_main.c`.

    c. Select Remove from the context menu that appears. (See Figure 4.4.)

Figure 4.4 Removing a File from a Project



    d. Click **OK** in the confirmation dialog box that appears.

    The IDE deletes the file from the project (but leaves it on the hard disk).

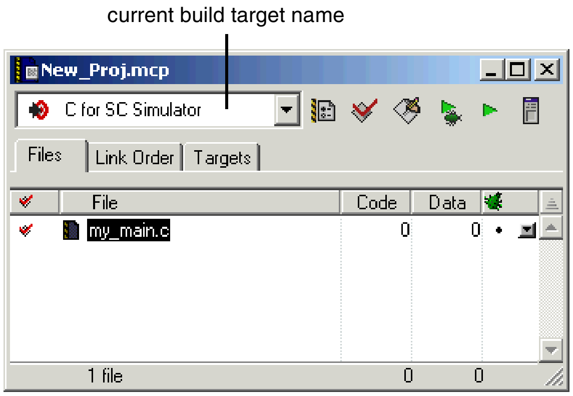
View Target Settings

To view target settings:

1. If you need to change the current build target, choose **Project > Set Default Target > *Target Name*** (where *Target Name* is the name of the target to make the current target).

   The project window displays the current build target name in the build target list box. (See Figure 4.5.)

Figure 4.5 The Project Window and Current Build Target

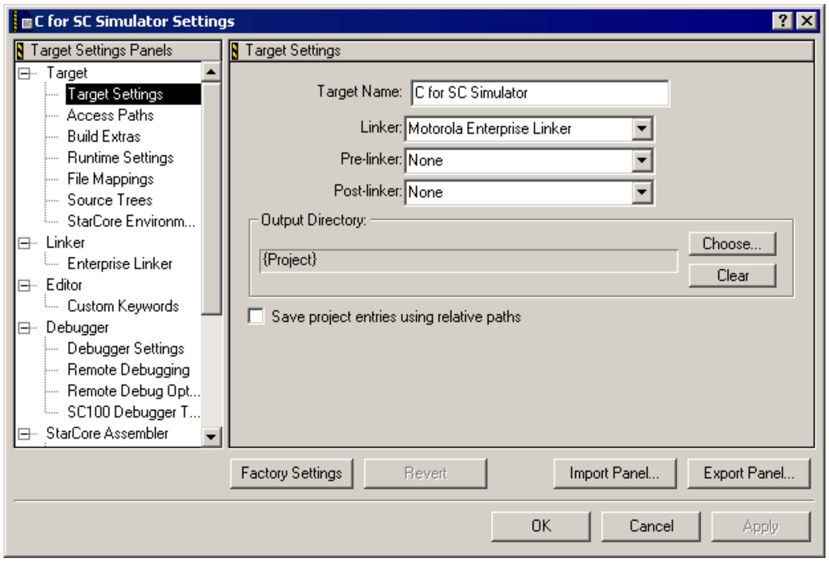


2. Choose **Edit > *Target Name* Settings**.

NOTE For this example, choose **Edit > C for SC Simulator Settings**

The **Target Settings** window appears. (See Figure 4.6.)

Figure 4.6 The Target Settings Window



C for SC Simulator Settings

Target Settings

Target Name: C for SC Simulator

Linker: Motorola Enterprise Linker

Pre-linker: None

Post-linker: None

Output Directory: {Project}

□ Save project entries using relative paths

TIP To quickly display a build target's settings, display the Targets view of the project window and double-click the build target name(s) of interest. Using this method, you can display the settings for two or more build targets simultaneously.

The **Target Settings** window groups all available build target settings into a series of panels. The list of panels appears on the left side of the window. When you select a panel, the options in that panel appear on the right side of the dialog box.

Different panels affect:

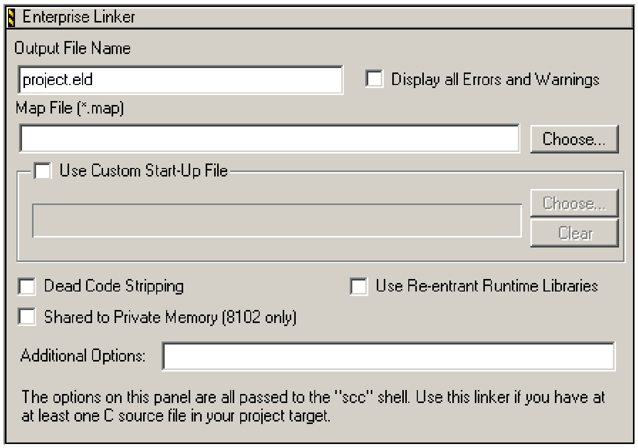
- Settings related to all build targets
- Settings that are specific to a particular build target (including settings that affect code generation and linker output)
- Settings related to a particular programming language

3. Select Enterprise Linker from the list of panels in the **Target Settings** window.

   The **Target Settings** window displays the Enterprise Linker panel. (See Figure 4.7.)

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

Figure 4.7 Enterprise Linker Panel



Enterprise 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)

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.

Examine the other settings before closing the **Target Settings** window.

Build the Project

To build the project, choose **Project > Make**

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

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

Debugging a Project

The debugging section of the tutorial shows you how to do these things:

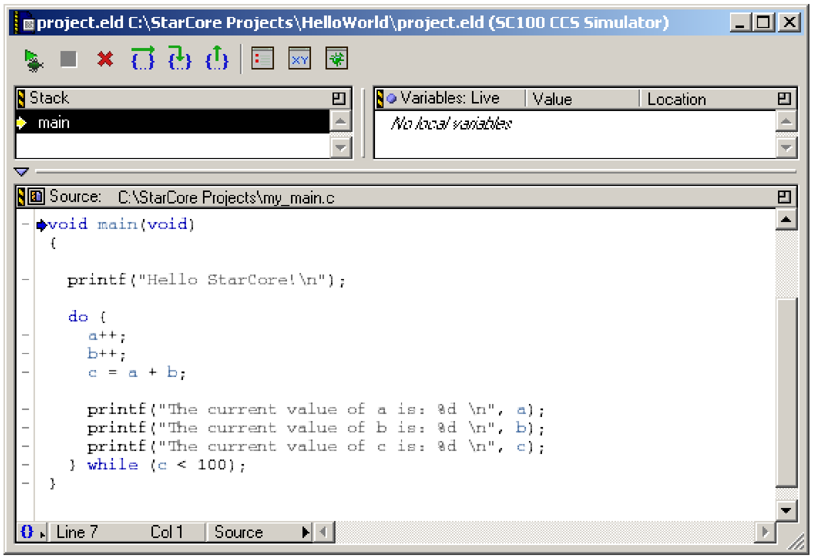
- Start Debugging
- Set a Breakpoint
- Show Registers
- Finish Debugging

Start Debugging

To run your project under control of the CodeWarrior debugger, choose **Project > Debug**.

The debugger displays a message box while downloading your program to the target board, and then the debugger window appears. (See Figure 4.8.)

Figure 4.8 Debugger Window


project.eld C:\StarCore Projects\HelloWorld\project.eld (SC100 CCS Simulator)
Stack
main
Variables: Live
Value
Location
No local variables
Source: C:\StarCore Projects\my_main.c
Line 7
Col 1
Source

```
void main(void)
{

   printf("Hello StarCore!\n");

   do {
     a++;
     b++;
     c = a + b;

     printf("The current value of a is: %d \n", a);
     printf("The current value of b is: %d \n", b);
     printf("The current value of c is: %d \n", c);
   } while (c < 100);
}
```

Set a Breakpoint

To set a breakpoint:

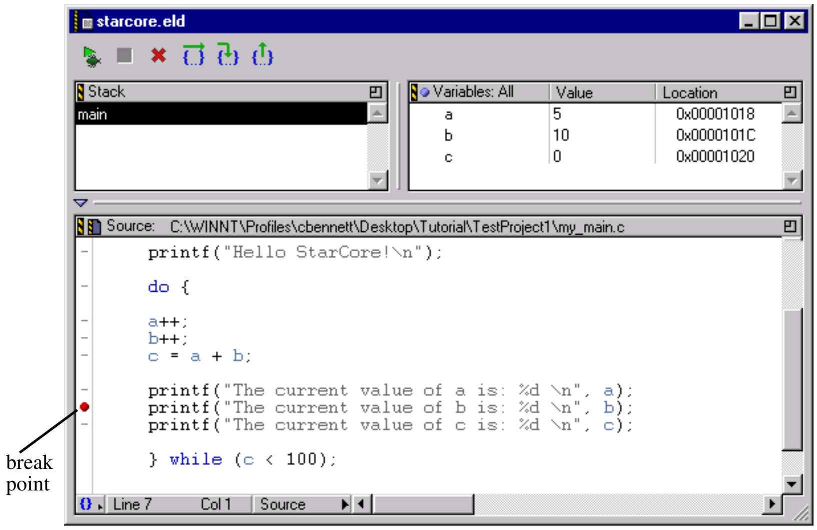
1. In the debugger window, click the gray dash in the breakpoint column next to this statement:

   ```
   printf("The current value of b is: %d \n", b);
   ```

   A red dot appears next to the statement. (See Figure 4.9.)

NOTE You also can set a breakpoint by clicking in the breakpoint column to next to any executable statement displayed in an editor window.

Figure 4.9 Debugger Window after Setting a Breakpoint


starcore.eld
Stack
main
Source: C:\WINNT\Profiles\cbennett\Desktop\Tutorial\TestProject1\my_main.c
Line 7
Col 1
Source
break point

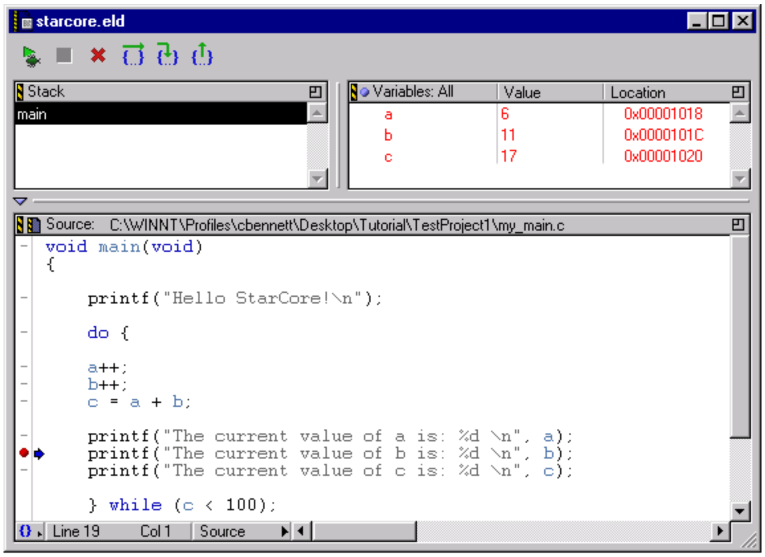
Variables: All	Value	Location
a	5	0x00001018
b	10	0x0000101C
c	0	0x00001020

```
    printf("Hello StarCore!\n");

    do {

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

    printf("The current value of a is: %d \n", a);
    printf("The current value of b is: %d \n", b);
    printf("The current value of c is: %d \n", c);

    } while (c < 100);
```

2. Select **Project > Run** to run to the breakpoint just set.

   Figure 4.10 shows the debugger window after your program has hit the breakpoint.

Figure 4.10 Debugger Window after Running to Breakpoint

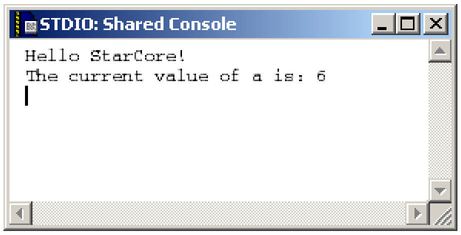

starcore.eld
Stack
main
Source: C:\WINNT\Profiles\cbennett\Desktop\Tutorial\TestProject1\my_main.c
Line 19
Col 1
Source

Variables: All	Value	Location
a	6	0x00001018
b	11	0x0000101C
c	17	0x00001020

```
void main(void)
{

    printf("Hello StarCore!\n");

    do {

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

    printf("The current value of a is: %d \n", a);
    printf("The current value of b is: %d \n", b);
    printf("The current value of c is: %d \n", c);

    } while (c < 100);
```

In addition, the IDE displays an output window. (See Figure 4.11.)

Figure 4.11 Example Program—Output Window



```
Hello StarCore!
The current value of a is: 6
```

That's it. You just executed your program under control of the debugger, set a breakpoint, and let your program run to this breakpoint.

Show Registers

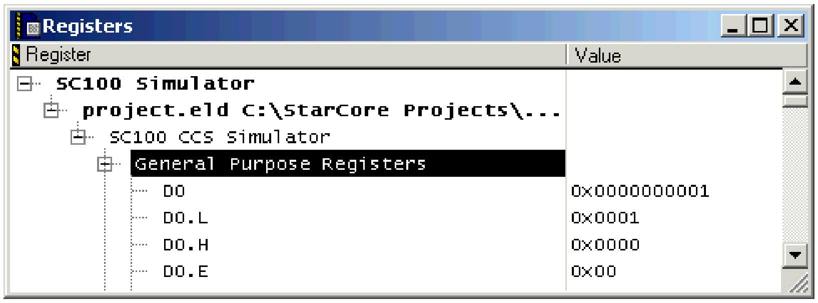
To display registers, follow these steps:

1. Choose **View > Registers**

   The **Registers** window appears. This window displays a tree control that lets you display the registers of the StarCore processor you are using. (See Figure 4.12.)

NOTE The **Registers** window displays different registers depending on the type of StarCore processor being used.

Figure 4.12 Registers Window



```
SC100 Simulator
  project.eld C:\StarCore Projects\...
    SC100 CCS Simulator
      General Purpose Registers
        D0        0x0000000001
        D0.L      0x0001
        D0.H      0x0000
        D0.E      0x00
```

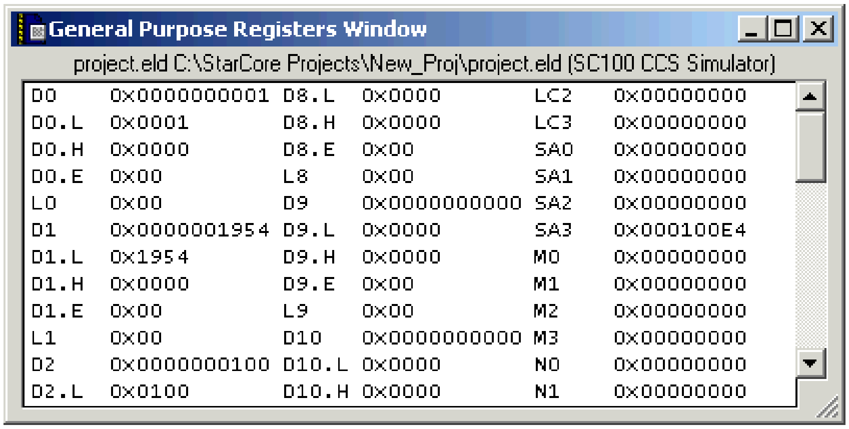
2. Choose a register from the menu.

   For this example, double-click:

   `SC100 CCS Simulator > General Purpose Registers`

   The CodeWarrior IDE displays an information window for the selected registers. (See Figure 4.13.)

Figure 4.13 General Purpose Registers Window



project.eld C:\StarCore Projects\New_Proj\project.eld (SC100 CCS Simulator)

```
D0     0x0000000001 D8.L  0x0000       LC2   0x00000000
D0.L   0x0001       D8.H  0x0000       LC3   0x00000000
D0.H   0x0000       D8.E  0x00         SA0   0x00000000
D0.E   0x00         L8    0x00         SA1   0x00000000
L0     0x00         D9    0x0000000000 SA2   0x00000000
D1     0x0000001954 D9.L  0x0000       SA3   0x000100E4
D1.L   0x1954       D9.H  0x0000       M0    0x00000000
D1.H   0x0000       D9.E  0x00         M1    0x00000000
D1.E   0x00         L9    0x00         M2    0x00000000
L1     0x00         D10   0x0000000000 M3    0x00000000
D2     0x0000000100 D10.L 0x0000       N0    0x00000000
D2.L   0x0100       D10.H 0x0000       N1    0x00000000
```

Finish Debugging

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

That’s it. Now, you know how to create a project, build it, and debug it.

5

Target Settings

This chapter documents the StarCore®-specific target settings panels.

Use these panels to control the behavior of the compiler, linker, debugger, and other CodeWarrior™ software development tools for StarCore DSP architectures.

NOTE For documentation of the target settings panels included in all CodeWarrior products, see the *IDE User Guide*.

The sections of this chapter are:

- Overview: Target Settings
- StarCore®-Specific Target Settings Panels

Overview: Target Settings

A CodeWarrior project contains one or more *build targets*. A build target is a named collection of files and settings that the CodeWarrior IDE uses to generate an output file.

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.

The build target feature lets you to create different versions of your program for different purposes. For example, you might have a *debug* build target. This build target would include no optimizations, so it is easy to debug. You might also have a *release* build target. This build target would be heavily optimized so it uses less memory or runs faster.

Changing Target Settings

If you create a project using stationery, the target settings of each build target are automatically set to reasonable defaults. That said, you may need to change some of them.

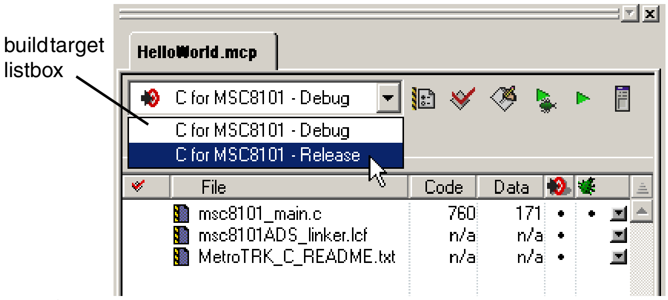
To change a build target’s target settings, follow these steps:

1. Start the CodeWarrior IDE.
2. Open the project that contains the build target to be modified.

   The IDE displays the project in a project window (docked to the left and bottom of the IDE’s main window).

3. From the build target listbox of the project window, select the build target that you want to modify. (See Figure 5.1.)

Figure 5.1 Project Window Showing the Selection of a Build Target



File	Code	Data		
msc8101_main.c	760	171	•	•
msc8101ADS_linker.lcf	n/a	n/a	•	
MetroTRK_C_README.txt	n/a	n/a	•	

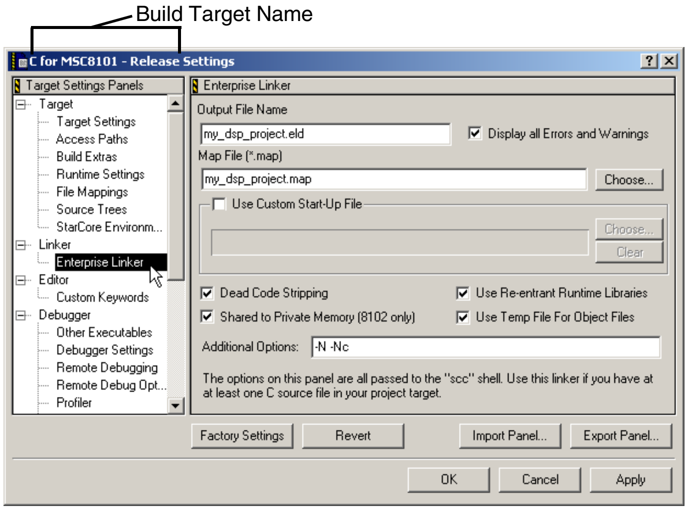
4. Press **ALT-F7**

The ***Target*** **Settings** window appears. (See Figure 5.2.)

NOTE In the sentence above, the word *Target* is in italics because it is a placeholder for the name of the current build target. For example, in Figure 5.2, the string C for MSC8101 - Release appears in place of *Target*.

The settings you make in the panels of the ***Target*** **Settings** window apply to the project build target currently selected.

Figure 5.2 The *Target* Settings Window Showing the Enterprise Linker Panel



Output File Name: my_dsp_project.eld

☑ Display all Errors and Warnings

Map File (*.map): my_dsp_project.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: -N -Nc

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.

On the left side of the **Target Settings** window is the Target Settings Panels list. This list contains the name of each target settings panel available for the current build target. Your selections for Linker and Post-linker in the Target Settings panel determine the panel names in this list.

5. In the Target Settings Panels list, click a target settings panel name.

   The selected panel appears in the right side of the **Target Settings** window.

   Figure 5.2 shows the **Enterprise Linker** target settings panel.

6. Change the settings in the displayed panel as dictated by the build target's purpose.
7. Click **Apply**

   The IDE saves your new settings.

8. In the Target Settings Panels list, click a different target settings panel name.

   The selected panel replaces the **Enterprise Linker** panel.

9. Again, change the settings in the panel as dictated by the build target's purpose.
10. Click **Apply**

    The IDE saves your new settings.

11. Continue this process for each target settings panel until you have made all settings your build target requires.
12. When you are done making settings, click **OK**

    The IDE saves your settings and closes the **Target Settings** window.

Creating Stationery

Once you have made the required settings for each build target of a project, you might want to save the project as stationery. This lets you and others can create new projects identical to the current one.

To create 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 desired.
3. Select **File > Save a Copy As**

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

4. Use this dialog box to save the project in a subdirectory of the CodeWarrior stationery directory.

   The CodeWarrior stationery directory is here:

   `installDir\Stationery`

   where `installDir` is a placeholder for the directory in which you installed your CodeWarrior product.

That's it. The next time you create a project using stationery, the stationery just created appears in the **New Project** dialog box.

Restoring Target Settings

If you change any of the settings of a build target, you can recover the original values.

To restore a build target's original settings, use one of these methods:

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

General Purpose Target Settings Panels

Some target settings panels are needed for all development done with the CodeWarrior IDE. Other panels are specific to the CodeWarrior for StarCore DSPs product.

Table 5.1 lists each target settings panel that is *not* StarCore-specific. Refer to the *IDE User Guide* for documentation of these panels.

Table 5.1 General Purpose Target Settings Panels

Target Settings Panel	Description
Access Paths	Use this target settings panel to define the list of directories that the build tools search for include files.
Build Extras	Use this target settings panel to select options that affect the performance of the software development tools. In addition, use the panel to set up a third-party debugger.
Runtime Settings	Use this target settings panel to supply information, such as command-line arguments, that your program needs when run under control of the CodeWarrior IDE.

Table 5.1 General Purpose Target Settings Panels (*continued*)

Target Settings Panel	Description
File Mappings	Use this target settings panel to associate each file extension with a tool used to manipulate file’s that have that extension.
Source Trees	Use this target settings panel to define aliases for paths that change from one developer’s workstation to another’s. Using source trees makes it easier to share a project.
Custom Keywords	Use this target setting panel to define up to four sets of custom keywords along with the color the editor uses for each.
Other Executables	Use this target settings panel to define the list of executables and shared libraries to debug along with the build target’s primary binary.
Debugger Settings	Use this target setting panel to configure the general (that is, not StarCore-specific) behavior of the debugger.t

StarCore®-Specific Target Settings Panels

Table 5.2 lists and describes each StarCore-specific target settings panels.

Table 5.2 StarCore®-Specific Target Settings Panels

Target Settings Panel	Description
Target Settings	Use this panel to define the name of the current build target, and the linker, pre-linker, post-linker, and output directory this build target uses.
StarCore Environment	Use this target settings panel to specify the StarCore architecture the build target is targeting, the endianness and memory mode this architecture uses, and how the IDE handles the command lines it passes to the shell program (scc).
Enterprise Linker	Use this target settings panel to select options that control the behavior of the Motorola Enterprise Linker. **NOTE:** This panel is available only if you select Motorola Enterprise Linker from the Linker listbox of the Target Settings panel.

Table 5.2 StarCore®-Specific Target Settings Panels (*continued*)

Target Settings Panel	Description
DSP Linker	Use this target settings panel to select options that control the behavior of the Motorola DSP Linker. **NOTE:** This panel is available only if you select Motorola DSP Linker from the Linker listbox of the Target Settings panel.
DSP Librarian	Use this target settings panel to specify the name of the library the build target uses and to pass command-line options to the archiver utility. **NOTE:** This panel is available only if you select DSP Librarian from the Linker listbox of the Target Settings panel.
Other Executables	Use this target settings panel to list other projects and files for the debugger to use in addition to the executable generated by the current build target. **NOTE:** to use the multi-core debugging feature, add the path and name of the CodeWarrior projects for each core to be debugged to this panel.
Remote Debugging	Use this target settings panel to select and configure the connection the CodeWarrior debugger uses to communicate with your target device or simulator.
Remote Debug Options	Use this panel to tell the debugger which parts of your program to download to the target, when to download these parts, and whether to verify them.
Profiler	Use this panel to configure the CodeWarrior profiler so it can interact with your target and collect information about the program running on this target.
SC100 Debugger Target	Use this target settings panel to select the simulator or device on which you will debug the binary produced by a build target. In addition, use the panel to control how the CodeWarrior debugger interacts with the selected device and how the debugger behaves at startup and during a debug session.
Assembler Preprocessors	Use this target settings panel to define the directories in which the assembler looks for files, how the assembler handles these files, and to pass the model number and revision of the processor you are using to the assembler.

Table 5.2 StarCore®-Specific Target Settings Panels (*continued*)

Target Settings Panel	Description
Listing File Options	Use this target settings panel to define how the assembler formats the listing file it generates. In addition, use this panel to pass command-line options to the assembler.
Code & Language Options	Use this target settings panel to select code and symbol generation options for the assembler.
C Language	Use this target settings panel to make settings that tell the compiler the version of the C language you are using.
Enterprise Compiler	Use this target settings panel to define the behavior of the Enterprise C compiler for the current build target.
I/O & Preprocessors	Use this target settings panel to specify additional directories in which the compiler will search when looking for include files. In addition, use this panel to define and undefine preprocessor macros.
Optimizations	Use this target settings panel to define and configure the optimizations the Enterprise C compiler performs.
Passthrough, Hardware	Use this target settings 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.
Source Folder Mapping	Use this target settings panel if you are debugging an executable file that was built in one place, but which is being debugged from another.
SC100 ELF Dump	Use this target settings panel to define the configure the behavior of the ELF file dump utility. **NOTE:** This panel is available only if you select SC100 ELF Dump from the Post-linker listbox of the Target Settings panel.
SC100 ELF to LOD	Use this target settings panel to define the name of the LOD file generated by the elflod utility. **NOTE:** This panel is available only if you select SC100 ELF to LOD from the Post-linker listbox of the Target Settings panel.

Table 5.2 StarCore®-Specific Target Settings Panels (*continued*)

Target Settings Panel	Description
SC100 ELF to S-Record	Use this target settings panel to define the name, addressability, and memory offset of the S-Record file generated by the elfsrec utility. **NOTE:** This panel is available only if you select SC100 ELF to S-Record from the Post-linker listbox of the Target Settings panel.

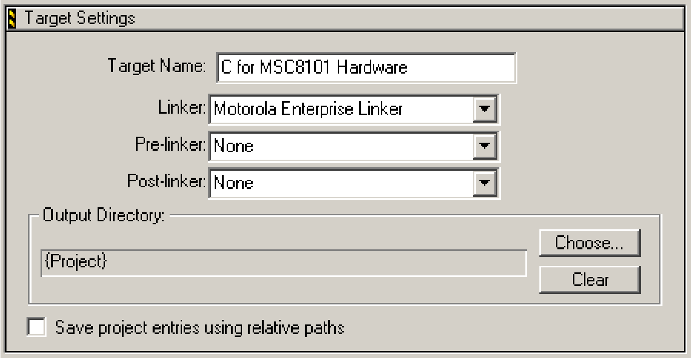
Target Settings

Use the **Target Settings** panel to define the name of the current build target, and the linker, pre-linker, post-linker, and output directory this build target uses.

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 5.3 shows the **Target Settings** panel.

Figure 5.3 Target Settings Panel



Target Settings

Target Name: C for MSC8101 Hardware

Linker: Motorola Enterprise Linker

Pre-linker: None

Post-linker: None

Output Directory:

{Project}

□ Save project entries using relative paths

Table 5.3 lists and defines each option of the **Target Settings** panel.

Table 5.3 Target Settings Panel Options

Option	Description
Target Name	Use this text box to assign a name to the current build target. The name you specify appears in the build target listbox and in the Targets view of a project window. **NOTE:** Target name is the name of the current build target, not the name of the file this build target generates. You define a build target's output file name in the Output File Name text box of the Enterprise Linker, DSP Linker, or DSP Librarian panel.
Linker	Use this listbox to select the linker the current build target uses. The choices are: • Motorola DSP Linker — Select this option to instruct the IDE to directly invoke 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. • Motorola Enterprise Linker or StarCore LLC Linker — Select either of these options to instruct the IDE to invoke 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 or DSP LLC Librarian — Select either of these linkers for a build target that puts its output in a library (`.elb`) file. **NOTE:** Use the DSP Librarian panel to specify the library a build target uses and whether the build target replaces or adds it output to the library. **NOTE:** Your linker choice determines which other target settings panels appear in the panel list of the **Target Settings** window.
Pre-linker	Unused in the CodeWarrior for StarCore DSPs product.

Table 5.3 Target Settings Panel Options (*continued*)

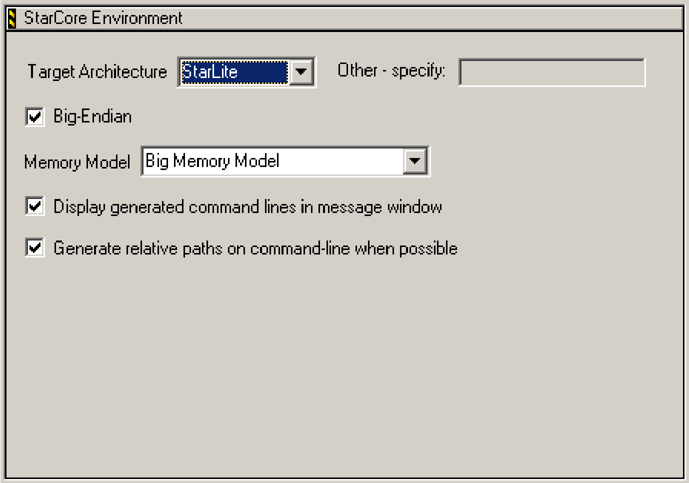
Option	Description
Post-linker	Use this listbox to select the post-linker current build target uses. The choices are: • None — Use no post-linker. • SC100 ELF Dump — Select this post-linker to run the sc100-elfdump utility on the output generated by the build target. If you select this post-linker, the SC100 ELF Dump panel appears in the panel list. Use this panel to define the behavior of the utility. See ELF Dump Utility for more information • SC100 ELF to LOD — Select this post-linker to run the elflod utility on the output generated by the build target. If you select this post-linker, the SC100 ELF to LOD panel appears in the panel list. Use this panel to define the behavior of the utility. See ELF to LOD Utility for more information • SC100 ELF to S-Record — Select this post-linker to run the elfsrec utility on the output generated by the build target. If you select this post-linker, the SC100 ELF to S-Record panel appears in the panel list. Use this panel to define the behavior of the utility. See ELF to S-Record Utility for more information
Output Directory	This read-only text box contains the path to which the build target writes its output. Click **Choose** to display a dialog box to use to select the desired output path. Click **Clear** to restore the default directory (the project directory).
Save project entries using relative paths	Check this box to instruct the IDE to save the relative path of each file in a build target along with the root file name of the file. If this box is checked, you can add two or more files that have the same name to a project. This is so because, 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, thereby producing a unique filename. If this box is unchecked, each file in a project must have a unique name because, 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 discriminate between two files that have the same name but different relative paths.

StarCore Environment

Use the **StarCore Environment** target settings panel to specify the StarCore architecture the build target is using, the endianness and memory mode this architecture uses, and how the IDE handles the command lines it passes to the shell program (scc).

Figure 5.4 shows the **StarCore Environment** target settings panel.

Figure 5.4 StarCore Environment Target Settings Panel



StarCore Environment

Target Architecture: StarLite ▾ Other - specify: [ ]

☑ Big-Endian

Memory Model: Big Memory Model ▾

☑ Display generated command lines in message window

☑ Generate relative paths on command-line when possible

Table 5.4 lists and defines each option of the **StarCore Environment** panel.

Table 5.4 StarCore® Environment Panel Options

Option	Description
Target Architecture	Use this listbox to select 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 listbox to the StarCore development tools. The IDE passes `-arch` followed by the string you enter to the tools. **NOTE:** The Other - specify text box is disabled unless you select OtherArch from the Target Architecture listbox.
Big-Endian	Check this box to instruct the compiler, assembler, and linker to use big-endian byte-ordering. Leave this box unchecked to instruct these tools to use little-endian byte ordering.

Table 5.4 StarCore® Environment Panel Options (*continued*)

Option	Description
Memory Model	Use this listbox to select the memory model for the build tools to use. The options are: • 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. Runtime library calls are made the same way as for the huge model. • Huge Memory Model Absolute addresses do not fit in 1MB.
Display generated command lines in message window	Check this box to instruct the IDE to display the command-line strings it passes to the build tools. The IDE displays command lines in the **Errors and Warnings** window.
Generate relative paths on command-line when possible	Check this box to instruct the IDE to put project-relative paths in the command lines it passes to the tools. Enabling this option eliminates OS error 87 because it shortens the command-string passed to the tools. If this option is disabled, the IDE puts absolute paths in the command lines it passes to the tools.

Enterprise Linker

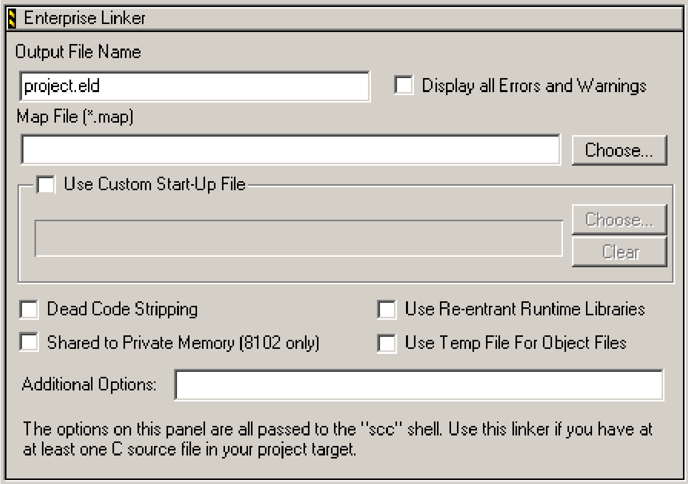
Use the **Enterprise Linker** target settings panel to select options that control the behavior of the Motorola Enterprise Linker.

NOTE The **Enterprise Linker** panel appears in the **Target Settings** window's panel list only if you select Motorola Enterprise Linker from the Linker listbox of the Target Settings panel.

NOTE Select the Enterprise Linker for build targets that include C language source files because this option lets the build tools optimize the build target's output. For build targets that consist entirely of assembly language source files, use the DSP Linker because hand-written assembly language requires no optimization.

Figure 5.5 shows the **Enterprise Linker** target settings panel.

Figure 5.5 Enterprise Linker Target Settings Panel



Enterprise 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 least one C source file in your project target.

Table 5.5 lists and defines each option of the **Enterprise Linker** target settings panel.

Table 5.5 Enterprise Linker Panel Options

Option	Description
Output File Name	Use this text box to specify the name the linker gives to the file it generates. This filename must have the `.eld` extension.
Display all Errors and Warnings	Check this box to instruct the IDE to display all error and warnings messages emitted by the linker.
Map File	Use this text box to type the path and name of the file to which the linker writes memory map information. This filename must have the `.map` extension
Use Custom Start-Up File	Check this box to use a custom start-up file instead of the default start-up file. Checking this checkbox enables a related text box. Use this text box to type the path and name of the custom start-up file.
Dead Code Stripping	Check this box to instruct the linker to strip both unreferenced code and unreferenced data from your program. Enabling this option reduces the memory footprint of a program.
Shared to Private Memory (8102 only)	Check this box to allow calls from shared memory to private memory. If this box is clear, such calls generate error messages.

Table 5.5 Enterprise Linker Panel Options (*continued*)

Option	Description
Use Re-entrant Runtime Libraries	Check this box to let the linker to select the correct thread-safe libraries and start-up code for your target architecture. If checked, the IDE passes `-reentrant` to the scc shell.
Use Temp File For Object Files	Check this box to instruct the IDE to pass object filenames to the linker in a temporary file. Use this option on Windows hosts to avoid exceeding the maximum command-line length imposed by Windows.
Additional Options	Use this text box to specify additional linker command-line options. The IDE passes these options to the scc shell during the link phase. **NOTE:** The IDE passes command-line options to the scc shell exactly as you type them in the Additional Options text box.

DSP Linker

Use the **DSP Linker** target settings panel to select options that control the behavior of the Motorola DSP Linker.

NOTE The **DSP Linker** panel appears in the panel list of the **Target Settings** window only if you select Motorola DSP Linker from the Linker listbox of the Target Settings panel.

NOTE Select the DSP Linker for build targets that consist entirely of assembly language source files because this option does not optimize the build target's output. It is not necessary to optimize hand-written assembly language.

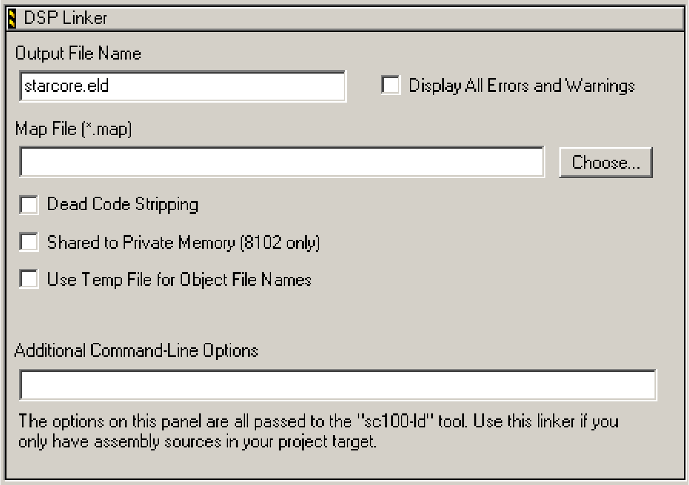
The options of the **DSP Linker** target settings panel are identical to the **Enterprise Linker** panel except that the **DSP Linker** panel does not include these options:

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

Therefore, refer to Table 5.5 for a definition of each option of the **DSP Linker** panel.

Figure 5.6 shows the **DSP Linker** target settings panel.

Figure 5.6 DSP Linker Target Settings Panel



DSP Linker

Output File Name: starcore.eld

☐ Display All Errors and Warnings

Map File (*.map):

☐ 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.

DSP Librarian

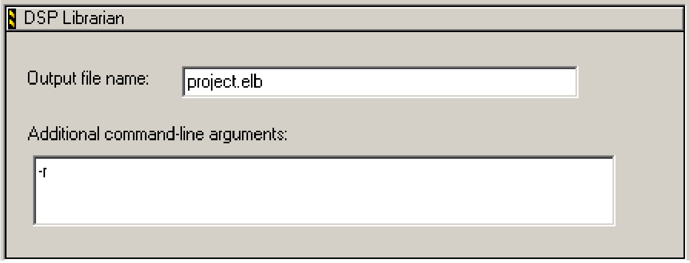
Use the **DSP Librarian** target settings panel to specify the name of the library the build target uses and to pass command-line options to the archiver utility.

NOTE If you select DSP Librarian from the Linker listbox of the Target Settings panel, the archiver utility is invoked each time you make the build target.

NOTE The **DSP Librarian** panel appears in the panel list only if you select DSP Librarian from the Linker listbox of the Target Settings panel.

Figure 5.7 shows the **DSP Librarian** target settings panel.

Figure 5.7 DSP Librarian Target Settings Panel



DSP Librarian

Output file name: project.elb

Additional command-line arguments: -r

Table 5.6 lists and defines each option of the **DSP Librarian** target settings panel.

Table 5.6 DSP Librarian Panel Options

Option	Description
Output file name	Use this text box to specify the filename of the library for the build target to use. A library filename uses the `.elb` extension.
Additional command-line arguments	Use this text box to supply additional command-line arguments. The the IDE passes these arguments to the archiver utility. See Archiver Utility for a list of archiver command-line options.

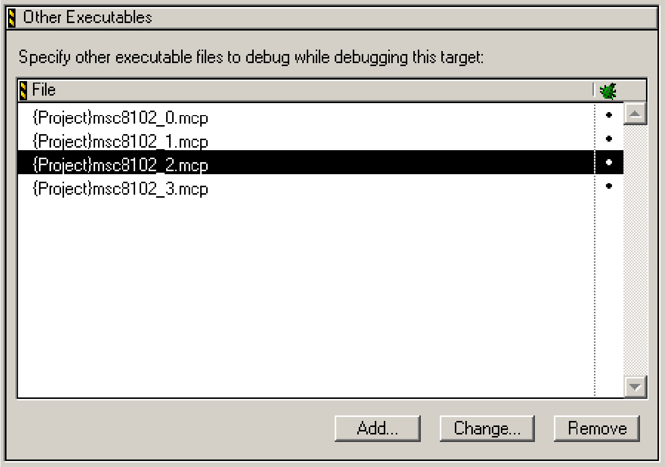
Other Executables

Use the **Other Executables** target settings panel to list other projects and files for the debugger to use in addition to the executable generated by the current build target.

NOTE To use the multi-core debugging feature, add the path and name of the CodeWarrior *project* for each core to be debugged to the Other Executables panel of the primary CodeWarrior project.

Figure 5.8 shows the **Other Executables** target settings panel.

Figure 5.8 Other Executables Target Settings Panel



Other Executables

Specify other executable files to debug while debugging this target:

File	
{Project}msc8102_0.mcp	•
{Project}msc8102_1.mcp	•
{Project}msc8102_2.mcp	•
{Project}msc8102_3.mcp	•

Table 5.7 lists and defines each option of the **Other Executables** target settings panel.

Table 5.7 Other Executables Panel Options

Option	Description
File	Listbox that displays the list of files and projects that the debugger uses during each debug session.
Add	Displays the **Debug Additional Executable** dialog box. Use this dialog box to specify the path and name of a file or project for the debugger to use in addition to the executable generated by the primary build target. Click **OK** to add the file to the File listbox.
Change	Displays the **Debug Additional Executable** dialog box. The fields of the dialog box display the settings for the entry currently selected in the File listbox. Change this information as required and click **OK** to update the information for the currently selected entry.
Remove	Removes the entry currently selected in the File listbox

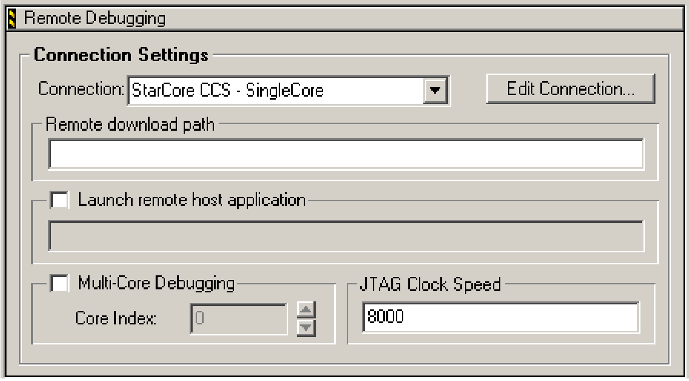
Remote Debugging

Use the **Remote Debugging** 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 Guide* for documentation of the **Remote Connections** preference panel.

Figure 5.9 shows the **Remote Debugging** target settings panel.

Figure 5.9 Remote Debugging Target Settings Panel


Remote Debugging

Connection Settings

Connection: StarCore CCS - SingleCore

Edit Connection...

Remote download path

□ Launch remote host application

□ Multi-Core Debugging

Core Index: 0

JTAG Clock Speed

8000

Table 5.8 lists and defines each option of the **Remote Debugging** target settings panel.

Table 5.8 Remote Debugging Panel Options

Option	Description
Connection	Use this listbox to select the remote connection for this build target to use.
Edit Connection	Click the **Edit Connection** button to display a dialog box with which you can change the configuration of the selected remote connection. **NOTE:** A remote connection has just one definition; as a result, if you change a remote connection's definition in the **Remote Debugging** targets settings panel, you have changed it *everywhere* this connection is used.
Remote download path	Use this text box to specify the path on the host workstation in which a program running on the target can read and write files.
Launch remote host application	Check this box to configure a build target to launch a host application on the target hardware at the start of each debug session. If you check this box, the related text box activates. Use this text box to specify the path to and name of the host application for the build target to use.
Multi-Core Debugging	Check this box to configure a build target for multi-core debugging. If you check this box, the Core Index text box activates. (See below.)

Table 5.8 Remote Debugging Panel Options (*continued*)

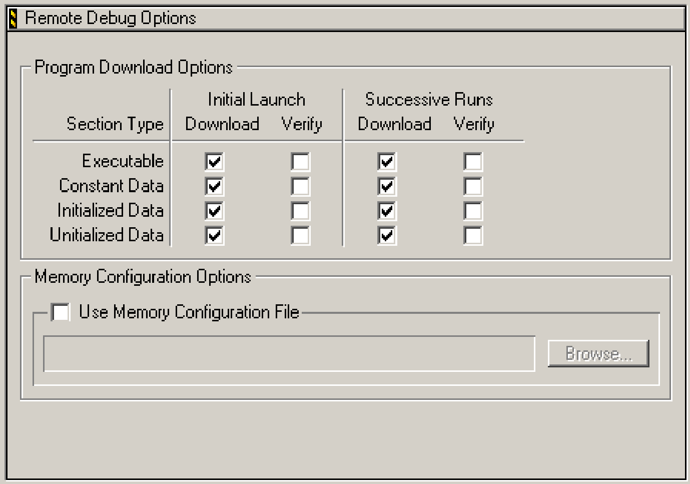
Option	Description
Core Index	Use this text box to specify the index of the core on which a build target's binary is loaded. This is the core with which the the debugger interacts. For most debuggers, 0 is the index of the first core, 1 is the index of the second core, etc.
JTAG Clock Speed	Use this text box to specify the clock speed (in MHz) of the connection between your workstation and the JTAG header of the target hardware.

Remote Debug Options

Use the **Remote Debug Options** panel to tell the debugger which parts of your program to download to the target, when to download these parts, and whether to verify them.

Figure 5.10 shows the **Remote Debug Options** target settings panel.

Figure 5.10 Remote Debug Options Target Settings Panel



Remote Debug Options

Program Download Options

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

Memory Configuration Options

☐ Use Memory Configuration File

Table 5.9 lists and defines each option of the **Remote Debug Options** settings panel.

Table 5.9 Remote Debug Options Panel Options

Option	Description			
Program Download Options	Use the options of this group to define which parts of your program are downloaded to the target, when these parts are downloaded, and whether each part is verified after download.			
Section Type	Initial Launch		Successive Runs	
	Download	Verify	Download	Verify
• Executable	Download executable sections the first time program is debugged.	Verify executable sections the first time program is debugged.	Download executable sections if debug session restarted.	Verify executable sections if debug session restarted.
• Constant Data	Download constant data the first time program is debugged.	Verify constant data the first time program is debugged.	Download constant data if debug session restarted.	Verify constant data if debug session restarted.
• Initialized Data	Download initialized data the first time program is debugged.	Verify initialized data the first time program is debugged.	Download initialized data if debug session restarted.	Verify initialized data if debug session restarted.
• Uninitialized Data	Download uninitialized data the first time program is debugged.	Verify uninitialized data the first time program is debugged.	Download uninitialized data if debug session restarted.	Verify uninitialized data if debug session restarted.
Memory Configuration File	Check this box to instruct the debugger to use a memory configuration file. Click **Browse** to display a dialog box you can use to select a configuration file.			

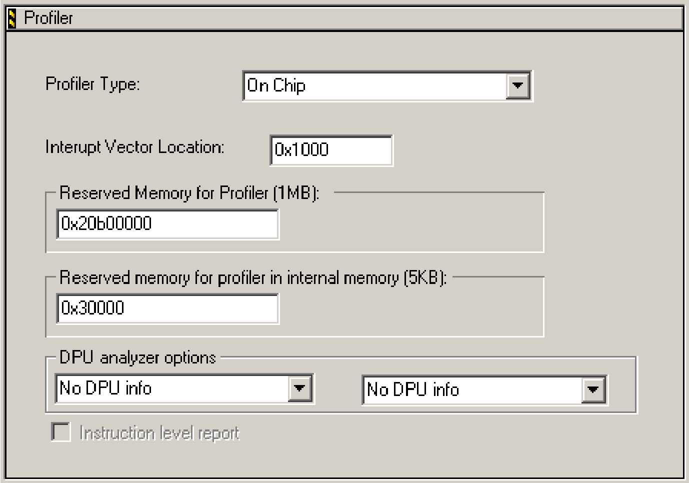
Profiler

Use the **Profiler** target settings panel to configure the CodeWarrior profiler so it can interact with your target and collect information about the program running on this target.

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

Figure 5.11 shows the **Profiler** target settings panel.

Figure 5.11 Profiler Target Settings Panel



Profiler Type: On Chip

Interupt Vector Location: 0x1000

Reserved Memory for Profiler (1MB): 0x20b00000

Reserved memory for profiler in internal memory (5KB): 0x30000

DPU analyzer options: No DPU info / No DPU info

□ Instruction level report

Table 5.10 lists and defines each option of the **Profiler** target settings panel.

Table 5.10 Profiler Panel Options

Option	Description
Profiler Type	Use this listbox to select the type of profiler you are using.
Interrupt Vector Location	Use this text box to specify the address (in hexadecimal) of the vector to the interrupt service routine used by the on chip profiler.
Reserved Memory for Profiler (1MB)	Use this text box to specify the base address (in hexadecimal) of a 1MB buffer in external memory for the on chip profiler to use.
Reserved memory for profiler in internal memory (5KB)	Use this text box to specify the base address (in hexadecimal) of a 5KB buffer of internal memory for the on chip profiler to use.

Table 5.10 Profiler Panel Options (*continued*)

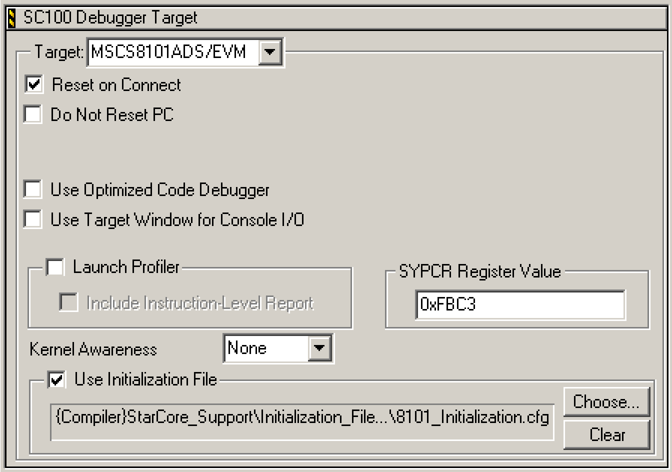
Option	Description
DPU Analyzer Options	Use the leftmost listbox to select the first parameter for the profiler to check during a profile session. Use the rightmost listbox to select the second parameter for the profiler to check during a profile session. **NOTE:** The profiler uses these options apply only if you select a Platform 2002 target from the Target listbox of the SC100 Debugger Target panel.
Instruction Level Report	Check this box to instruct the profiler to perform an instruction-level trace. Leave this box clear to instruct the profiler to perform a change-of-flow trace. **NOTE:** This item activates only if you select On Host from the Profiler Type listbox.

SC100 Debugger Target

Use the **SC100 Debugger Target** panel to select the simulator or device on which you will debug the binary produced by a build target. In addition, use the panel to control how the CodeWarrior debugger interacts with the selected device and how the debugger behaves at startup and during a debug session.

Figure 5.12 shows the **SC100 Debugger Target** target settings panel.

Figure 5.12 SC100 Debugger Target Settings Panel



SC100 Debugger Target

Target: MSC8101ADS/EVM

☑ Reset on Connect

□ Do Not Reset PC

□ Use Optimized Code Debugger

□ Use Target Window for Console I/O

□ Launch Profiler

□ Include Instruction-Level Report

SYPCR Register Value: 0xFBC3

Kernel Awareness: None

☑ Use Initialization File

{Compiler}StarCore_Support\Initialization_File...\8101_Initialization.cfg

Table 5.11 lists and defines each option of the **SC100 Debugger Target** settings panel.

Table 5.11 SC100 Debugger Target Panel Options

Option	Description
Target	Use this listbox to select the simulator or target hardware on which you will debug the binary produced by a build target.
Simulator Options	Use this listbox to select the message types that you want the selected simulator to generate. Use **Ctrl** and **Shift** to select multiple message types. **NOTE:** This listbox appears only if you select SC100 CCS Simulator or SC1000 LLC Simulator from the Target listbox.
Reset on Connect	Check this box to instruct the debugger to reset the target each time you download the program for debugging. If you are using a JTAG chain, all boards are reset.
Stop After Error	Check this box to instruct the debugger to stop after the first error message.
Do Not Reset PC	Check this box to instruct the debugger to preserve the program counter value when you restart a debug session.
Dynamic Error Checking	Check this box to instruct the debugger to check for dynamic errors. Dynamic errors are violations of the StarCore architecture programming rules that occur at runtime.
Enable ICache Performance Tool	Check this box to use the ICache Performance Tool to analyze an MSC8102 binary. **NOTE:** This option appears only if the selected target is MSC8102ADS or the MSC8102 Simulator. **NOTE:** This option and the Launch Profiler option are mutually exclusive.
Load Symbolics Only	Check this box to instruct the debugger to download just a program’s symbolic debug information to the target. This option is useful if: • You are debugging a program that is in ROM. • You repeatedly debug the same program. In this situation, this feature saves time because it skips the (sometimes lengthy) download of code that is already on the target.
Use Optimized Code Debugger	Check this box to debug code that has been optimized. See Debugging Optimized Code for instructions that explain how to use this debugger feature.

Table 5.11 SC100 Debugger Target Panel Options (*continued*)

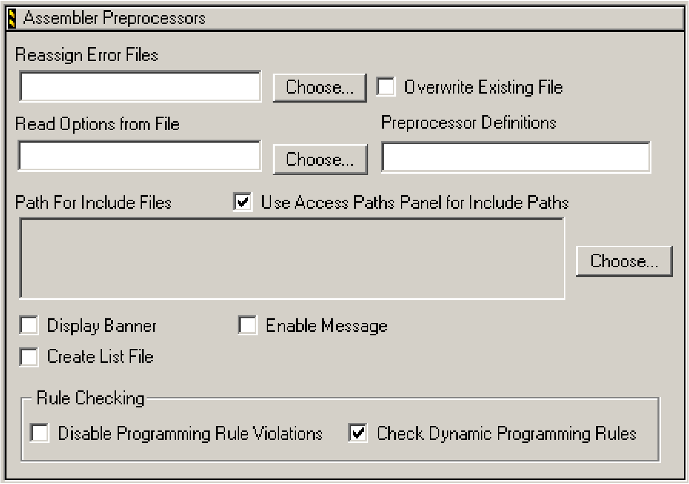
Option	Description
Use Target Window for Console I/O	Check this box to instruct the debugger to display output generated by the target in a separate console window. If you do not check this box, the debugger displays all output in the same console window.
Launch Profiler	Check this box to instruct the debugger to launch the profiler at the start of a debug session. **NOTE:** If you are debugging multiple targets, select this checkbox for each target to be profiled. **NOTE:** This option and the Enable ICache Performance Tool option are mutually exclusive.
Include Instruction-Level Report	Check this box to instruct the Profiler to create an instruction count and parallelism report. **NOTE:** this feature increases Profiler execution time.
SYPCR Register Value	Use this text box to specify the value for the debugger to write to the System Protection Control Register (SYPCR) before reset. The default value is `0xFBC3000` **NOTE:** Available for just the MSC8101ADS/EVM target.
Kernel Awareness	Use this listbox to select the real-time operating system (RTOS) you are running on the target device. The debugger can display kernel objects for the selected RTOS. **NOTE:** Select None if you are not using an RTOS.
Use Initialization File	Check this box to instruct the debugger to use an initialization file at the start of each debug session. Use the **Choose** button to select the desired 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. Use initialization file commands to write values to various registers, core registers, and memory locations.

Assembler Preprocessors

Use the **Assembler Preprocessors** target settings panel to define the directories in which the assembler looks for files, how the assembler handles these files, and to pass the model number and revision of the processor you are using to the assembler.

Figure 5.13 shows the **Assembler Preprocessors** target settings panel.

Figure 5.13 Assembler Preprocessors Target Settings Panel



Assembler Preprocessors

Reassign Error Files
Choose... □ Overwrite Existing File

Read Options from File
Choose...

Preprocessor Definitions

Path For Include Files ☑ Use Access Paths Panel for Include Paths
Choose...

□ Display Banner □ Enable Message
□ Create List File

Rule Checking
□ Disable Programming Rule Violations ☑ Check Dynamic Programming Rules

Table 5.12 lists and defines each option of the **Assembler Preprocessors** settings panel.

Table 5.12 Assembler Preprocessors Panel Options

Option	Description
Reassign Error Files	Use this text box to specify the path and name of a file for the assembler to use in place of the default error file (`errfil`). **NOTE:** If you do not check the Overwrite Existing File checkbox (see below), the assembler appends information to file specified in this text box rather than overwriting the file.
Overwrite Existing File	Check this box to instruct the assembler to overwrite the file named in the Reassign Error Files checkbox if this file exists.
Read Options from File	Use this text box to specify the name of a file that contains command-line options for the assembler to use.
Preprocessor Definitions	Use this text box to enter substitution strings that the assembler applies to all the assembly language modules in the build target. Pass just the string portion of a substitution string: the IDE prepends the `-d` token. Further, separate each substitution string with a comma. For example: `opt1 x, opt2 y` produces the command line: `-dopt1 x -dopt2 y` **NOTE:** This option is similar to the `DEFINE` directive, but applies to all assembly language modules in a build target.

Table 5.12 Assembler Preprocessors Panel Options (*continued*)

Option	Description
Use Access Paths Panel for Include Paths	Check this box to instruct the assembler to use the user paths defined in the **Access Paths** target settings panel instead of the paths specified in the Path for Include Files text box.
Path For Include Files	Use this text box to define a list of search paths for the assembler to use when searching for include files. Separate each path with a comma. You can specify absolute or relative paths. The assembler first looks for an include file in the current directory or the directory specified in the `INCLUDE` directive (if this directive is used). If the file is not found, the assembler next appends the string specified in the `INCLUDE` directive to the first/next path entered in Path For Include Files text box and again looks for the file. The assembler continues this process until it finds the include file or reaches the end of the paths in the Path For Include Files text box. **NOTE:** The assembler issues an error message if a header file is in a different directory from the referencing source file (and sometimes if a header file is in the same directory as the referencing source file). If you get a message such as this: `Could not open source file myfile.h` you must add the path on which `myfile.h` resides to the Path for Include Files text box.
Processor	This option is unused. Use the `-arch` command-line option instead.
Revision	This option is unused. Use the `-arch` command-line option instead.
Display Banner	Check this box to instruct the assembler to display banner information. **NOTE:** This option has no effect on hosts where the banner is not displayed by default.
Enable Message	Check this box to instruct the assembler to report the progress of the assembly process (for example, the beginning of each pass and the opening and closing of input files) to the standard error output stream. The displayed information helps you ensure that assembly is proceeding normally.
Create List File	Check this box to instruct the assembler to create a list file named `lstfil.lst`.

Table 5.12 Assembler Preprocessors Panel Options (*continued*)

Option	Description
Disable Programming Rule Violations	Check this box to instruct the assembler to skip all checks for violations of StarCore DSP programming rules, that is, to skip checks of both the static rules and the dynamic rules. Leave the box clear to instruct the assembler to check for violations of just the static programming rules. **NOTE:** Checking this box causes the IDE to pass the `-u all` option to the assembler. **NOTE:** Checking for static rule violations is the default behavior if you invoke the assembler from the command line.
Check Dynamic Programming Rules	Check this box to instruct the assembler to report violations of the just the dynamic StarCore DSP programming rules. Leave the box clear to instruct the assembler to skip checks for violations of the dynamic rules. **NOTE:** Checking this box causes the IDE to pass the `-s all` option to the assembler. **NOTE:** Skipping checks for dynamic rule violations is the default behavior if you invoke the assembler from the command line.

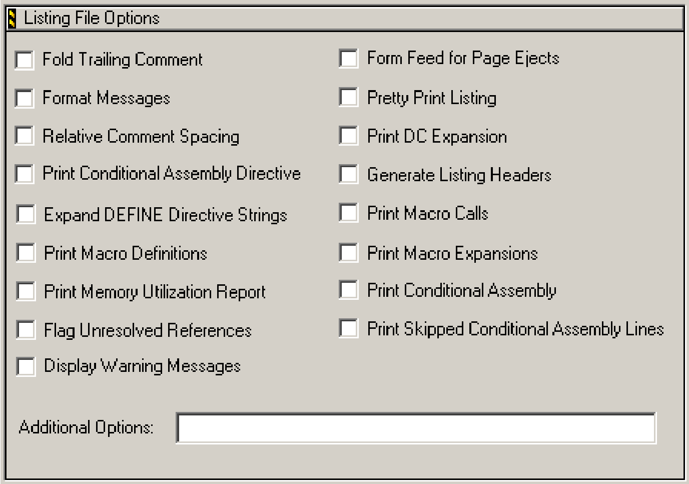
Listing File Options

Use the **Listing File Options** target settings panel to define how the assembler formats the listing file it generates. In addition, use this settings panel to pass command-line options to the assembler.

NOTE Use the Additional Options text box of the **Listing File Options** panel for 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 5.14 shows the **Listing File Options** target settings panel.

Figure 5.14 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 5.13 lists and defines each option of the **Listing File Options** settings panel.

Table 5.13 Listing File Options Panel Options

Option	Description
Fold Trailing Comment	Check this box to instruct the assembler to fold a comment that trails a source code statement underneath the statement, aligned with the opcode field. This setting corresponds to the `FC` option of the `OPT` directive and to the `-ofc` command-line option.
Form Feed for Page Ejects	Check this box to instruct the assembler to insert form feeds into the listing file. Each form feed causes a printer to to perform a page eject. This setting corresponds to the `FF` option of the `OPT` directive and to the `-off` command-line option.
Format Messages	Check this box to instruct the assembler to insert format messages in the listing file such that the message text is aligned and broken at word boundaries. This setting corresponds to the `FM` option of the `OPT` directive and to the `-ofm` command-line option.

Table 5.13 Listing File Options Panel Options (*continued*)

Option	Description
Pretty Print Listing	Check this box to instruct the assembler to align 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.
Relative Comment Spacing	Check this box to instruct the assembler to use relative comment spacing in the listing file. If relative comment spacing is enabled, the position of comments in the listing file floats. This setting corresponds to the `RC` option of the `OPT` directive and to the `-orc` command-line option.
Print DC Expansion	Check this box to instruct the assembler to print DC expansions in the listing file. This setting corresponds to the `CEX` option of the `OPT` directive and to the `-ocex` command-line option.
Print Conditional Assembly Directive	Check this box to instruct the assembler to print 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.
Generate Listing Headers	Check this box to instruct the assembler to generate 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.
Expand DEFINE Directive Strings	Check this box to instruct the assembler to print 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.
Print Macro Calls	Check this box to instruct the assembler to print macro calls in the listing file. This setting corresponds to the `MC` option of the `OPT` directive and to the `-omc` command-line option.
Print Macro Definitions	Check this box to instruct the assembler to print macro definitions in the listing file. This setting corresponds to the `MD` option of the `OPT` directive and to the `-omd` command-line option.

Table 5.13 Listing File Options Panel Options (*continued*)

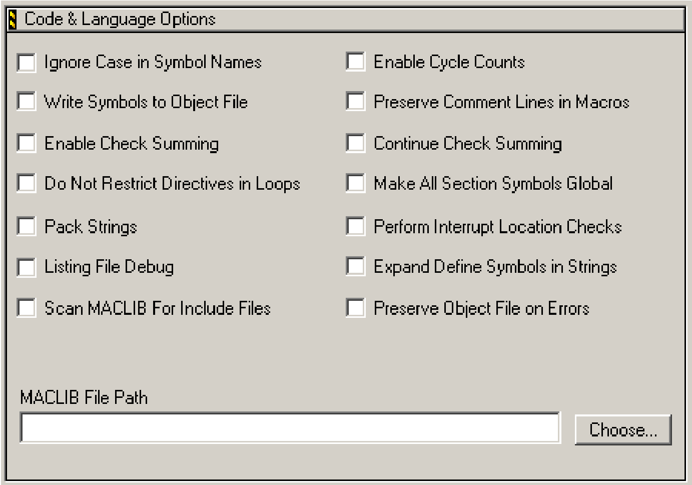
Option	Description
Print Macro Expansions	Check this box to instruct the assembler to print macro expansions in the listing file. This setting corresponds to the `MEX` option of the `OPT` directive and to the `-omex` command-line option.
Print Memory Utilization Report	Check this box to instruct the assembler to put 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.
Print Conditional Assembly	Check this box to instruct the assembler to include 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.
Flag Unresolved References	Check this box to instruct the assembler to generate a warning at assembly-time for each unresolved external reference **NOTE:** valid in relocatable mode only. This setting corresponds to the `UR` option of the `OPT` directive and to the `-our` command-line option.
Print Skipped Conditional Assembly Lines	Check this box to instruct the assembler to include assembly language statements skipped due to conditional assembly in the listing file. This setting corresponds to the `U` option of the `OPT` directive and to the `-ou` command-line option.
Display Warning Messages	Check this box to instruct the assembler to print 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.
Additional Options	Use this text box to pass additional command-line options to the assembler.

Code & Language Options

Use the **Code & Language Options** target settings panel to select code and symbol generation options for the StarCore assembler.

Figure 5.15 shows the **Code & Language Options** target settings panel.

Figure 5.15 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 5.14 lists and defines each option of the **Code & Language Options** settings panel.

Table 5.14 Code & Language Options Panel Options

Option	Description
Ignore Case in Symbol Names	Check this box to instruct the assembler to ignore 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.
Enable Cycle Counts	Check this box to enable the assembler's cycle counter and clear total cycle count features. If you do this, the assembler's listing file shows a cycle count for each instruction entry in the file. **NOTE:** Cycle counts assume a full instruction fetch pipeline and no wait states. This setting corresponds to the `CC` option of the `OPT` directive and to the `-occ` command-line option.
Write Symbols to Object File	Check this box to instruct the assembler to write 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.

Table 5.14 Code & Language Options Panel Options (*continued*)

Option	Description
Preserve Comment Lines in Macros	Check this box to instruct the assembler to preserve comment lines in macros. This setting corresponds to the `CM` option of the `OPT` directive and to the `-ocm` command-line option.
Enable Check Summing	Check this box to instruct the assembler to allow check summing of instruction and data values and to clear the cumulative checksum. **NOTE:** You can use the `@CHK()` function to obtain the checksum value. **NOTE:** A comment line in a macro definition that starts with two consecutive semicolons (`;;`) is never preserved. This setting corresponds to the `CK` option of the `OPT` directive and to the `-ock` command-line option.
Continue Check Summing	Check this box to instruct the assembler to re-enable check summing of instructions and data. **NOTE:** This option does not cause the assembler to clear the cumulative checksum value. This setting corresponds to the `CONTCK` option of the `OPT` directive and to the `-ocontck` command-line option.
Do Not Restrict Directives in Loops	Check this box to instruct the assembler to suppress error messages related to directives that may not be valid in `DO` loops. This setting corresponds to the `DLD` option of the `OPT` directive and to the `-odld` command-line option.
Make All Section Symbols Global	Check this box to create the same effect as explicitly declaring every section `GLOBAL`. **NOTE:** You must check this box before explicitly defining any section in a source file. This setting corresponds to the `GL` option of the `OPT` directive and to the `-ogl` command-line option.
Pack Strings	Check this box to instruct the assembler to pack strings that appear in the `DC` directive. The assembler packs individual bytes of strings into consecutive target words for the length of the string. This setting corresponds to the `PS` option of the `OPT` directive and to the `-ops` command-line option.

Table 5.14 Code & Language Options Panel Options (*continued*)

Option	Description
Perform Interrupt Location Checks	Check this box to instruct the assembler to check 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.
Listing File Debug	Check this box to instruct the assembler to use the source listing as the debug source file instead of using the assembly language source file. **NOTE:** For this option to work, you also check the Create List File box of the Assembler Preprocessors panel. This setting corresponds to the `LDB` option of the `OPT` directive and to the `-oldb` command-line option.
Expand Define Symbols in Strings	Check this box to instruct the assembler to expand `DEFINE` symbols in strings. This setting corresponds to the `DEX` option of the `OPT` directive and to the `-odex` command-line option.
Scan MACLIB for Include Files	Check this box to instruct the assembler to search the `MACLIB` directory paths for include files in addition to the `INCLUDE` directory or the paths defined in the Path For Include Files option of the Assembler Preprocessors panel. in addition to the usual locations. This setting corresponds to the `MI` option of the `OPT` directive and to the `-omi` command-line option.
MACLIB File Path	Use this text box to specify the path to the directory that contains macro definitions. This option corresponds to the `MACLIB` directive.
Preserve Object File on Errors	Check this box to instruct the assembler to preserve object files produced during assembler if assembly errors occur. This setting corresponds to the `SVO` option of the `OPT` directive and to the `-osvo` command-line option.

C Language

Use the **C Language** target settings panel to make settings that tell the compiler the version of the C language you are using.

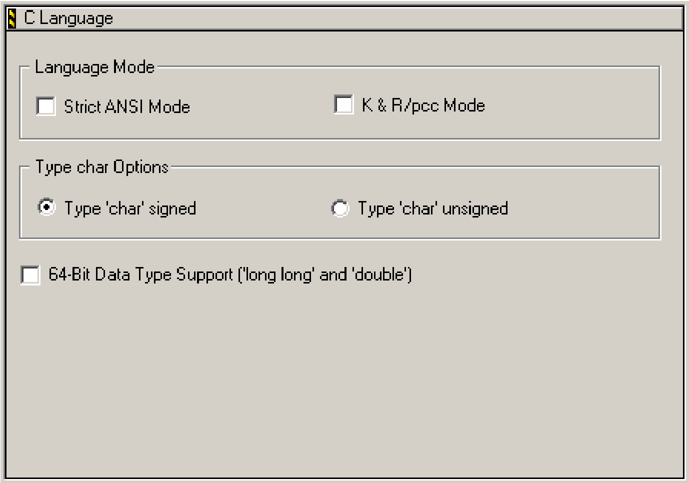
The CodeWarrior C compiler's default mode is ANSI/ISO mode with extensions.

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

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

Figure 5.16 shows the **C Language** target settings panel.

Figure 5.16 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 5.15 lists and defines each option of the **C Language** target settings panel.

Table 5.15 C Language Panel Options

Option	Description
Strict ANSI Mode	Check this box to instruct the C compiler to apply the rules defined by the ANSI/ISO specification strictly to all input files. The compiler issues a warning for each ANSI/ISO extension it finds. This setting is equivalent to the `-ansi` command-line option.
K & R/pcc Mode	Check this box to instruct the C compiler to apply the Kernighan & Ritchie syntax rules to all input files. This setting is equivalent to the `-kr` command-line option.

Table 5.15 C Language Panel Options (*continued*)

Option	Description
Type char Options	The options of this group instruct the compiler how to interpret `char` data types. • Type 'char' signed — select this radio button to instruct the compiler to assume all `char` data types are signed. This setting is the default. • Type 'char' unsigned — select this radio button to instruct the compiler to assume all `char` data types are unsigned, that is, as if declared `unsigned char`. This setting is equivalent to the `-usc` command-line option.
64-Bit Data Type Support ('long long' and 'double')	Check this box to enable the C compiler's support for the 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.

Enterprise Compiler

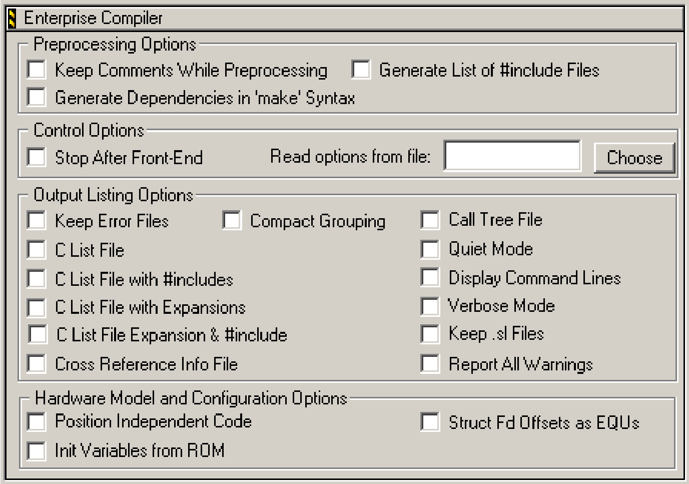
Use the **Enterprise Compiler** target settings panel to define the behavior of the Metrowerks Enterprise C compiler for the current build target.

Refer to the *Metrowerks Enterprise C Compiler User Guide* for complete documentation of this tool.

NOTE The compiler uses your preprocessing selections only if you select a C source file in the project window and select **Project > Preprocess**. Otherwise, the compiler ignores your preprocessing selections.

Figure 5.17 shows the **Enterprise Compiler** target settings panel.

Figure 5.17 Enterprise Compiler Target Settings Panel



Enterprise Compiler

Preprocessing Options

□ Keep Comments While Preprocessing □ Generate List of #include Files

□ Generate Dependencies in 'make' Syntax

Control Options

□ Stop After Front-End Read options from file: Choose

Output Listing Options

□ 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

Hardware Model and Configuration Options

□ Position Independent Code □ Struct Fd Offsets as EQUs

□ Init Variables from ROM

Table 5.16 lists and defines each option of the **Enterprise Compiler** target settings panel.

Table 5.16 Enterprise Compiler Panel Options

Option	Description
Preprocessor Options	Use the options of this group box to control the behavior of the preprocessor.
• Keep Comments While Preprocessing	Check this box to instruct the preprocessor to include source file comments in its output. Equivalent to the `-C` command-line option.
• Generate List of #include Files	Check this box to instruct the preprocessor to list the full path and name of each include file referenced by the selected source files. This list includes all nested include files. Equivalent to the `-MH` command-line option.
• Generate Dependencies in 'make' Syntax	Check this box to instruct the preprocessor to list (in `MAKE` format) the name of each generated output file along with the full path and name of the source files upon which the output file is dependent. Equivalent to the `-M` command-line option.
Control Options	Use the options of this group box to control the behavior of the compiler and of the shell.

Table 5.16 Enterprise Compiler Panel Options (*continued*)

Option	Description
• Stop After Front-End	Check this box to instruct the compiler to perform just front-end processing on the source files in the current build target. Use this option to verify that your source files meet the essential requirements for processing by the StarCore build tools, in particular, that the files contain no syntax errors. This feature is useful if you are preparing source files for global optimization. Equivalent to the `-cfe` command-line option.
• Read options from file	Use this text box to specify the path and name of a file that contains compiler command-line options. The filename must use the `.opt` extension. The shell treats the options in such a file as if they were passed on the command-line. Each time you invoke the compiler, you can select a file with the set of options that suits your needs. Equivalent to the `-F` *`file`* command-line option. **NOTE:** The IDE does not verify that the options in a command file are valid.
Output Listing Options	Use the options of this group box to control how the compiler formats its listing file, and its error and warning messages.
• Keep Error Files	Check this box to instruct the compiler to keep the intermediate error files it generates instead of displaying the error messages these files contain in the **Errors and Warnings** window. If this option is enabled, the compiler: • Creates an error file for each source file in the build target (whether or not the file contains an error). • Names each error file using the root filename of the related source file followed by the suffix `.err`. E.g., `main.c` -> `main.err` • Puts each error file in the project directory. Equivalent to the `-de` command-line option.

Table 5.16 Enterprise Compiler Panel Options (*continued*)

Option	Description
• Compact Grouping	Check this box to instruct the compiler to place each instruction of a variable length execution set (VLES) on the same line of the intermediate assembly language source file the compiler generates instead of placing each VLES instruction on a separate line. Equivalent to the `-Xllt -mlp` command-line option. **NOTE:** this option does nothing unless you also check the Keep .sl Files box (discussed below).
• Call Tree File	Check this box to instruct the compiler to create a postscript file that contains a call tree graph. You can print this file on a postscript printer. Equivalent to the `-dc 4` command-line option.
• C List File	Check this box to instruct the compiler to create a list file for each C source file in the build target. Each list file contains 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. Equivalent to the `-dL` command-line option.
• C List File with #includes	Check this box to instruct the compiler to create a list file for each C source file in the build target. Each list file contains the contents of the related source file along with the contents of each included file. The content of each include file is inserted 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. Equivalent to the `-dL1` command-line option.
• C List File with Expansions	Check this box to instruct the compiler to create a list file for each C source file in the build target. Each list file contains the contents of the related source file and expansions of, for example, macros, line splices, and trigraphs. Each list filename consists of the root filename of the related source file followed by the `.lis` extension. Equivalent to the `-dL2` command-line option.

Table 5.16 Enterprise Compiler Panel Options (*continued*)

Option	Description
• C List File Expansion & #include	Check this box to instruct the compiler to create a list file for each C source file in the build target. Each list file contains the contents of the related source file, the contents of each included file, and expansions of, for example, macros, line splices, and trigraphs. The content of each include file is inserted 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. Equivalent to the `-dL3` command-line option.
• Cross Reference Info File	Check this box to instruct the compiler to create a cross-reference file for each C source file in the build target. Each cross-reference filename consists of the root filename of the related source file followed by the `.xrf` extension. Equivalent to the `-dx` command-line option.
• Quiet Mode	Check this box to instruct the IDE to display just error messages emitted by the compiler. Warning and informational messages are suppressed. Equivalent to the `-q` command-line option.
• Display Command Lines	Check this box to instruct the IDE to display the command lines it will pass to each build tool without actually invoking these tools. Use this option to verify that each command line includes the options you expect (based on the selections made in each target settings panel). If this option is enabled no object files are created. Equivalent to the `-n` command-line option.
• Verbose Mode	Check this box to instruct the IDE to display each command line it passes to the shell along with all progress, error, warning, and informational messages emitted by these tools. Equivalent to the `-v` command-line option.

Table 5.16 Enterprise Compiler Panel Options (*continued*)

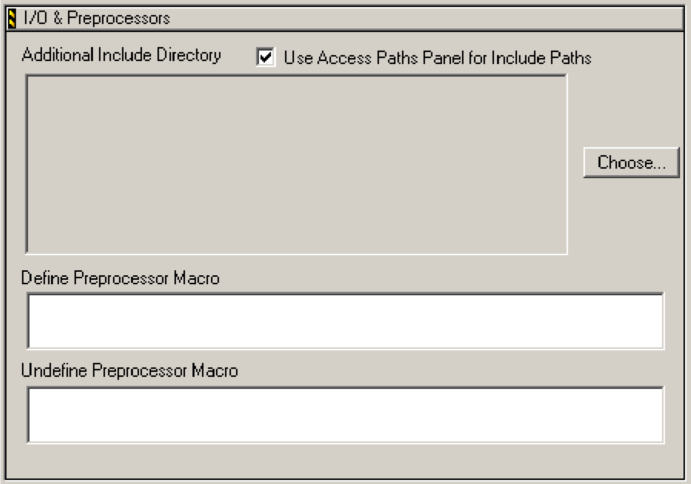
Option	Description
• Keep .sl Files	Check this box to instruct the compiler to keep the intermediate assembly language source files (`.sl` files) it creates instead of deleting them. The compiler generates one `.sl` file for each C source file in the build target. Equivalent to the `-s` command-line option.
• Report All Warnings	Check this box to instruct the compiler to report all possible warnings to the IDE for display in the **Errors and Warnings** window. Equivalent to the `-Wall` command-line option.
Hardware Model and Configuration Options	Use the options of this group box to control how the compiler structures the code it generates.
• Position Independent Code	Check this box to instruct the compiler to generate position independent code. Equivalent to the `-pic` command-line option.
• Init Variables from ROM	Check this box to instruct the compiler to place the data used to initialize your program's global variables in a separate section that can be manipulated at link-time and load-time. During development, you typically leave this box unchecked because a separate loader program handles initializing your programs globals. Once development is complete, you typically check this box so you can put the segment that contains the data used to initialize your globals in ROM. Equivalent to the `-mrom` command-line option.
• Struct Fd Offsets as EQUs	Check this box to instruct the compiler to include the offsets of C data structure field definitions in each intermediate assembly language source file created. Equivalent to the `-do` command-line option.

I/O & Preprocessors

Use the **I/O & Preprocessors** target settings panel to specify additional directories in which the compiler will search when looking for include files. In addition, use this panel to define and undefine preprocessor macros.

Figure 5.18 shows the **I/O & Preprocessors** target settings panel.

Figure 5.18 I/O & Preprocessors Target Settings Panel



I/O & Preprocessors

Additional Include Directory ☑ Use Access Paths Panel for Include Paths

Define Preprocessor Macro

Undefine Preprocessor Macro

Table 5.17 lists and defines each option of the **I/O & Preprocessors** target settings panel.

Table 5.17 I/O & Preprocessors Panel Options

Option	Description
Additional Include Directory	Use this text box to define a list of search paths for the compiler to use when searching for include files. Separate each path with a comma. You can specify absolute or relative paths. Equivalent to the `-I` *`path`* command-line option.
Use Access Paths Panel for Include Paths	Check this box to instruct the compiler to search for include files in the user access paths defined in the **Access Paths** settings panel instead of the paths in the Path for Include Files text box. Equivalent to the `-I` *`path`* command-line option. **NOTE:** On a Windows PC, the maximum command-line length is 32 KB. If you get errors about too many include paths, try removing recursive path definitions from the **Access Paths** target settings panel.

Table 5.17 I/O & Preprocessors Panel Options (*continued*)

Option	Description
Define Preprocessor Macro	Use this text box to define preprocessor macros and, optionally, assign them values. Assign a value using the equal sign (=) with no white space. Separate multiple macro definitions with commas. For example: `EXTENDED_FEATURE=ON, DEBUG_CHECK` Equivalent to the `-D` *`name`*[=*`value`*] command-line option. **NOTE:** If you do not assign a macro a value, the shell assumes the value is 1.
Undefine Preprocessor Macro	Use this text box to undefine preprocessor macros. Separate multiple macro undefinitions with commas. For example: `EXTENDED_FEATURE, DEBUG_CHECK` Equivalent to the `-U` *`name`* command-line option. **NOTE:** The shell processes any Undefine Preprocessor Macro items after its processes all Define Preprocessor Macro items.

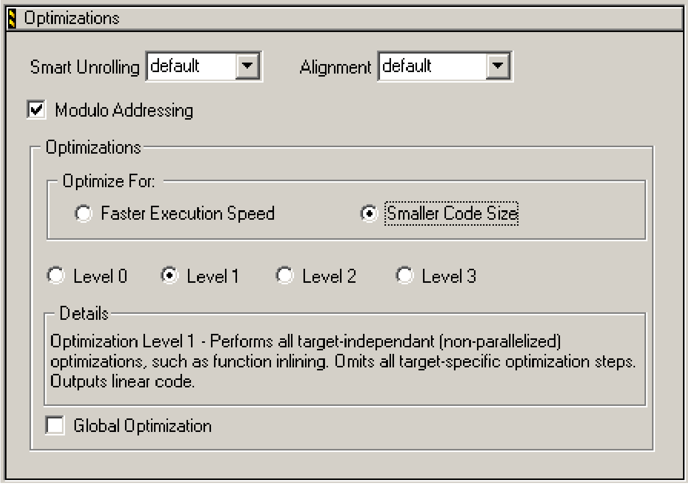
Optimizations

Use the **Optimizations** target settings panel to define and configure the optimizations the C compiler performs.

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

Figure 5.19 shows the **Optimizations** target settings panel.

Figure 5.19 Optimizations Target Settings Panel



Optimizations

Smart Unrolling default ▾   Alignment default ▾

☑ Modulo Addressing

Optimizations

Optimize For:

○ Faster Execution Speed   ● Smaller Code Size

○ Level 0   ● Level 1   ○ Level 2   ○ Level 3

Details

Optimization Level 1 - Performs all target-independant (non-parallelized) optimizations, such as function inlining. Omits all target-specific optimization steps. Outputs linear code.

☐ Global Optimization

Table 5.18 lists and defines each option of the **Optimizations** target settings panel.

Table 5.18 Optimizations Panel Options

Option	Description
Smart Unrolling	Use this listbox to select the *maximum* unrolling factor the compiler will use in the automatic loop unrolling optimization. Select default to let the compiler select the unrolling factor. Equivalent to the `-ulevel` command-line option. **NOTE:** If you select 0, no loops are unrolled. However, if you select 4, loops may be unrolled with a factor of either 2 or 4, depending upon the gain (as determined by the compiler).

Table 5.18 Optimizations Panel Options (*continued*)

Option	Description
Alignment	Use this listbox to select the alignment level the compiler uses. The options are: • default - 0 used for size optimizations - 2 used for 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 Equivalent to the `-align` *`level`* command-line option. **NOTE:** Using a higher alignment constraint usually increases execution speed but can also increase code size.
Modulo Addressing	Check this box to instruct the compiler to use modulo addressing. Equivalent to the `-mod` command-line option.
Optimizations	Use the options of this group box to select the optimizations the compiler performs. The Details area of this group box describes the specific optimizations performed by the selected optimization level.
• Optimize For -- Faster Execution Speed -- Smaller Code Size	Select Faster Execution Speed to instruct the compiler to favor faster execution speed over reducing code size as it applies the selected optimization level. Select Smaller Code Size to instruct the compiler to favor reducing code size over faster execution speed when applying an optimization level. Equivalent to the `-Os` command-line option.
• Level 0	Select to instruct the compiler to perform no optimizations. The compiler generates unoptimized, linear assembly language code. Equivalent to the `-O0` command-line option.
• Level 1	Select to instruct the compiler to perform all target-independent (that is, non-parallelized) optimizations, such as function inlining. The compiler omits all target-specific optimizations and generates linear assembly language code. Equivalent to the `-O1` command-line option.

Table 5.18 Optimizations Panel Options (*continued*)

Option	Description
• Level 2	Select to instruct the compiler to perform all optimizations. The compiler outputs optimized, non-linear assembly language code. Equivalent to the `-O2` command-line option.
• Level 3	Select to instruct the compiler to perform all Level 2 optimizations and then perform global register allocation. Equivalent to the `-O3` command-line option.
• Global Optimization	Check this box to instruct the compiler to apply the selected optimizations across all the files in the build target. Global optimization is the most effective optimization method. If this option is enabled, the compiler creates intermediate files that have the `.obj` file extension. Equivalent to the `-cfe` compiler command-line option followed by the `-Og` link-phase command-line option. **NOTE:** The Global Optimization option is available for all optimization levels except Level 0. **NOTE:** If you select global optimization, the IDE applies a value of -1 for code size and data size to a file that is globally optimized rather than reporting the code and data size in the project window.

Source Folder Mapping

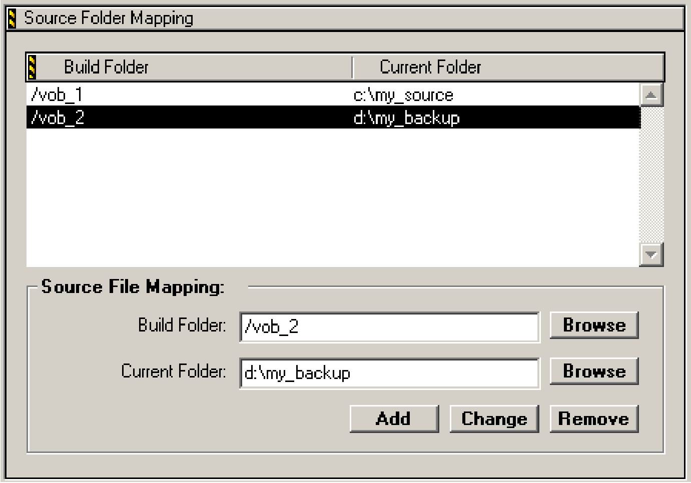
Use the **Source Folder Mapping** target settings panel if you are debugging an executable file that was built in one place, but which is being debugged from another.

The mapping information you supply 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 the source files used to build the file to the new project.
If the IDE cannot find a particular source file, the IDE displays a dialog box that you use to tell the IDE where the missing file is currently. The IDE uses the current location information in conjunction with the debug information in the ELF file to create entries in the **Source Folder Mapping** panel.

Figure 5.20 shows the **Source Folder Mapping** target settings panel.

Figure 5.20 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 5.19 lists and defines each option of the **Source Folder Mapping** settings panel.

Table 5.19 Source Folder Mapping Panel Options

Option	Description

Build Folder	Use this text box to enter the path that contained the executable’s source files when this executable was built. The supplied 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`.
	If the debugger cannot find a file referenced in the executable’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.

Table 5.19 Source Folder Mapping Panel Options (*continued*)

Option	Description

Current Folder	Use the Current Folder text box to enter the path that contains the executable's source files now, that is, at the time of the debug session.
	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 box.
	If the debugger cannot find a file referenced in the executable'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 this button to add a new association to the Source Folder Mapping list.
Change	Click this button to change the association currently selected in the Source Folder Mapping list.
Remove	Click this button to delete the association currently selected in the Source Folder Mapping list.

Passthrough, Hardware

Use the **Passthrough, Hardware** target settings 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 command-line options specified in the Passthrough, Hardware panel are applied to the compilation of the C language source files in a build target (even the options entered 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 5.21 shows the **Passthrough, Hardware** target settings panel.

Figure 5.21 Passthrough, Hardware Target Settings Panel

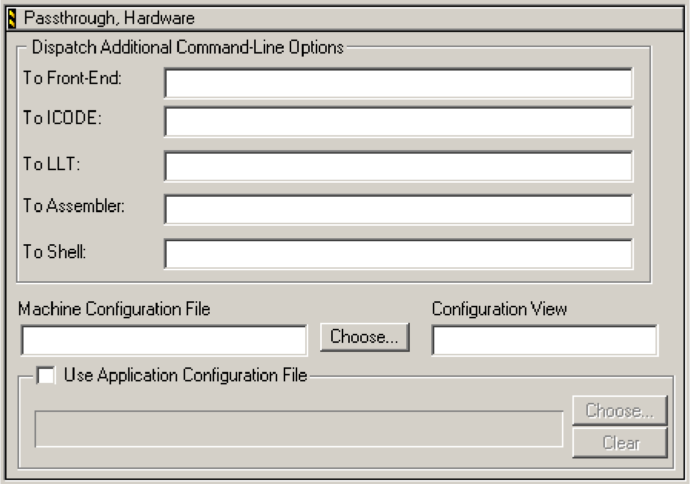


Table 5.20 lists and defines each option of the **Passthrough, Hardware** settings panel.

Table 5.20 Passthrough, Hardware Panel Options

Option	Description
To Front-End	Use this text box to enter command-line options for the shell (scc) to pass to the compiler front-end. Equivalent to the `-Xcfe` options command-line option.
To ICODE	Use this text box to enter command-line options for the shell (scc) to pass to the compiler's high-level optimizer. Equivalent to the `-Xicode` options command-line option.
To LLT	Use this text box to enter command-line options for the shell (scc) to pass to the compiler's low-level optimizer. Equivalent to the `-Xllt` options command-line option.
To Assembler	Use this text box to enter command-line options for the shell program (scc) to pass to the assembler. Equivalent to the `-Xasm` options command-line option.
To Shell	Use this text box to enter command-line options for the IDE to pass to the shell program (scc). The IDE passes the options exactly as you type them and does not check for errors.

Table 5.20 Passthrough, Hardware Panel Options (*continued*)

Option	Description
Machine Configuration File	Use this text box to specify the path and name of a custom machine configuration file for the build target to use. Equivalent to the `-mc` *`filename`* command line option.
Use Application Configuration File	Check this box to instruct the compiler to use a custom application configuration file. Use the **Choose** button to select the desired file. Disable this option to instruct the compiler to use the default application configuration file. Use a custom application configuration file to apply different optimization levels and options to the different files and functions of the build target. Equivalent to the `-ma` *`filename`* command line option. **NOTE:** An application configuration file must have the `.appli` extension.
Configuration View	Use this text box to specify the application configuration file view for the build target to use. Equivalent to the `-view` *`identifier`* command line option. **NOTE:** This option has no effect if you do not specify an application configuration file.

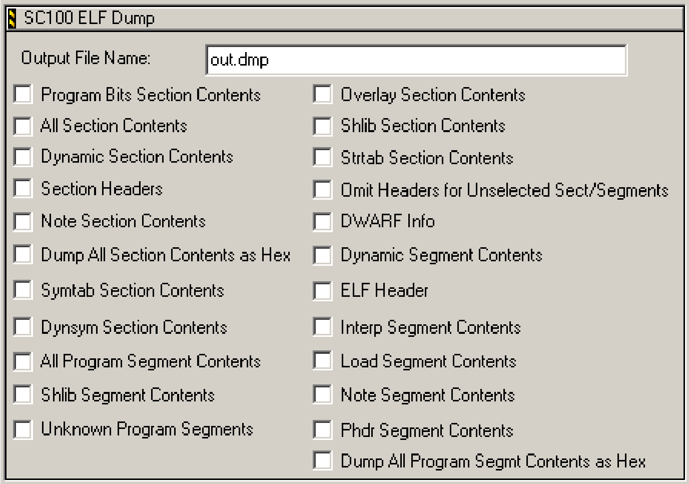
SC100 ELF Dump

Use the **SC100 ELF Dump** target settings panel to define the configure the behavior of the ELF file dump utility.

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

Figure 5.22 shows the **SC100 ELF Dump** target settings panel.

Figure 5.22 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 5.21 lists and defines each option of the **SC100 ELF Dump** target settings panel.

Table 5.21 SC100 ELF Dump Panel Options

Option	Description
Output File Name	Use this text box to enter the path and name of the file to which the ELF file dump utility writes its output. **NOTE:** If you leave this text box empty, the utility writes its output to the **Errors and Warnings** window.
Program Bits Section Contents	Check this option to include the contents all program bits sections in the dump. Equivalent to the `-b` command-line option.
All Section Contents	Check this option to include the contents of all sections in the dump. Equivalent to the `-a` command-line option.
Dynamic Section Contents	Check this option to include the contents of all dynamic sections in the dump. Equivalent to the `-d` command-line option. **NOTE:** This option does not apply to the SC140 core.
Section Headers	Check this option to include the contents of all hash sections in the dump. Equivalent to the `-h` command-line option. **NOTE:** This option does not apply to the SC140 core.

Table 5.21 SC100 ELF Dump Panel Options (*continued*)

Option	Description
Note Section Contents	Check this option to include the contents of all note sections in the dump. Equivalent to the `-n` command-line option.
Dump All Section Contents As Hex	Check this option to instruct the dump utility to write the contents of all sections in hexadecimal. Equivalent to the `-x` command-line option.
Symtab Section Contents	Check this option to include the contents of all symtab sections in the dump. Equivalent to the `-y` command-line option.
Dynsym Section Contents	Check this option to include the contents of all dynsym sections in the dump. Equivalent to the `-z` command-line option. **NOTE:** This option does not apply to the SC140 core.
All Program Segment Contents	Check this option to include the contents of all program segments in the dump. Equivalent to the `-A` command-line option.
Shlib Segment Contents	Check this option to include the contents of all shlib segments in the dump. Equivalent to the `-S` command-line option. **NOTE:** This option does not apply to the SC140 core.
Unknown Program Segments	Check this option to include the contents of all unknown type segments (in hexadecimal) in the dump. Equivalent to the `-U` command-line option.
Overlay Section Contents	Check this option to include the contents of all overlay table sections in the dump. Equivalent to the `-o` command-line option.
Shlib Section Contents	Check this option to include the contents of all shlib sections in the dump. Equivalent to the `-s` command-line option.
Strtab Section Contents	Check this option to include the contents of all strtab sections in the dump. Equivalent to the `-t` command-line option.
Omit Headers for Unselected Sect/Segments	Check this option to limit header information to the specified sections and segments. Equivalent to the `-q` command-line option.

Table 5.21 SC100 ELF Dump Panel Options (*continued*)

Option	Description
DWARF Info	Check this option to Equivalent to the `-g` command-line option.
Dynamic Segment Contents	Check this option to include the contents of all dynamic segments in the dump. Equivalent to the `-D` command-line option. **NOTE:** This option does not apply to the SC140 core.
ELF Header	Check this option to include ELF header information in the dump. This is the default behavior of the dump utility. Equivalent to the `-E` command-line option.
Interp Segment Contents	Check this option to include the contents of all interp segments in the dump. Equivalent to the `-I` command-line option.
Load Segment Contents	Check this option to include the contents of all load segments in the dump. Equivalent to the `-L` command-line option.
Note Segment Contents	Check this option to include the contents of all note segments in the dump. Equivalent to the `-N` command-line option. **NOTE:** This option does not apply to the SC140 core.
Phdr Segment Contents	Check this option to include the contents of all phdr segments in the dump. Equivalent to the `-P` command-line option.
Dump All Program Segmt Contents as Hex	Check this option to instruct the dump utility to write the contents of all segments in hexadecimal. Equivalent to the `-X` command-line option.

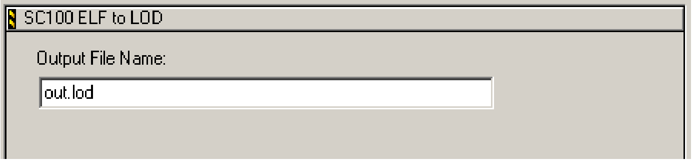
SC100 ELF to LOD

Use the **SC100 ELF to LOD** target settings panel to define the name of the LOD file generated by the elflod utility.

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.

Figure 5.23 shows the **SC100 ELF to LOD** target settings panel.

Figure 5.23 SC100 ELF to LOD Target Settings Panel



SC100 ELF to LOD

Output File Name: out.lod

Table 5.22 lists and defines each option of the **SC100 ELF to LOD** target settings panel.

Table 5.22 SC100 ELF to LOD 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 LOD post-linker writes.

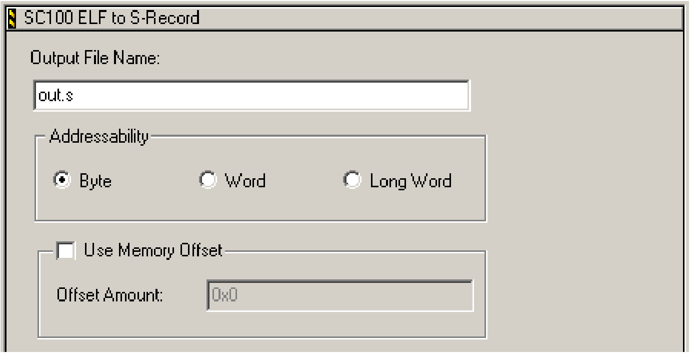
SC100 ELF to S-Record

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

Figure 5.24 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 5.23 lists and defines each option of the **SC100 ELF to S-Record** settings panel.

Table 5.23 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.

6

Debugging StarCore® DSP Programs

This chapter describes the StarCore DSP-specific features of the CodeWarrior™ debugger. The *IDE User Guide* describes standard features of the CodeWarrior debugger.

The sections are:

- Stack Crawl Depth
- Registers Window
- Register Window Formatter
- Register Details Window
- Tips for Debugging Assembly Language
- Cycle Counter in the Simulator
- Debugging a .eld File Without a CodeWarrior Project
- System-Level Connect
- Initialization File
- Kernel Awareness
- Command-Line Debugging
- Manipulating Target Memory
- Save Restore Registers

Stack Crawl Depth

The maximum depth of the stack crawl is 26 stack frames.

Registers Window

The **Registers** window displays the register sets and registers of the target device you are debugging. Using this window, you can see and modify the value of any register.

To display and use the **Registers** window, follow these steps:

1. Open the CodeWarrior project that you want to debug.
2. Select the build target of this project that you want to debug.
3. Select **Project > Debug**

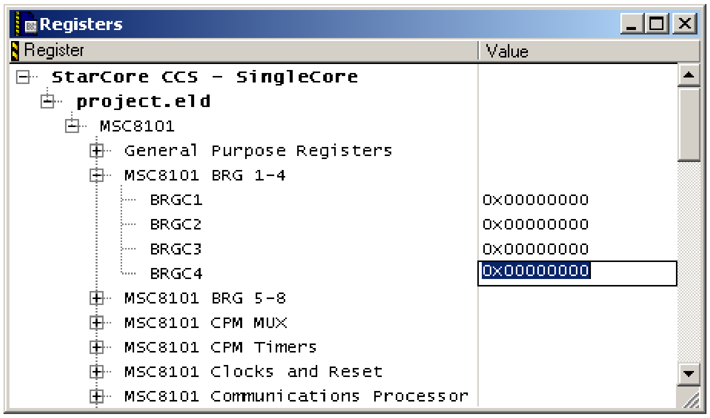
   The debugger downloads your program to the target device and halts execution at the program's entry point.
4. Debug your program as required.
5. Select **View > Registers**

   The debugger displays the **Registers** window. (See Figure 6.1.)

   The window displays register groups and registers in a tree format.

NOTE Before you can display the **Registers** window, you must halt your program.

Figure 6.1 Registers Window



```
⊟ StarCore CCS - SingleCore
   ⊟ project.eld
      ⊟ MSC8101
         ⊞ General Purpose Registers
         ⊟ MSC8101 BRG 1-4
              BRGC1                          0x00000000
              BRGC2                          0x00000000
              BRGC3                          0x00000000
              BRGC4                          0x00000000
         ⊞ MSC8101 BRG 5-8
         ⊞ MSC8101 CPM MUX
         ⊞ MSC8101 CPM Timers
         ⊞ MSC8101 Clocks and Reset
         ⊞ MSC8101 Communications Processor
```

6. To modify a register's value:

   a. Double-click on a register's value text box.

      The value text box activates.

   b. Type the new value to assign to the register (in hexadecimal).

Register Window Formatter

The Register Window Formatter window lets you define the contents and layout of the Registers Window by letting you:

- Select the registers to display in the **Registers** window from the total set of registers available.
- Define the order in which the **Registers** window displays the selected registers.
- Create a different **Registers** window format for each remote connection you use.
- Export register layouts to an XML file.
- Import register layouts from an XML file.

The sections that follow explain how to use the **Register Window Formatter** window.

Selecting Register Sets and Registers for Display

To select the registers that appear in the **Registers** window, follow these steps:

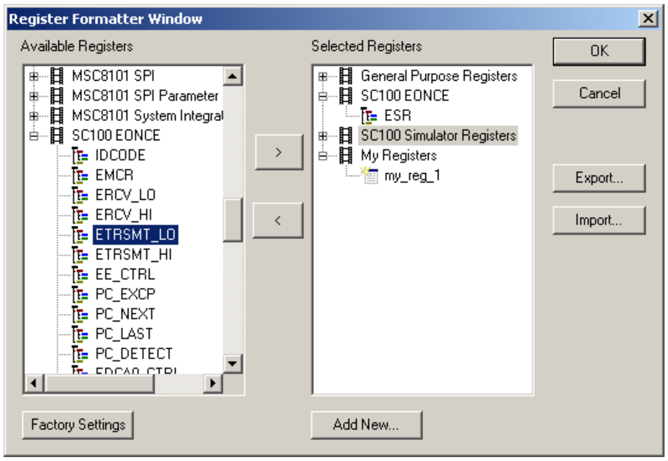
1. Select **Project > Debug**

   The debugger downloads your program to the target device or simulator using the selected remote connection.

2. Select **Debug > Register Window Formatter**

   The **Register Window Formatter** window appears. (See Figure 6.2.)

Figure 6.2 Register Window Formatter



3. To add an entire register group:
   a. From the Available Registers listbox, select a register group that you want to appear in the **Registers** window each time you use the current remote connection.
   b. Click **>**

      The selected register group moves from the Available Registers listbox to the Selected Registers listbox.
4. To add one register from a register group:
   a. In the Available Registers listbox, expand 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 selected register (along with the name of the group this register is in) moves from the Available Registers listbox to the Selected Registers listbox.
5. Click **OK**

   The **Register Window Formatter** window saves your configuration and closes.
6. Select **View > Registers**

   The **Registers** window appears and displays the selected register sets.

NOTE If the **Registers** window is open when you modify its layout using the **Register Window Formatter** window, you must close and reopen the Registers window to see your new register configuration.

Adding a Register to a Register Group

To add a register to a register group, follow these steps:

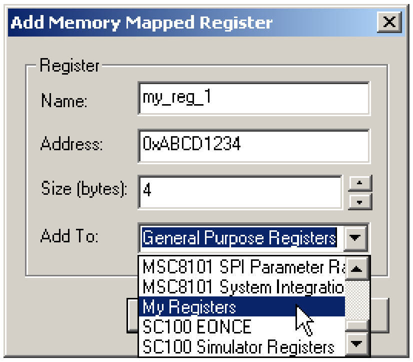
1. Select **Project > Debug**

   The debugger downloads your program to the target device or simulator using the selected remote connection.
2. Select **Debug > Register Window Formatter**

   The **Register Window Formatter** window appears. (See Figure 6.2.)
3. Click **Add New**

   The **Add Memory Mapped Register** dialog box appears. (See Figure 6.3.)

Figure 6.3 Add Memory Mapped Register Dialog Box



Add Memory Mapped Register

Register

Name: my_reg_1

Address: 0xABCD1234

Size (bytes): 4

Add To: General Purpose Registers

4. In the Name text box, type the name of the new register.
5. In the Address text box, type the absolute address of the new register (in hexadecimal).
6. In the Size (bytes) text box, type the width of the register (in decimal).
7. From the Add To listbox, select the register group to which to add the register.

NOTE To create a new register group, type a new group name in the Add To listbox.

8. Click **OK**

   The **Add Memory Mapped Register** dialog box closes; the **Registers Window Formatter** window adds the new register to the specified register group.

Register Details Window

You can use the **Register Details** window to view the values of StarCore DSP registers and see descriptions of these registers.

XML files contain the register descriptions.

The XML register description files are in these paths:

Windows

installDir`\bin\Plugins\support\Registers`

Solaris

installDir/CodeWarrior_ver_dir/
`CodeWarrior_IDE/CodeWarrior_Plugins/support/Registers`

By default, the CodeWarrior IDE searches all folders in the `Registers` directory when searching for a register description file. Register description files must end with the extension `.xml`.

The minimum resolution of bitfield descriptions is limited to two 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

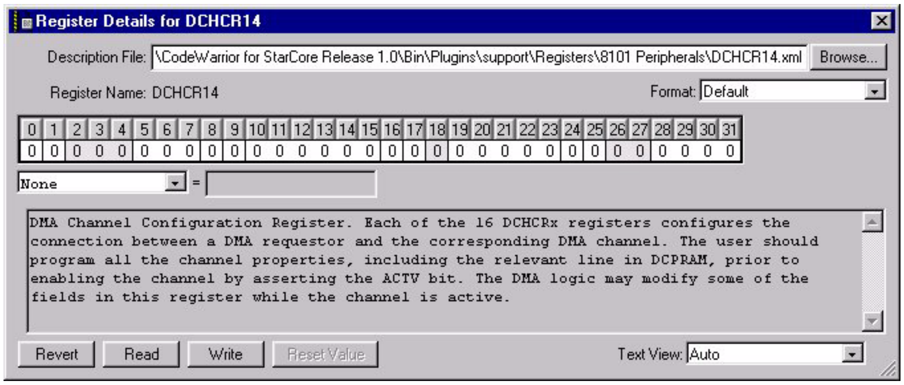
View Register Descriptions

To see registers and register descriptions:

1. Choose **View > Register Details** (Windows operating system) or **Window > Register Details** (Solaris operating system).

   The IDE displays the **Register Details** window. (See Figure 6.4.)

Figure 6.4 Register Details Window



Description File: \CodeWarrior for StarCore Release 1.0\Bin\Plugins\support\Registers\8101 Peripherals\DCHCR14.xml

Register Name: DCHCR14

Format: Default

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
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 =

```
DMA Channel Configuration Register. Each of the 16 DCHCRx registers configures the
connection between a DMA requestor and the corresponding DMA channel. The user should
program all the channel properties, including the relevant line in DCPRAM, prior to
enabling the channel by asserting the ACTV bit. The DMA logic may modify some of the
fields in this register while the channel is active.
```

Text View: Auto

2. In the Description File text box, type the name of the register description file to display. Alternatively, click **Browse** to display a dialog box that you can use to select the register description file to display.

NOTE Some registers have multiple modes (meaning that the bits of the register have different meanings depending on the mode the register is in). If the register you are examining has multiple modes, browse the register description files to find the correct file for the register and mode that you are examining.

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, as follows:

`ORx_GCPM`
`ORx_UPM`
`ORx_SDRAM`

(where *x* is a number between 0 and 11, excluding 8 and 9.)

Similarly, other multi-mode registers have description files that use an underscore followed by a descriptive suffix.

The **Register Details** window displays the applicable register values and descriptions.

NOTE You can change the format in which the CodeWarrior IDE displays registers using the Format listbox. You also can change the textual information the CodeWarrior IDE displays using the options of the Text View listbox.

Tips for Debugging Assembly Language

If you set a breakpoint in assembly language source code, the source pane of the debugger 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 in the program) to view that source code.

NOTE Ensure that the address value that you enter is less than that of the current location when you change the program counter value.

(Alternatively, you can view assembly language source in the memory window.)

Change the Program Counter Value

To change the program counter value:

1. Choose **Debug > Change Program Counter**

   The **Change Program Counter** dialog box appears.

2. Enter an address (in hexadecimal).

   The source pane in the debugger window updates with the program counter at the specified location.

Cycle Counter in the Simulator

If you are debugging using the simulator, CodeWarrior for StarCore DSPs lets you get the cumulative machine cycle count and the machine instruction count.

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 interactive use.

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

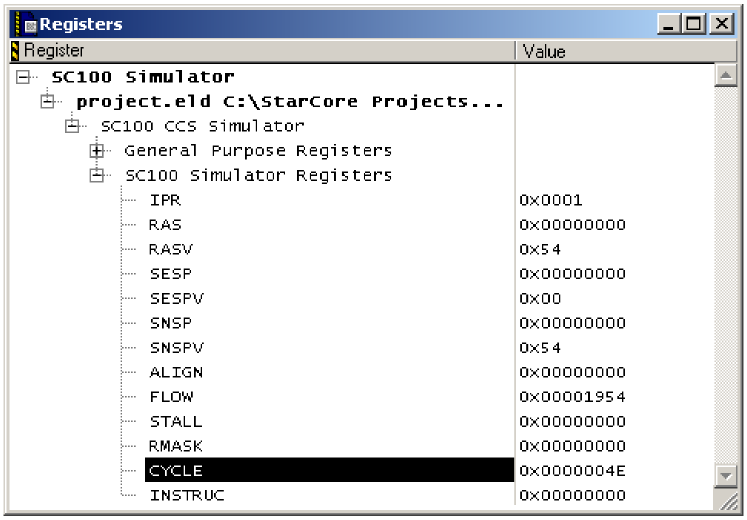
1. Set a breakpoint before the beginning of the algorithm of interest.
2. Set a breakpoint after the end of this algorithm.
3. Execute the program to the first breakpoint.
4. Reset the machine cycle count.

   To do this, follow these steps:

   a. Select **View > Registers**

      The simulator's **Registers** window appears. (See Figure 6.5.)

Figure 6.5 Simulator Registers Window



Register	Value
SC100 Simulator	
project.eld C:\StarCore Projects...	
SC100 CCS Simulator	
General Purpose Registers	
SC100 Simulator Registers	
IPR	0x0001
RAS	0x00000000
RASV	0x54
SESP	0x00000000
SESPV	0x00
SNSP	0x00000000
SNSPV	0x54
ALIGN	0x00000000
FLOW	0x00001954
STALL	0x00000000
RMASK	0x00000000
CYCLE	0x0000004E
INSTRUC	0x00000000

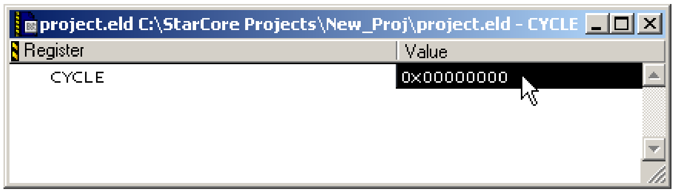
b. Expand the SC100 Simulator Registers node of the displayed tree.

A list of the simulator register appears.

c. Double-click the register named `CYCLE`

The `CYCLE` register and its value appear in the debugger's register window. (See Figure 6.6.)

Figure 6.6 The CYCLE Register in the Debugger Register Window


project.eld C:\StarCore Projects\New_Proj\project.eld - CYCLE

Register	Value
CYCLE	0x00000000

d. Set the value of the `CYCLE` register to 0.

5. Execute the program to the second breakpoint.

The `CYCLE` register field of the register window shows the machine cycles used.

Debugging a .eld File Without a CodeWarrior Project

To debug an `.eld` file that does not have an associated CodeWarrior project:

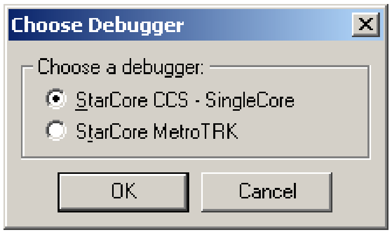
1. Run the CodeWarrior IDE.
2. Choose **File > Open**

   The **Open** dialog box appears.
3. Using this dialog box, select the `.eld` file to be debugged and click **Open**.

   The IDE displays the **Choose a debugger** dialog box. (See Figure 6.7.)

TIP Alternatively, you can drag and drop an `.eld` file onto the IDE.

Figure 6.7 The Choose a debugger Dialog Box



Choose Debugger

Choose a debugger:

● StarCore CCS - SingleCore
○ StarCore MetroTRK

4. In the **Choose Debugger** dialog box, select the appropriate debugger
5. Press **OK**

   The IDE creates a CodeWarrior project for the `.eld` file and displays the result in a project window.
6. Press **Alt-F7**

   The IDE displays the **Target Settings** window.
7. From the panel name 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. From the Target listbox, choose the appropriate debug target.

NOTE If the source code files used to build the `.eld` file to be debugged are not in the same directory as the `.eld` file, add the paths to the source code files in the **Access Paths** target settings panel.

9. Choose **Project > Debug**

   The CodeWarrior debugger starts a debug session.

NOTE If you debug an `.eld` file without a corresponding CodeWarrior project, the IDE sets the Build before running option of the **Build Settings** preference panel to Never.
As a result, once you have debugged an `.eld` file, you cannot build another project until you change the Build before running option to Ask or Always.

System-Level Connect

You can use the CodeWarrior debugger to perform a system-level connect to a target board, either before or after downloading your program to the board. Once you connect to the target board, you can examine system registers and memory.

Perform a System-Level Connect

You can perform a system-level connect (by choosing **Debug > Connect**) any time you have a project window open and your target board is connected.

To perform a system-level connect, follow these steps:

1. Choose **Project > Run**

   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. Choose **Debug > Kill**

   The debugger stops running.

3. Ensure that the project window for the program you downloaded is selected.
4. Choose **Debug > Connect**

   The debugger connects to the board.

You now can examine registers and the contents of memory on the board.

Initialization File

The initialization file is a text file that contains commands that tell the debugger how to initialize your hardware after reset but before downloading code. Use the 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 box and specify the name of your initialization file in the SC100 Debugger Target settings panel.

The topics in this section are:

- Example Initialization File
- Customizing an Initialization File and JTAG Initialization File for 8101 Hardware
- Setting the IMMR Value
- Initialization File Commands

Example Initialization File

Listing 6.1 shows part of an 8101 initialization file named `8101_Initialization.cfg`. This file is in one of the directories listed below.

Windows

`installDir``\StarCore_Support\`
`Initialization_Files\RegisterConfigFiles\MSC8101`

Solaris

`installDir/CodeWarrior_ver_dir``/starcore_support/`
`Initialization_Files/RegisterConfigFiles/MSC8101`

You can customize the contents of `8101_Initialization.cfg` if needed.

Listing 6.1 Excerpt from an 8101 Initialization File

```
#-------------------------------------------------------------
#  8101 Initialization File
#-------------------------------------------------------------
POST-RESET-ON

writemmr16 IMMR 0x1470
writemmr32 BCR 0x00900000
```

```
######### bank0_init ##########################
#q001->mem_regs[0].memc_or = 0xff800866  ;
writemmr32 OR0 0xff800866

#q001-> mem_regs[0].memc_br  = 0xfc001801;
writemmr32 BR0 0xff801801
 .
 .
 .
```

Customizing an Initialization File and JTAG Initialization File for 8101 Hardware

Two files define labels for 8101 registers. One uses ordinary data structures (`MMapQ001.h`); the other uses packed data structures (`msc8101.h`). You can customize either of the files if needed.

If you are using 8101 hardware, include either `MMapQ001.h` or `msc8101.h` in your project. Alternatively, you can include a customized version of either file, if you previously created one.

For Windows, `MMapQ001.h` is in this directory:

`installDir``\StarCore_Support\flash_programmer_support`

For the Solaris operating system, `MMapQ001.h` is in this directory:

`installDir/CodeWarrior_ver_dir/`
`starcore_support/flash_programmer_support`

For Windows, the file `msc8101.h` is in this directory:

`installDir``\Stationery\`
`StarCore\Msc8101\C_Source_Big_Endian`

For the Solaris operating system, the file `msc8101.h` is in this directory:

`installDir/CodeWarrior_ver_dir/``CodeWarrior_IDE/`
`(Project Stationery)/MSC8101/C_Source_Big_Endian`

Setting the IMMR Value

The Internal Memory Map register (IMMR) holds the base address for the 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 register to determine the address of other PPC-bus memory-mapped registers.

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

If you initialize the IMMR by address, the debugger behaves as if you have left the IMMR unchanged. In that case, the debugger uses the default reset value for the IMMR register (`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 6.1 Initialization File Commands

CCSConfigTemplate	CCSCoreRunMode
CCSStopCore	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	

CCSConfigTemplate

Sets the value of the specified option for the specified core of a StarCore chip.

```
CCSConfigTemplate JTAG_index option value
```

Parameters

JTAG_index

JTAG chain identifier of the core to which to set the specified option to the specified value.

option

Option to which to assign the specified value.

value

Value to assign to the specified option.

Remarks

The meaning of the option specified depends on the *type* of the targeted core. In other words, an option that has one effect on one core type may have an entirely different effect on a different core type.

For core type `MSC8102Sync`, just this form of the CCSConfigTemplate command is supported:

```
CCSConfigTemplate JTAG_index  0  < 0 | 1 >
```

The first argument is the core ID to which the command applies. The second argument is always 0. The third argument defines whether to use the DSI boot mode (use 1) or system mode (use 0).

For example, consider two 8102 chips. To make the second of these chips operate in DSI mode, place this command in your initialization file:

```
CCSConfigTemplate 5 0 1
```

For core type `SC140`, the CCSConfigTemplate command supports these values for *option*:

- `0`

  Deprecated

- `1`

  Deprecated

- 2

  Enables the sc140cfg_fast_mem option. This option controls use of fast memory writes.

  If this option is enabled and a reasonably large data transfer into target storage is requested, the command converter server (CCS) loads a small piece of code into target storage and executes that code to expedite transfer of data. This greatly increases data write performance.

  value = 0 disable fast memory write routines
  value = 1 [DEFAULT] enable fast memory write routines

- 3

  Enables the sc140cfg_endian option. This option tells CCS how to handle data transfers to the target board.

  value = 0 SC140CFG_ENDIAN_BIG - board is treated as big endian
  value = 1 SC140CFG_ENDIAN_LITTLE - board is treated as little endian
  value = 2 SC140CFG_ENDIAN_AUTO - board is treated with endianess it reports.

NOTE The debugger normally sets this value based upon project settings.

- 4

  Enables the sc140cfg_slow_memory option.

  Enabling this option makes memory transfers with many wait-states more reliable.

  This feature indicates that CCS must wait for the core to return to debug mode after executing instructions used to read or write memory.

  value = 0 off
  value = 1 on [default]

NOTE It is highly recommended that you not modify this setting.

NOTE This option does not conflict with the sc140cfg_fast_mem option.

- 5

  Enables the sc140cfg_msc8101_sypcr option.

  Use this option to specify the value to be written to the System Protection Control Register (SYPCR) after reset of 8101 targets (that is. targets that CCS recognizes as 8101 based upon JTAG ID).

  The value specified must disable the watchdog timer. Failure to do this results in the target repeatedly resetting at some unknown interval as the watchdog timer triggers.

  The default value is `0x0000FBC3`

- 6

  Enables the sc140cfg_disable_polling option.

  Use this option to enables and disable polling of the EOnCE transmit and receive buffers by CCS for purposes of HSST and real-time data transfer.

  value = 0 enable polling [default]
  value = 1 disable polling

NOTE It is highly recommended that you not modify this setting.

- 7

  Enables the sc140cfg_eonce_base option.

  Use this option to specify the EOnCE register base address.

  – The default value for msc8101 and msc8102 is 0x00EFFE00

  – The default value for Rainbow is 0x00FFFE00

  – The default value for Default for StarCore LLC boards is 0x80000000

CCSCoreRunMode

Instructs CCS to poll the run mode of the specified core.

Use this command to force CCS to discover that a core has stopped when CCS does not expect the core to have stopped.

For example, during initialization of core 0, CCS does not expect cores 1-4 (which are not being explicitly manipulated) to enter a stop state. However, if you perform a synchronous stop on core 0, all of the cores stop. By forcing CCS to poll, CCS discovers that the other cores have stopped and, as a result, you can manipulate these cores as well.

```
CCSCoreRunMode core_id
```

Parameter

core_id

Identifier of the core to query.

Remarks

The CCSCoreRunMode command is used primarily to prompt CCS to poll and discover that a core has changed run mode as an indirect consequence of communication with a different JTAG device.

The command does not return a value to the caller.

Example:

```
CCSStopCore 0 #causes synch-stop of all 4 cores on chip
CCSCoreRunMode 1 # CCS polls & finds that core #1 has stopped
CCSCoreRunMode 2 # CCS polls & finds that core #2 has stopped
CCSCoreRunMode 3 # CCS polls & finds that core #3 has stopped
CCSCoreRunMode 4 # CCS polls & finds that core #4 has stopped
```

CCSStopCore

Stops the identified core by putting it in debug mode. If the identified core is of type `MSC8102Sync`, then the entire device is stopped (that is, all four cores are synchronously stopped).

In multi-core initialization, you may need this command to quiesce the slave cores while the aggregator is being configured.

`CCSStopCore` *`core_id`*

Parameter

`core_id`

Identifier of the core to stop.

DSPJTAGClockSpeed

Sets the speed of the JTAG clock.

`DSPJTAGClockSpeed` *`speed`*

Parameter

`speed`

Kilohertz rate, such as 1000.

PRE-RESET-ON

Specifies commands to be executed 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 commands `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 command `PRE-RESET-ON` or `POST-RESET-ON`, the debugger executes all the commands before the reset.

PRE-RESET-OFF

Indicates the final command to be executed before the reset.

```
PRE-RESET-OFF
```

POST-RESET-ON

Indicates commands to be executed after the reset.

```
POST-RESET-ON
```

Remarks

The command `POST-RESET-OFF` halts this functionality. For example, if the initialization file includes both commands `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 command `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 an MMR (Memory Mapped Register).

```
setMMRBase16 MMR_register_name base_value
```

Parameters

MMR_register_name

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

NOTE Typically, specify IMMR (Internal Memory Map Register) for this parameter.

base_value

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

Remarks

Use this command to attach to a program that is already running on a board. The command is required in this scenario because, at attach-time, the debugger cannot determine the value of an MMR's base 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 box of this panel.
3. Check the Use Initialization File box in this panel and enter the path and name 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).

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).

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).

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).

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).

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).

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).

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).

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).

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).

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).

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).

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).

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).

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).

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).

Kernel Awareness

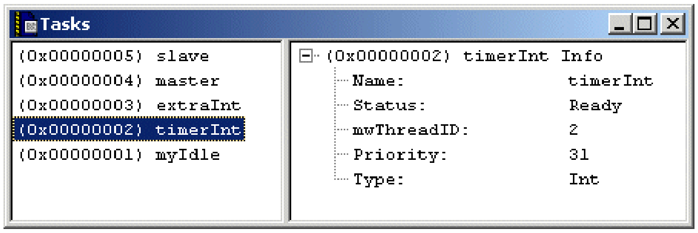
You can indicate that you are using one of the supported real-time operating systems (RTOS) by selecting your RTOS from the Kernel Awareness listbox of the SC100 Debugger Target settings panel.

If you are not using an RTOS, select None from the Kernel Awareness listbox.

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

When you select Task Info from the OSE menu, the **Tasks** window displays information about all the running tasks. (See Figure 6.8.)

Figure 6.8 Tasks Window



```
(0x00000005) slave
(0x00000004) master
(0x00000003) extraInt
(0x00000002) timerInt
(0x00000001) myIdle
```

```
⊟ (0x00000002) timerInt Info
   Name:          timerInt
   Status:        Ready
   mwThreadID:    2
   Priority:      31
   Type:          Int
```

Clicking a task name in the left pane of the **Tasks** window selects a task and causes the right pane of the window to display information relevant to the currently selected task.

Table 6.2 lists the **Tasks** window descriptors.

Table 6.2 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.
Priority	An integer value that indicates the priority for running a task.
Type	The type of the task. The possible values are: • 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

Command-Line Debugging

You can debug from the command line as well as from within the CodeWarrior IDE. When you debug from the command line, you can use:

- Tcl commands
- Command-line debugger commands

Tcl Support

This section describes command-line debugger’s Tcl support.

Resolution of Conflicting Command Names

The names of several command-line debugger commands conflict with the Tcl commands. Table 6.3 explains how the command-line debugger resolves such conflicts (if the mode is set to auto).

Table 6.3 Resolving Clashing Commands

Command	Resolution
load	If you pass the command-line debugger a load command that includes a filename containing the suffix `.eld` or `.mcp`, the debugger loads the project. Otherwise, the debugger invokes the Tcl load command.
break	If you pass the command-line debugger a break command from within a script and the command has no arguments, the debugger invokes the Tcl break command. Otherwise, the debugger interprets a break command as a command to control breakpoints.
close	If you pass the command-line debugger a close command that has no arguments, the debugger terminates the debug session. Otherwise, the debugger invokes the Tcl close command.

Execution of Script Files

Tcl usually executes a script file as one large block, returning only after execution of the entire file. For the `run` command, however, the command-line debugger executes script files line-by-line. If a particular line is not a complete Tcl command, the debugger appends the next line. The debugger continues appending lines until it gets a complete Tcl script block.

For example, Listing 6.2 shows code that includes a script. For the Tcl `source` command, the debugger executes this script as one block. But for the `run` debug command, the debugger executes this script as two blocks: the `set` statement and the `while` loop.

Listing 6.2 Example Tcl Script

```
set x 0;

  while {$x < 5}
  {
    puts "x is $x";

    set x [expr $x + 1]
  }
```

NOTE The `run` debug command synchronizes debug events between blocks in a script file. For example, after a `go`, `next`, or `step` command, `run` polls the debug thread state and does not execute the next line or block until the debug thread terminates.

However, the Tcl `source` command does not consider the debug thread state. Consequently, use the `run` debug command to execute script files that contain these debug commands: `debug`, `go`, `next`, `stop`, and `kill`.

Tcl Startup Script

The command-line debugger can automatically run a Tcl script each time you open the command-line debugger window. This script is called a startup script.

You can use both Tcl and command-line debugger commands in the startup script. For example, you might include commands that set an alias or a define color configuration in a startup script.

To create a command-line debugger startup script, follow these steps:

1. Put the desired Tcl and command-line debugger commands in a text file.
2. Name this file `tcld.tcl`
3. Place `tcld.tcl` in one of the directories listed below.
   - On a Windows® PC, put `tcld.tcl` in the system directory.
     For example, on Windows XP, put `tcld.tcl` in the `WINDOWS` directory.
   - On a Solaris Workstation, put `tcld.tcl` in your home directory.

NOTE There is no synchronization of debug events in the startup script. Consequently, put the `c` debug command to the startup script and place these debug commands in another script so they will execute properly:
`debug`, `go`, `stop`, `kill`, `next`, and `step`.

Command-Line Debugging Tasks

Table 6.4 provides instructions for common command-line debugging tasks.

Table 6.4 Common Command-Line Debugging Tasks

Task	Instructions	Comments
Open command-line debugger window	From IDE menu bar, select **Debug > Command Line Debugger**	
Enter one command	1. On the command line, type a command followed by a space. 2. Type any valid command-line options, separating each with a space. 3. Press **Enter**	You can use shortcuts instead of complete command names, such as `b` for `break`.

Table 6.4 Common Command-Line Debugging Tasks (*continued*)

Task	Instructions	Comments
Enter multiple commands	1. Put several commands to a file. 2. Save the file with the `.tcl` extension. 3. Execute the script using the `run` command.	The extension `.tcl` makes the file a script file.
View debug command hints	Type command and space — Hint shows syntax for rest of command.	
	Type space at start of command line — Hint text for commands appears at bottom of command-line debugger window.	
Repeat a command	1. Type command and press **Enter** — Debugger executes the command. 2. Press **Enter** again — Debugger executes the command again.	To see ID numbers of commands, execute the history command.
Review previous commands	Press **Up Arrow** and **Down Arrow** keys.	
Clear command from the command line	Press the **Esc** key.	
Stop an executing script	Press the **Esc** key.	
Toggle between insert/overwrite mode	Press the **Insert** key.	
Scroll up/ down a page	Press **Page Up** or **Page Down** key.	
Scroll up/ down one line	Press **Ctrl-Up Arrow** or **Ctrl-Down Arrow** keys.	
Scroll left/right one column	Press **Ctrl-Left Arrow** or **Ctrl-Right Arrow** keys.	
Scroll to beginning or end of buffer	Press **Ctrl-Home** or **Ctrl-End** keys.	
Copy text from command-line debugger window	1. Drag cursor over text. 2. Press **Enter**	Selecting **Edit > Copy** is an alternative to pressing **Enter**.
Paste text into command-line debugger window	1. Put mouse cursor on the command line. 2. Right-click the mouse button.	Selecting **Edit > Paste** is an alternative to right-clicking.

Command-Line Debugger Commands

This section lists and defines each command-line debugger command.

alias

Creates an alias for a debug command, removes such an alias, or lists all current aliases.

```
al[ias] [alias_name] [alias_definition]
```

Parameters

`alias_name`

New alias value.

`alias_definition`

Command.

Examples

Table 6.5 shows examples of the `alias` command.

Table 6.5 alias Command-Line Debugger Command—Examples

`alias x close`	Makes `x` an alias for the `close` command.
`alias`	Shows all current aliases.
`alias x`	Removes previously defined alias `x`.

break

Sets a breakpoint, removes a breakpoint, or displays current breakpoints.

```
b[reak] [func_name | machine_addr] |
     [file_name line_num [column_number]] |
     [func_name | brkpt_num off]
```

Parameters

`func_name`

Function name.

`machine_addr`

Machine address.

`file_name`

Name of a file.

`line_num`

Line number.

`column_number`

Column number.

`brkpt_num`

Breakpoint number.

Examples

Table 6.6 shows examples of the `break` command.

Table 6.6 break Command-Line Debugger Command—Examples

`break fie`	Sets breakpoint on function `fie`.
`break fum off`	Removes breakpoint from function `fum`.
`break $1048`	Sets breakpoint on machine address $1048.
`break`	Displays all current breakpoints, including breakpoint numbers.
`break #4 off`	Removes breakpoint 4.
`Break sc_main.c 15`	Sets breakpoint on line 15 of file `sc_main.c`.

bringtofront

Displays the command-line debugger window in front of all other windows, or lets other windows appear in front of the command-line debugger window.

```
bri[ngtofront] [on | off]
```

Examples

Table 6.7 shows examples of the `bringtofront` command.

Table 6.7 bringtofront Command-Line Debugger Command—Examples

`bringtofron on`	Moves command-line debugger window to front of screen.
`bringtofront off`	Lets other windows appear in front of the command-line debugger window.
`bringtofront`	Toggles current bringtofront setting.

cd

Changes to a different directory or displays the current directory. (Pressing the **Tab** key completes the directory name automatically.)

```
cd [path]
```

Parameter

`path`

Directory pathname; accepts asterisks and wildcards.

Examples

Table 6.8 shows examples of the `cd` command.

Table 6.8 cd Command-Line Debugger Command—Examples

`cd`	Displays current directory.
`cd c:`	Changes to the C: drive root directory.
`cd d:/mw/0622/test`	Changes to the specified D: drive directory.
`cd c:p*s`	Changes to any C: drive directory whose name starts with p and ends with s.

change

Changes the contents of a register, a memory location, or a block of registers or memory locations.

```
c[hange] [ register | reg_block | address | addr_block ]
         [ value ] [ 8bit |16bit | 32bit | 64bit ]
```

Parameters

`register`

Register name.

`reg_block`

Names of the first and last registers of a block, specified as *register_first*`..`*register_last.*

`address`

Memory location address.

`addr_block`

Memory location block, specified in either of two ways:

- First and last addresses of the block, in the form *address_first*`..`*address_last*
- First address and number of locations, in the form *address#count*

`value`

New fractional, hexadecimal, or decimal value.

Remarks

NOTE You cannot change some memory locations or registers if you are using a hardware board (for example, ROM memory).

The default memory-access mode depends on the type and size of the new value:

- If `value` is fractional, the default mode is 16-bit.
- If `value` is hexadecimal, the length determines the default mode:
  - length <=2: 8-bit, as in $1, $12, or $01
  - 2 < length <= 4: 16-bit, as in $0001 or $123
  - 4 < length <= 8: 32-bit, as in $00000123 or $1234567
  - length > 8: 64-bit, as in $123456789
- If `value` is decimal, the size determines the default mode:
  - value <= 0xff: 8-bit, as in 0, 54, or 255
  - value > 0xff: 16-bit, as in 256, 65535, or 1000
  - 0xffff < value <= 0xffffffff: 32-bit, as in 65536 or 3253532
  - value > 0xffffffff: 64-bit, as in 4294967296

Examples

Table 6.9 shows examples of the `change` command.

Table 6.9 change Command-Line Debugger Command—Examples

`change R1 $123`	Assigns value 123 to register R1.
`change R1..R5 $5432`	Assigns value 5432 to registers R1 through R5.
`c p:10..17 3456`	Assigns value 3456 to addresses 10 through 17.
`c p:18..1f $03456`	Assigns value 00003456 to addresses 18 through 1F.

cls

Clears the command line debugger window.

```
cl[s]
```

close

Closes the opened default project.

```
clo[se]
```

config

Displays current configuration information, provides the name of the default project or build target, or configures:

- command-line debugger window colors
- command-line debugger window scrolling size
- command-line debugger window mode
- Default build target
- Hexadecimal prefix
- Memory identifier
- Processor name
- Subprocessor name

```
conf[ig] [ c[olor] [r | m | c | s | e | n]
                text_color [background_color] |
m[ode] [ dsp | tcl | auto] |
s[croll] number_of_lines |
h[exprefix] hexadecimal_prefix |
mem[identifier] memory_identifier |
p[rocessor] processor_name [subprocessor_name] ]
```

Parameters

color text indicators — `r` (registers), `m` (memory), `c` (commands), `s` (script), `e` (errors), or `n` (normal)

`text_color`

Text color values for red, green, and blue, each from 0 through 255.

`background_color`

Background color values for red, green, and blue, each from 0 through 255.

`mode`

Command-name conflict resolution mode:

- `dsp` (use command-line debug commands)
- `tcl` (use tcl commands)
- `auto` (resolve automatically)

`number_of_lines`

Number of lines to scroll.

`hexadecimal_prefix`

Prefix for display of hexadecimal values.

`memory_identifier`

Memory identifier.

`processor_name`

Name or identifier of target processor.

`subprocessor_name`

Name or identifier of target subprocessor.

`target_name`

Name of build target.

Examples

Table 6.10 shows examples of the `config` command.

Table 6.10 config Command-Line Debugger Command—Examples

`config`	Displays current configuration information.
`config c e $ff $0 $0`	Sets error text to red.
`config c r $0 $0 $0 $ff $ff $ff`	Sets register display to black, on a white background.
`config s $10`	Sets page scrolling to 16 (decimal) lines.
`config m dsp`	Sets clash resolution to dsp mode.
`config hexprefix 0x`	Specifies 0x prefix for hexadecimal values.
`config memidentifier m`	Sets memory identifier to m.
`config processor 8101`	Sets processor to 8101.
`config project`	Displays default-project name.
`config target`	Displays default build-target name.
`config target debug release x86`	Changes default build-target name to debug release x86.

copy

Copies contents of a memory address or address block to another memory location.

```
co[py] addr_group addr
```

Parameters

`addr_group`

One of these memory-address specifications:

- A single address
- First and last block addresses, in the form *address_first*`..`*address_last*
- First address and number of locations, in the form *address#count*

`addr`

First address of the destination memory block.

Examples

Table 6.11 shows examples of the `copy` command.

Table 6.11 copy Command-Line Debugger Command—Examples

`copy p:00..1f p:30`	Copies contents of memory addresses `00` through `1f` to a contiguous memory block beginning at address `30`.
`copy p:20#10 p:50`	Copies contents of `10` consecutive memory locations, starting at address `20`, to a contiguous memory block beginning at address `50`.

debug

Starts a command-line debugger session.

`de[bug] [`*`project_file_name`*`]`

Parameters

`project_file_name`

Name of a project file.

Examples

Table 6.12 shows examples of the `debug` command.

Table 6.12 debug Command-Line Debugger Command—Examples

`debug`	Starts a command-line debugger session for the open, default project.
`debug des.mcp`	Starts a command-line debugger session for project `des.mcp`.

dir

Lists directory contents.

NOTE You can use the `dir` command-line debugger command the same way you use the `dir` operating system command with one exception: You cannot use dir with any option that requires user keyboard input (such as `/p` for the `dir`

operating system command).
The same is true of the `ls` command.

Examples

Table 6.13 shows examples of the `dir` command.

Table 6.13 dir Command-Line Debugger Command—Examples

`dir`	Lists all files of the current directory.
`di *.txt`	Lists all current-directory files that have the `.txt` filename extension.
`dir c:/tmp`	Lists all files in the `tmp` directory on the `C:` drive.
`dir /ad`	Lists only the subdirectories of the current directory.

disassemble

Disassembles the instructions of the specified memory block.

`di[sassemble] addr_block`

Parameter

`addr_block`

Memory location block, specified in either of two ways:

- First and last addresses of the block, in the form *address_first*`..`*address_last*
- First address and number of locations, in the form *address#count*

Examples

Table 6.14 shows examples of the `disassemble` command.

Table 6.14 disassemble Command-Line Debugger Command—Examples

`disassemble`	Disassembles instructions from the PC (if changed) or from the last address.
`disassemble p:0..20`	Disassembles contents of memory address block 0 to 20.
`disassemble p:$50#10`	Disassembles contents of memory location 50, plus nine subsequent locations.

display

Displays the contents of a register or memory location; lists all register sets of a target; adds register sets, registers, or memory locations; or removes register sets, registers, or memory locations.

```
d[isplay] [ regset ] | {on all] | [off all] |  [off id_number ]
      | [on reg_group | reg_block
      | addr_group [8bit | 16bit | 32bit | 64bit]]
      | [off reg_group | reg_block | addr_group [8bit | 16bit
      | 32bit | 64bit]]
```

Parameters

`id_number`

Display-item identification number.

`reg_group`

List of register sets, separated by spaces. Possible sets are:

```
GPR, SIM, EONCE, GEN_SIU, MEM_CTRL, SYS_INT_TIM, DMA,
INT_CTRL, ClocksReset, IOPort, CPMTimers, SDMAGen,
IDMA, FCC, BRG, I2C, SP, SCC, SMC, SPI, CPMMux, SI, MCC,
HDI16, EFCOP, PIC, QBUS, ALL
```

`reg_block`

Names of the first and last registers of a block, specified as *register_first*`..`*register_last.*

`addr_group`

One of these memory-address specifications:

- A single address
- First and last block addresses, in the form *address_first*`..`*address_last*
- First address and number of locations, in the form *address#count*

Remarks

The **Registers** window lists the register sets that can be part of a *reg_group*. (To display this window, make sure that the debugger window is open, and then select **View > Registers**.)

When you display registers or memory locations, the display command returns the values to Tcl. Consequently, you can embed the display command to Tcl as follows:

```
set r0 [display r0]           ; puts $r0 ;
set r0M [display p:$r0 32bit] ; puts $r0M
set r0r1 [display r0..r1]     ; puts $r0r1 ;
```

The default memory display unit is 16 bits, but you can specifying a different size. To change the number base of display values, use the radix command.

Examples

Table 6.15 shows examples of the display command.

Table 6.15 display Command-Line Debugger Command—Examples

display	Shows default display items, such as register sets. (The command-line debugger executes this command whenever program execution stops.)
display on	Lists the default display items.
display regset	Lists all available register sets for the target processor.
display on EONCE QBUS	Adds the EONCE and QBUS register sets to the default display items.
display off SIM	Removes the SIM register set from the default display items.
display on ALL	Adds all supported register sets to the default display items.
display on p:230#10	Adds the specified memory locations to the default display items.
display off #2	Removes item number 2 from the default display items.
display R1	Shows contents of register R1, then returns to Tcl.
display R1..R5	Shows contents of registers R1 through R5.
display p:00..$100	Shows contents of memory addresses 0 to 256 (hexadecimal 100).
display p:00#$200 8bit	Shows contents of the 512-address (200 hexadecimal address) block, starting with address 0, in 8-bit mode.

evaluate

Displays the type or value of a C variable.

```
e[valuate] [ b | d |f | h | u] variable
```

Parameters

Display format — b (binary), d (decimal), f (fraction), h (hexadecimal), or u (unsigned)

variable

Variable name.

Examples

Table 6.16 shows examples of the `evaluate` command.

Table 6.16 evaluate Command-Line Debugger Command—Examples

`evaluate`	Lists the types for all variables in the current and global stack.
`evaluate i`	Returns the value of variable `i`.

exit

Closes the command-line debugger window.

```
[ex]it
```

go

Starts to debug your program from the current instruction.

```
g[o] [ all | time_period ]
```

Parameters

`all`

Specifier for all target programs (of multiple cores).

time_period

Number of seconds program executes, if no breakpoint or stop command halts execution.

Remarks

If you execute the go command interactively, the command returns immediately, and target-program execution starts. Then you can wait for execution to stop (for example, due to a breakpoint) or type the stop command.

If you execute the go command in a script, the command-line debugger polls until the debugger stops (for example, due to a breakpoint). Then the command-line debugger executes the next command in the script. If this polling continues indefinitely because debugging does not halt, you can press the **Esc** key to stop the script.

Examples

Table 6.17 shows examples of the go command.

Table 6.17 go Command-Line Debugger Command—Examples

go	Returns immediately. The program stops at the first breakpoint or when you enter a stop command.
go 1	Stops polling the target if execution does not reach a breakpoint within 1 second. (Also sets Tcl variable $still_running to 1.)
go all	Starts all target programs of multiple cores.

help

Displays debug command help in the command-line debugger window.

h[elp] [*command*]

Parameter

command

Name or short-cut name of a command.

Examples

Table 6.18 shows examples of the help command.

Table 6.18 help Command-Line Debugger Command—Examples

`help`	Lists all debug commands.
`help b`	Displays help information for the `break` command.

history

Lists the history of the commands entered during the current debug session.

`hi[story]`

hsst_attach_listener

Sets up a Tcl procedure that the debugger notifies any time there is data in an communication channel.

`hsst_a[ttach_listener]` *`channel_id tcl_proc_name`*

Parameters

`channel_id`

Channel identifier.

`tcl_proc_name`

Name for the Tcl procedure.

Example

This command automatically executes procedure `call_back` when target data is available on a communication channel.

```
proc call_back { } {
  global hsst_descriptor;
  global hsst_nmemb;
  global hsst_size;
  puts [ hsst_read $hsst_size
         $hsst_nmemb $hsst_descriptor ]
}
```

```
set cid [ hsst_open channel1 ]
```

```
hsst_attach_listener $cid call_back;
```

hsst_block_mode

Specifies blocked mode for a communication channel. This blocks all `hsst_read` calls until the requested amount of data is available from the target. (Blocked mode is the default setting for all channels.)

```
hsst_b[lock_mode] channel_id
```

Parameter

`channel_id`

Channel identifier.

Example

Specify blocked mode for channel `$cid`:

```
hsst_block_mode $cid
```

hsst_close

Closes a communication channel with the host machine.

```
hsst_c[lose] channel_id
```

Parameter

`channel_id`

Channel identifier.

Example

Close channel `$cid`:

```
hsst_close $cid
```

hsst_detach_listener

Detaches a listener (previously attached for automatic data notification).

```
hsst_d[etach_listener] channel_id
```

Parameter

channel_id

Channel identifier.

Example

Detach the listener from channel `$cid`:

```
hsst_detach_listener $cid
```

hsst_log

Logs data to a directory.

```
hsst_l[og] [ directory_name ]
```

Parameter

directory_name

Name of the directory to receive logging data.

Example

Table 6.19 shows examples of the `hsst_log` command:

Table 6.19 hsst_log Command-Line Debugger Command—Examples

`hsst_log c:\logdata`	Debugger logs the data to directory logdata.
`hsst_log`	Debugger stops logging.

hsst_noblock_mode

Specifies unblocked mode for a communication channel. This admits all `hsst_read` calls immediately, with any available data, although limited by the requested size.

```
hsst_n[oblock_mode] channel_id
```

Parameter

channel_id

Channel identifier.

Example

Specify unblocked mode for channel $cid:

```
set cid [ hsst_open channel1 ]
hsst_noblock_mode $cid
```

hsst_open

Opens a communication channel with the host machine.

`hsst_o[pen]` *`channel_id`*

Parameter

`channel_id`

Channel identifier.

Example

Open channel `$cid`:

```
set cid [hsst_open ochannel1]
```

hsst_read

Reads data from an open communication channel.

`hsst_r[ead]` *`size nmemb channel_id`*

Parameters

`size`

Number of bytes in data items.

`nmemb`

Number of data items.

`channel_id`

Channel identifier.

Example

Read 15 data items, each of 1 byte, from channel `$cid`:

```
puts [hsst_read 1 15 $cid]
```

hsst_write

Writes data to an open communication channel.

`hsst_w[rite]` *`size data channel_id`*

Parameters

`size`

Number of bytes in data items.

`data`

Number of data items.

`channel_id`

Channel identifier.

Example

Write `0x1234`, as 2 bytes, to `$cid`:

```
hsst_write 2 0x1234 $cid
```

kill

Stops one or all current debug sessions.

`k[ill] [all]`

Parameter

`all`

Specifier for all debug sessions.

Examples

Table 6.20 shows examples of the `kill` command.

Table 6.20 kill Command-Line Debugger Command—Examples

`kill`	Stops the current debug session.
`kill all`	Stops all debug sessions (of multiple cores).

load

Opens a project or load records into memory.

`l[oad]` *`project_file_name`* `|` *`eld_file_name`*

or

`l[oad] -h | -b` *`file_name`* `[` *`memory_location`* `]`

Parameters

`project_file_name`

Name of project file to be loaded.

`eld_file_name`

Name of object (.eld) file to be converted to a project file, then loaded.

`-h`

Hexadecimal specifier.

`-b`

Binary specifier

`file_name`

Name of file to be loaded into memory.

`memory_location`

Destination address in memory.

Examples

Table 6.21 shows examples of the `load` command.

Table 6.21 load Command-Line Debugger Command—Examples

`load des.mcp`	Loads project `des.mcp`.
`load des.eld`	Creates default project from the `des.eld` object file, then loads the project.
`load -h dat.lod`	Loads contents of hexadecimal file `dat.lod` into memory.
`load -b dat.lod p:$20`	Loads contents of binary file `dat.lod` into memory, beginning at `$20`.

log

Logs the commands or display entries of a debug session. If issued with no parameters, the command lists all open log files.

`lo[g] [off] [c | s file_name]`

Parameters

`c`

Command specifier.

`s`

Display-entry specifier.

`file_name`

Name of a log file.

Examples

Table 6.22 shows examples of the `log` command.

Table 6.22 log Command-Line Debugger Command—Examples

`log`	Lists currently opened log files.
`log s session.log`	Logs all display entries to file `session.log`.
`log c command.log`	Logs your commands to file `command.log.`
`log off c`	Stops command logging.
`log off`	Stops all logging to the command-line debugger window.

ls

Lists directory contents.

NOTE You can use the `ls` debug command the same way you use the `dir` operating system command with one exception: You cannot use any option that requires user keyboard input (such as `/p` for the `dir` operating system command). The same is true of the `dir` command.

Examples

Table 6.23 shows examples of the `ls` command.

Table 6.23 ls Command-Line Debugger Command—Examples

`ls`	Lists all files of the current directory.
`ls *.txt`	Lists all current-directory files that have the `.txt` filename extension.
`ls c:/tmp`	Lists all files in the `tmp` directory of the `C:` drive.
`ls /ad`	Lists only the subdirectories of the current directory.

next

Steps over subroutine invocations.

```
n[ext]
```

Remarks

If you execute the `next` command interactively, the command returns immediately, and target-program execution starts. Then you can wait for execution to stop (for example, due to a breakpoint) or type the `stop` command.

If you execute the `next` command in a script, the command-line debugger polls until the debugger stops (for example, due to a breakpoint). Then the command-line debugger executes the next command in the script. If this polling continues indefinitely because debugging does not stop, press the **Esc** key to stop the script.

pwd

Displays the working directory.

```
pwd
```

radix

Shows or changes the default input radix (number base) for command entries and display of registers and memory locations. Entering this command without any parameter values displays the current default radix.

```
r[adix] [b | d | f | h | u]
        [ register | reg_block | addr_group ]...
```

Arguments

`Display format — b (binary), d (decimal), f (fraction), h (hexadecimal), or u (unsigned).`

`register`

Register name.

`reg_block`

Names of the first and last registers of a block, specified as *register_first*`..`*register_last*

`addr_group`

One of these memory-address specifications:

- A single address
- First and last block addresses, in the form *address_first*`..`*address_last*
- First address and number of locations, in the form *address#count*

Remarks

The factory default radix is hexadecimal, but you can override this default by typing the appropriate radix specifier before a constant. These specifiers are:

- hexadecimal — dollar sign ($)
- decimal — left quote (‘)
- binary — percent sign (%)
- fraction — caret (^)

To avoid typing many of the same specifier characters, use the `radix` command to change the default radix.

Examples

Table 6.24 shows examples of the `radix` command.

Table 6.24 radix Command-Line Debugger Command—Examples

`radix`	Shows the current default radix.
`radix D`	Changes the default input radix to decimal.
`radix H`	Changes the default input radix to hexadecimal.
`radix f r0..r7`	Changes the display radix for the specified registers to fraction.
`radix d x:0#10 r1`	Changes the display radix for the specified register and memory blocks to decimal.

restart

Restarts the debug session.

```
[re]start
```

run

Executes a Tcl script file block by block.

ru[n] *file_name*

Parameter

file_name

Name of the file to be executed

Remarks

You can use the `run` command to execute a script that includes any of the commands `load`, `close`, `debug`, `kill`, or `run`. However, such commands must not be in a loop or other such block.

Example

Execute file `test.tcl`:

```
run test.tcl
```

save

Saves the contents of memory locations to a binary file or a text file containing hexadecimal values.

```
sa[ve] -h | -b addr_block ... filename [-a | -c | -o]
```

Parameters

`-h`

Instructs the debugger to write information in hexadecimal. This option includes memory location information in the file. As a result, when loading a hexadecimal file, you do not have to specify its address.

`-b`

Instructs the debugger to write information in binary format. This option does *not* include memory location information in the file. As a result, if you load a binary file, you must specify its address.

`addr_block`

Memory location block, specified in either of two ways:

- First and last addresses of the block, in the form *address_first*`..`*address_last*
- First address and number of locations, in the form *address#count*

`-a`

Append specifier. Instructs the command-line debugger to append the saved memory contents to the current contents of the specified file.

`-c`

No-overwrite specifier. Instructs the debugger to cancel the save command if the specified file already exists, thereby preventing changes to this file's contents.

`-o`

Overwrite specifier: tells the debugger to overwrite any existing contents of the specified file.

Remarks

You can use the Tcl `set` command to name a particular block of memory, then use that name in the save command.

Examples

Table 6.25 shows examples of the `save` command.

Table 6.25 save Command-Line Debugger Command—Examples

set addressBlock1 "p:10..`31" set addressBlock2 "p:10000#20" save -h $addressBlock1 $addressBlock2 hexfile -a	Dumps contents of two memory blocks to the text file hexfile.lod (in append mode).
set addressBlock1 "p:10..`31" set addressBlock2 "p:10000#20" save -b $addressBlock1 $addressBlock2 binfile -o	Dumps contents of two memory blocks to the binary file binfile.lod (in overwrite mode).

step

Steps through a program, automatically executing the display command.

```
st[ep] [li | in | into | out]
```

Parameters

li

One line specifier.

in

One instruction specifier.

into

Step into specifier.

out

Step out specifier.

Examples

Table 6.26 shows examples of the step command.

Table 6.26 step Command-Line Debugger Command—Examples

step li	Steps one line.
step in	Steps one instruction.
step into	Steps into a function.
step out	Steps out of a function.

stop

Stops a running program (started by a `go`, `step`, or `next` command).

`s[top] [all]`

Parameter

`all`

Specifies all target programs (of multiple cores).

Examples

Table 6.27 shows examples of the `stop` command.

Table 6.27 kill Command-Line Debugger Command—Examples

`stop`	Stops the currently running target program.
`stop all`	Stops all currently running target programs (of multiple cores).

switchtarget

For multi-core or multi-chip debugging, lists the available debug sessions or specifies the session for your debug commands.

`sw[itchtarget] [`*`index`*`]`

Parameter

`index`

Session index number.

Examples

Table 6.28 shows examples of the `switchtarget` command.

Table 6.28 switchtarget Command-Line Debugger Command—Examples

`switchtarget`	Lists current debug sessions.
`switchtarget 0`	Specifies debug session `0` as the recipient of subsequent debug commands.

system

Executes a system command.

`sy[stem] system_command`

Parameter

system_command

Any system command that does not use a full screen display.

Remarks

The command-line debugger supports system commands that require keyboard input. But the command-line debugger does not support the DOS edit command or other system commands that use the full screen display.

Example

Delete from the current directory all files that have the `.tmp` filename extension:

```
system del *.tmp
```

view

Specifies assembly or register view mode. Entering this command without a parameter value toggles the mode.

`v[iew] [a | r] [address]`

Parameter

`a`

Assembly specifier.

`r`

Register specifier.

address

Memory address.

Examples

Table 6.29 shows examples of the `view` command.

Table 6.29 view Command-Line Debugger Command—Examples

`view`	Toggles view mode.
`view a`	Sets assembly view mode.
`view r`	Sets register view mode.
`view a $100`	Displays the assembly language instructions that begin at hexadecimal address `100`.

wait

Tells the debugger to wait for a specified amount of time, or until you press the space bar.

`w[ait] [`*`milliseconds`*`]`

Parameter

`milliseconds`

Number of milliseconds to wait.

Examples

Table 6.30 shows examples of the `wait` command:

Table 6.30 wait Command-Line Debugger Command—Examples

`wait`	Debugger waits until you press the space bar.
`wait 2`	Debugger waits for two milliseconds.

watchpoint

Sets, removes, or displays a watchpoint.

NOTE Due to hardware limitations, you can set just one watchpoint at a time.

`wat[chpoint] [`*`variable_name`*` | `*`watchpoint_id`*` off]`

Parameter

`variable_name`

A program variable.

`watchpoint_id`

Watchpoint identifier.

`off`

Remove specifier.

Examples

Table 6.31 shows examples of the `watchpoint` command.

Table 6.31 watchpoint Command-Line Debugger Command—Examples

`watchpoint`	Displays the watchpoint.
`watchpoint i`	Sets a watchpoint on variable `i`.
`watchpoint tally off`	Removes watchpoint from variable `tally`.

Manipulating Target Memory

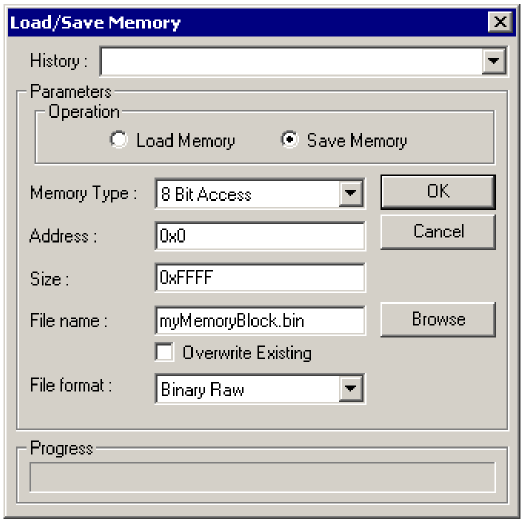
The Debug menu provides two commands that let you manipulate target memory while you are debugging. These commands are:

- 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. (See Figure 6.9.)

Figure 6.9 Load/Save Memory Dialog Box



Load/Save Memory

History :

Parameters

Operation

○ Load Memory ● Save Memory

Memory Type : 8 Bit Access

Address : 0x0

Size : 0xFFFF

File name : myMemoryBlock.bin

□ Overwrite Existing

File format : Binary Raw

Progress

History

The History listbox lists all previous load and save memory operations. Select a previous load or save operation to repeat the action.

Operation

The Operation radio buttons let you select between load operations and save operations.

Memory Type

The Memory Type listbox lets you select the size of the memory units. You can select:

- 8-bit access
- 16-bit access
- 32-bit access

Address

The Address text box lets you specify the memory address where you want to start loading or saving memory.

Size

The Size text box lets you specify the size in bytes of the memory region you want to load or save.

Filename

The Filename text box lets you specify the file you wish to use for the desired memory operation.

Overwrite Existing

The Overwrite Existing checkbox lets you specify that you wish to overwrite any existing files. This option is only available when you are performing Save operations.

File Formats

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

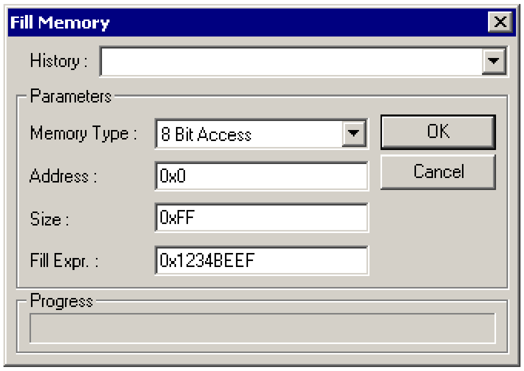
  A text file in which each memory unit is represented by a hexadecimal value.
- Text Unsigned Decimal

  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 with a given value, select **Debug > Fill Memory**. The **Fill Memory** dialog box appears. (See Figure 6.10.)

Figure 6.10 Fill Memory Dialog Box



Memory Type : 8 Bit Access
Address : 0x0
Size : 0xFF
Fill Expr. : 0x1234BEEF

History

The History listbox lists all the previous fill operations. Select a previous fill operation to repeat the action.

Memory Type

The Memory Type listbox lets you select the size of the memory units. You can select:

- 8-bit access
- 16-bit access
- 32-bit access

Address

The Address text box lets you specify the memory address at which to start the filling target memory.

Size

The Size text box lets you specify the size (in bytes) of the memory region to fill.

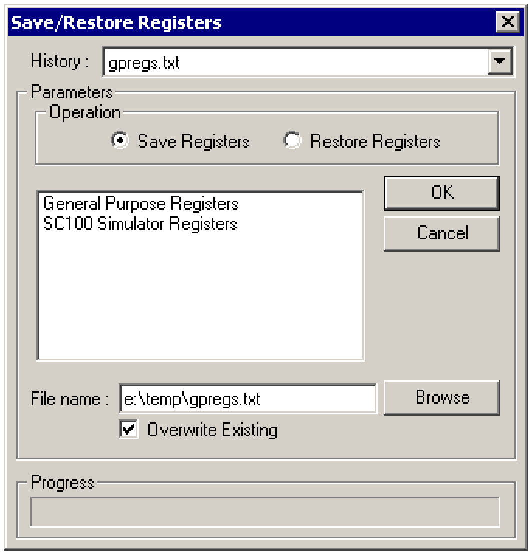
Fill Expr

The Fill Expr text box lets you specify the value (in hexadecimal) with which to fill the memory region.

Save Restore Registers

The **Debug > SaveRestoreRegs** option (Figure 6.11) lets you save or restore the values of register banks while you are debugging.

Figure 6.11 Save/Restore Registers Dialog Box



Save/Restore Registers

History : gpregs.txt

Parameters

Operation

● Save Registers ○ Restore Registers

General Purpose Registers
SC100 Simulator Registers

File name : e:\temp\gpregs.txt

☑ Overwrite Existing

Progress

History

The History listbox lists all previous save and restore operations. Select a previous save or restore operation to repeat the action.

Operation

The Operation radio buttons let you select between load operations and save operations.

Register List

The register list lets you select the register banks that you want to save. You may select more than one register bank. The register list is only available for save operations.

Filename

The Filename text box lets you specify the file you wish to use for the desired save or restore operation.

Overwrite Existing

The Overwrite Existing checkbox lets you specify that you wish to overwrite any existing files. This option is only available for save operations.

7

Multi-Core Debugging

This chapter explains how to use the CodeWarrior™ debugger's multi-core debugging capabilities.

Multi-core debugging lets you debug multiple StarCore® DSP 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.

The sections of this chapter are:

- Setting Up to Debug Multiple Cores
- JTAG Initialization File
- Debugging Multiple Cores
- Using Multi-Core Debugging Commands
- Synchronized Stop

Setting Up to Debug Multiple Cores

To set up for multi-core debugging, follow these steps:

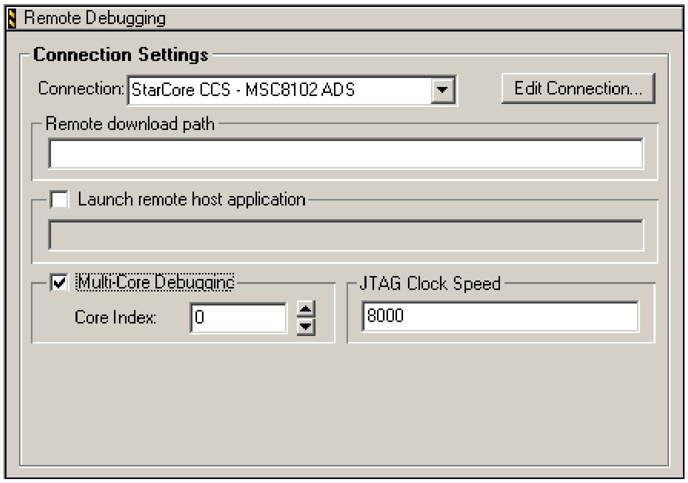
1. Set up and connect your JTAG chain.

NOTE This chain can consist of multiple boards or multiple chips on the same board.

2. Create a JTAG initialization file that describes the items on the JTAG chain.
3. Open the CodeWarrior project to be debugged.

NOTE If you are debugging more than one core, each core must have its own project.

4. In the **Remote Debugging** target settings panel, enable the Multi-Core Debugging option and specify the core index.



Remote Debugging

Connection Settings

Connection: StarCore CCS - MSC8102 ADS

Remote download path

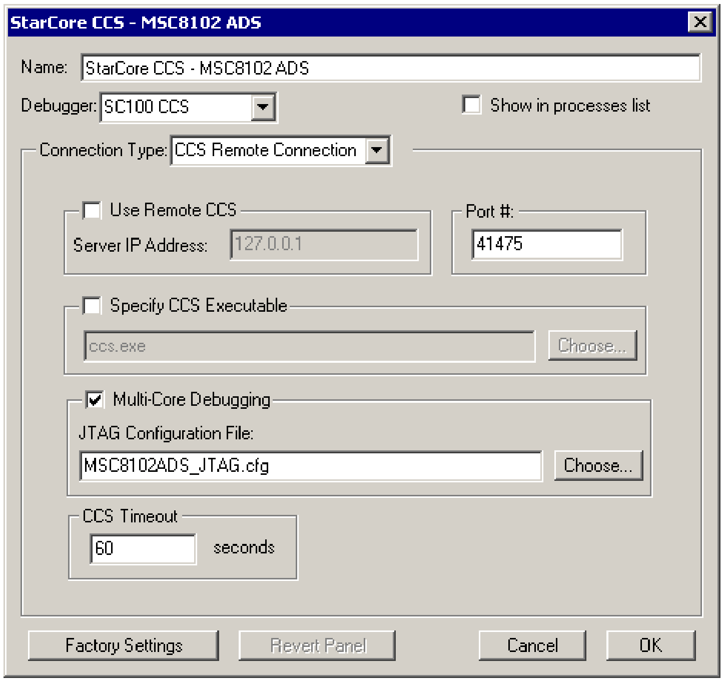
□ Launch remote host application

☑ Multi-Core Debugging

Core Index: 0

JTAG Clock Speed: 8000

5. Click **Edit Connection** to display the remote connection configuration dialog box.



StarCore CCS - MSC8102 ADS

Name: StarCore CCS - MSC8102 ADS

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: MSC8102ADS_JTAG.cfg

CCS Timeout: 60 seconds

6. Check the Multi-Core Debugging checkbox.

   The JTAG Configuration File text box enables.

7. Use the JTAG Configuration File text box to enter the name of the JTAG configuration file created above.

NOTE Depending on the project being debugged and the stationery used to create this project, you may need to change additional target settings. This section discusses just those target settings related to multi-core debugging.

8. Select **Project > Run**

   The IDE downloads the program to the specified core. You now debug your program.

JTAG Initialization File

To debug multiple cores connected in a JTAG chain, you must create a JTAG initialization file that specifies the type and the chain order of the cores to be debugged.

If you are using a StarCore chip that has just one core, you must use `SC140` as the chip name in your JTAG initialization file. Listing 7.1 shows the JTAG initialization file required to debug three, single-core StarCore chips connected in a JTAG chain.

Listing 7.1 Example JTAG Initialization File for Three Single-Core StarCore® Chips

```
# JTAG Initialization File

SC140    # JTAG chain index is 0
SC140    # JTAG chain index is 1
SC140    # JTAG chain index is 2
```

You can also debug multiple cores on a single StarCore chip. The MSC8102 chip, for example, has four cores on a single chip. Listing 7.2 shows the JTAG initialization file required to debug each core on an MSC8102 chip that appears first on a JTAG chain.

NOTE The CodeWarrior debugger considers each MSC8102 to have 5 cores.

NOTE The device type MSC8102Sync represents the chip as a whole. A device of this type can respond to only a subset of the multi-core debugger commands (such as synchronized stop and the command that puts the chip into DSI boot mode).

Listing 7.2 Example JTAG Initialization File for a Single MSC8102 Chip

```
# JTAG Initialization File

# Indicates the first (and only) 8102 on the chain
MSC8102Sync # JTAG chain index is 0
MSC8102     # JTAG chain index is 1
MSC8102     # JTAG chain index is 2
MSC8102     # JTAG chain index is 3
MSC8102     # JTAG chain index is 4
```

You also can to debug multiple cores on multiple MSC8102 chips. Listing 7.3 shows the JTAG initialization file required to debug each core on two MSC8102 chips that appear first and second in a JTAG chain.

Listing 7.3 Example JTAG Initialization File for Two MSC8102 Chips

```
# JTAG Initialization File

# Indicates the first 8102 on the JTAG chain
MSC8102Sync # JTAG chain index is 0
MSC8102     # JTAG chain index is 1
MSC8102     # JTAG chain index is 2
MSC8102     # JTAG chain index is 3
MSC8102     # JTAG chain index is 4

# Indicates the second 8102 on the JTAG chain
MSC8102Sync # JTAG chain index is 5
MSC8102     # JTAG chain index is 6
MSC8102     # JTAG chain index is 7
MSC8102     # JTAG chain index is 8
MSC8102     # JTAG chain index is 9
```

Finally, you can include entries for other devices (StarCore and non-StarCore) connected to a JTAG chain by adding an entry of this form to your JTAG initialization file:

`Generic` *`instruct_reg_len data_reg_bypass_len JTAG_bypass_instruct`*

Table 7.1 shows the definitions of the variables that you must specify for a generic device.

Table 7.1 Syntax Variables to Specify a Generic Device on a JTAG Chain

Variable	Description
instruct_reg_len	Length (in bits) of the JTAG instruction register.

Table 7.1 Syntax Variables to Specify a Generic Device on a JTAG Chain

Variable	Description
data_reg_bypass_len	Length (in bits) of the JTAG bypass register.
JTAG_bypass_instruct	Value of the JTAG bypass instruction (in hexadecimal).

Listing 7.4 shows a JTAG initialization file for a a StarCore chip and a generic device connected in a JTAG chain.

Listing 7.4 JTAG Initialization File for a JTAG Chain that Includes a Generic Device

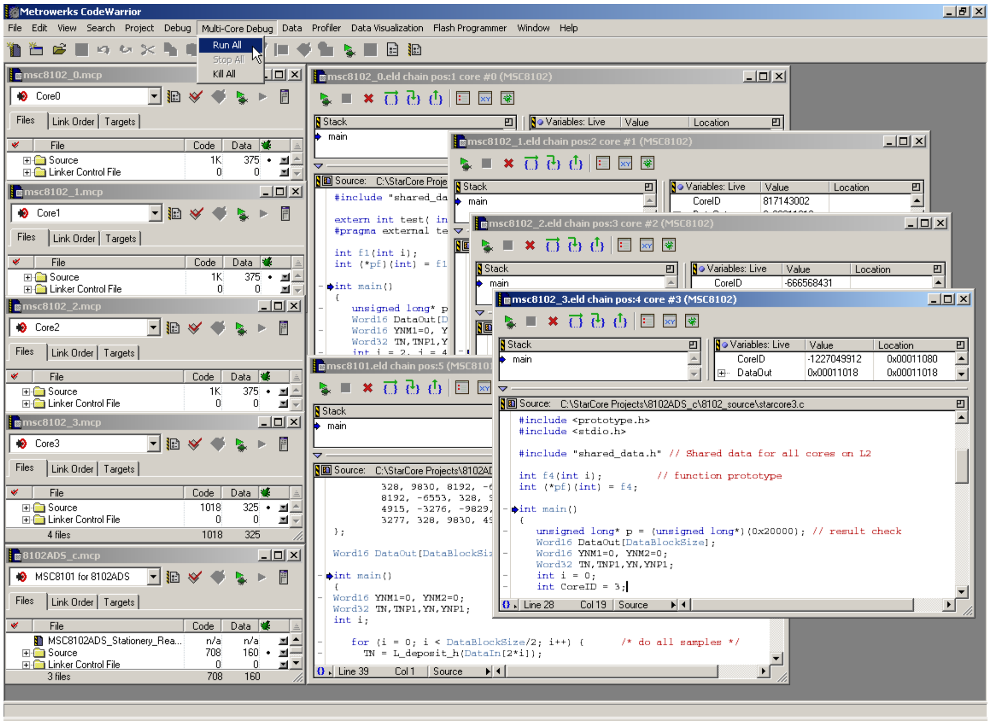
```
# JTAG Initialization File

SC140              # JTAG chain index is 0
Generic 4 1 Oxf   # JTAG chain index is 1
```

Debugging Multiple Cores

When you start to debug a multi-core project, the CodeWarrior debugger downloads each build target to the appropriate core (with correct settings for multi-core debugging).

Figure 7.1 shows an initial download of a multi-core project created from a multi-core stationery (MSC8102ADS). As the figure shows, the IDE displays a separate debugging window for each project in the multi-core project.

Figure 7.1 Debugger Windows After Initial Download of Multi-Core MSC8102ADS 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
msc8102_0.mcp
msc8102_1.mcp
msc8102_2.mcp
msc8102_3.mcp
8102ADS_c.mcp
Files Link Order Targets
msc8102_0.eld chain pos:1 core #0 (MSC8102)
msc8102_1.eld chain pos:2 core #1 (MSC8102)
msc8102_2.eld chain pos:3 core #2 (MSC8102)
msc8102_3.eld chain pos:4 core #3 (MSC8102)
msc8101.eld chain pos:5 (MSC8101)
Stack
main
Variables: Live
Value
Location
Line 28 Col 19 Source
Line 39 Col 1 Source

File	Code	Data
MSC8102ADS_Stationery_Rea...	n/a	n/a
Source	708	160
Linker Control File	0	0
3 files	708	160

Variables: Live	Value	Location
CoreID	-1227049912	0x00011080
DataOut	0x00011018	0x00011018

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;
```

```
};

Word16 DataOut[DataBlockSiz

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]);
```

To debug multiple cores, follow these steps:

1. Use the simulator projects to create your own applications to debug, adding and deleting files and 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, choose **Project > Debug**

   The debugger downloads your multi-core projects to the simulator.

4. Debug 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 7.2 lists and defines the affect of each multi-core debugging command available in the Multi-Core Debug menu.

Table 7.2 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.

In addition to the menu commands listed above, you may also find these debugger initialization file commands useful when debugging multiple cores:

- CCSConfigTemplate
- CCSCoreRunMode
- CCSStopCore

Synchronized Stop

If you perform multi-core debugging using the MSC8102 simulator, the debugger offers an additional feature called synchronized stop.

Synchronized stop means that when any of the executing cores stops (for example, because the core encounters a software breakpoint or because you issue an explicit stop command), execution on all other cores stop as well.

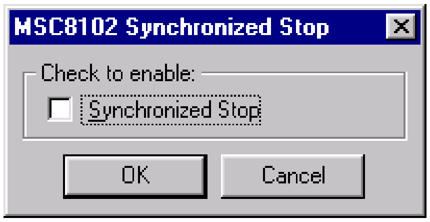
Before you can use the synchronized stop feature, you must enable it. Enabling this feature sets bit 10 and bit 15 of the ESEL_DM register. Disabling this feature clears those bits.

To enable synchronized stop, follow these steps:

1. Start debugging a multi-core project.
2. Choose **SC100 > MSC8102 Sim/ADS > MSC8102 Sync Stop**.

   The **MSC8102 Synchronized Stop** dialog box appears. (See Figure 7.2.)

Figure 7.2 MSC8102 Synchronized Stop Dialog Box



MSC8102 Synchronized Stop

Check to enable:

□ Synchronized Stop

3. Check the Check to enable checkbox.
4. Click **OK**

8

iCache Performance Tool

This chapter explains how to use the CodeWarrior™ iCache Performance Tool. Use this tool to examine information obtained from an instruction cache dump.

The sections are:

- iCache Performance Tool Input Files
- Starting the iCache Performance Tool
- Loading and Displaying iCache Data
- iCache Performance Window Toolbar
- Viewing and Analyzing iCache Data

iCache Performance Tool Input Files

The iCache Performance tool uses these types of files to generate performance information from the contents of a core's instruction cache:

- An executable file
- An instruction cache trace buffer file (dump file)

You can load data from an instruction cache trace buffer file that contains data for one core or from an instruction cache trace buffer file that contains data for four cores.

To generate an executable file for use with the iCache Performance tool, modify your linker command file so the linker places instructions in cacheable memory.

To create a dump file for use with the ICache Performance tool, issue commands similar to those in Listing 8.1 to the MSC8102 simulator.

The MSC8102 simulator is in this directory:

`installDir\StarCore Support\compiler\bin\simscsc100`

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

Listing 8.1 Simulator Commands that Generate an Instruction Cache Trace Buffer File

```
device dv0 msc8102
reset d m1
core a
load core0.eld
```

```
log eqbs core.dmp
break __dhalt
go
log off
quit
```

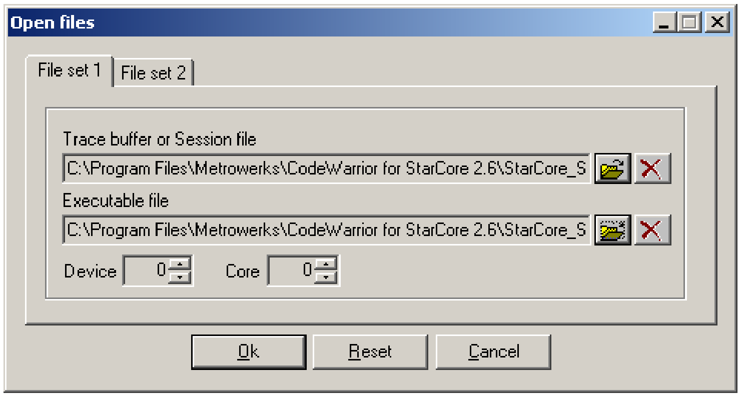
Once you enter the appropriate simulator commands, the simulator generates an instruction cache trace buffer file. You can then quit the simulator.

Starting the iCache Performance Tool

To start the iCache Performance tool, choose **View > iCache Performance**

The IDE displays the **Open Files** window. (See Figure 8.1.)

Figure 8.1 Open Files Window



Trace buffer or Session file
C:\Program Files\Metrowerks\CodeWarrior for StarCore 2.6\StarCore_S
Executable file
C:\Program Files\Metrowerks\CodeWarrior for StarCore 2.6\StarCore_S
Device 0
Core 0

Loading and Displaying iCache Data

Using the **Open Files** window, you can load data for one or more file sets (that is, for one or more pairs of `.dmp` and `.eld` files). To do this, follow these steps:

1. In the **Open Files** window, click the **File set 1** tab.
2. In the Trace buffer or Session file field, select the trace buffer dump file that contains the data for the first core you want to examine.

   Use the folder 📂 button to display a dialog box you can use to select the dump file.

NOTE You can select an ICache Performance Tool data file (`.icp` file) for this field. If you do, skip step 3 because you do not have to specify an executable file.

3. In the Executable file field, select the executable file used to produce the data captured in the dump file.

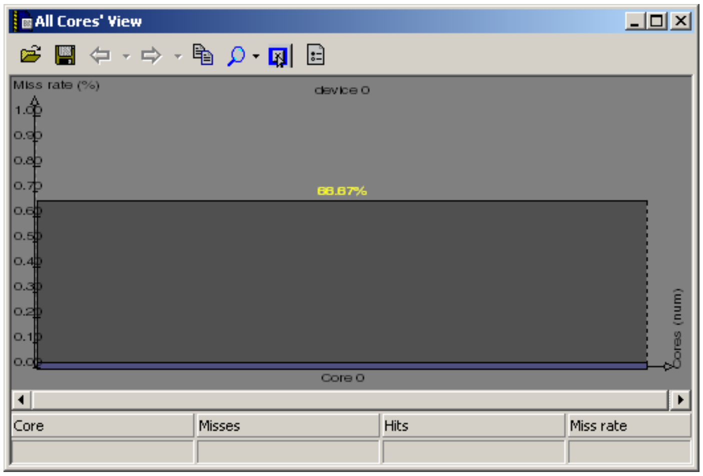
   Use the folder button to display a dialog box that you can use to select the executable file.

   A second tab, labeled **File set 2**, appears in the **Open Files** window.

4. Use the Device spin control to specify the JTAG chain position of the StarCore device for which the selected dump file contains data.
5. Use the Core spin control to specify the JTAG chain position of the core for which the selected dump file contains data.
6. If you want to load data from a second file set, click the **File set 2** tab and go to step 2.
7. Click **OK**

   The iCache Performance tool loads the specified data and displays it in the **ICache Performance** window. This window can display various views of the raw instruction cache data. The default view is the **All Cores** view. (See Figure 8.2.)

Figure 8.2 The All Cores View of the ICache Performance Window



Core	Misses	Hits	Miss rate

At this point, you can use the **ICache Performance** window to analyze the instruction cache data.

iCache Performance Window Toolbar

Once you have loaded and displayed data in the **iCache Performance** window, you can use the buttons of this window's toolbar to manipulate the data. Table 8.1 lists and describes each button in this toolbar.

Table 8.1 iCache Performance Window Toolbar Buttons

Button	Description
📂	Displays the **Open Files** window in which you can specify a new set of files with which to work.
💾	Displays the **Save As** dialog box. Use this dialog box to specify the name and path of the file in which to save the ICache performance data currently loaded.
⇦	The **ICache Performance** window maintains a list of the views you examine in the order in which you viewed them. Go Back displays the previous view in this list, that is, the view displayed before the current view.
⇨	The **ICache Performance** window maintains a list of the views you examine in the order in which you viewed them. Go Forward displays the next view in this list, that is, the view displayed after the current view.
⎘	Creates and displays a copy of the current **iCache Performance** window.
🔍▾	Displays a dropdown list from which you can select the magnitude of the y-axis for the current view.
☒	Closes the view currently displayed in the **iCache Performance** window.
☰	Displays a dialog box that lets you select the fonts and colors the **iCache Performance** window uses.

Viewing and Analyzing iCache Data

You can use the **ICache Performance** window to view the raw instruction cache data in various ways. You can cycle through each available view by double-clicking in the graph portion of the window.

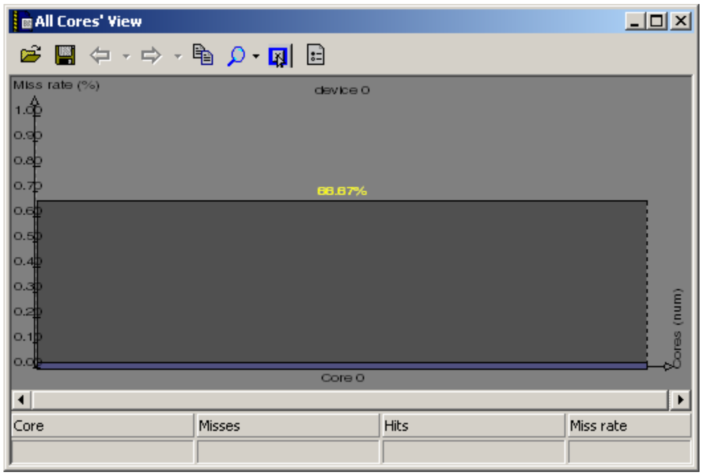
The available views are:

- All Cores View
- Core View
- Function View
- PC View

All Cores View

The **All Cores** view presents information about all cores for which you loaded instruction cache data. There is one column to each core. (See Figure 8.3.)

Figure 8.3 All Cores View of ICache Performance Window



Core	Misses	Hits	Miss rate

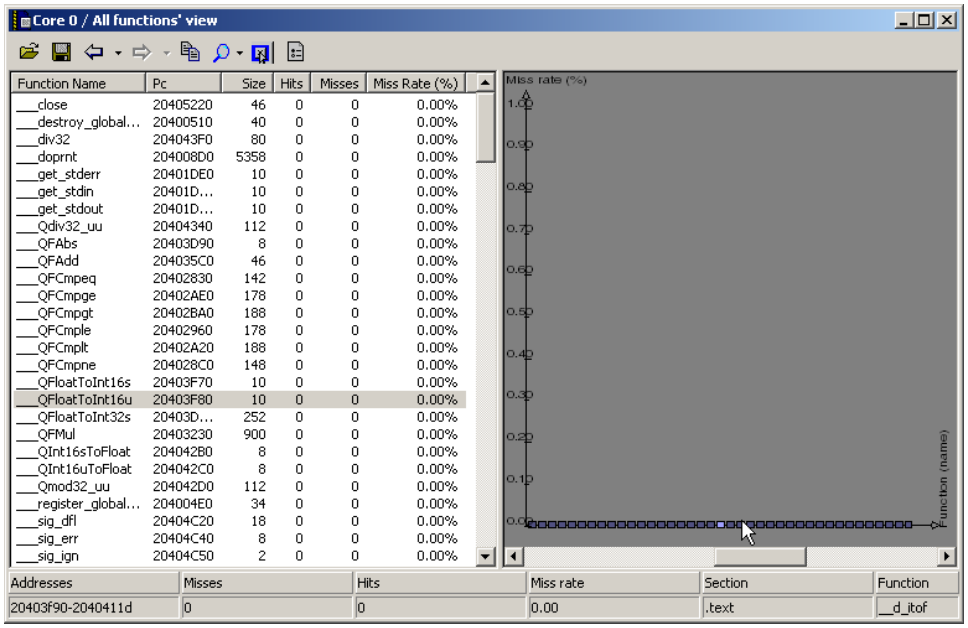
The bottom panel of the **All Cores** view displays useful information when the mouse cursor passes over different parts of the view.

Double-clicking on a bar displays the next view (Core).

Core View

The **Core** view (Figure 8.4) presents information about the iCache data for the specified core (one bar per function). The function can be a regular function or executable code that is part of a function.

Figure 8.4 Core View of ICache Performance Window



Core 0 / All functions' view

Function Name	Pc	Size	Hits	Misses	Miss Rate (%)
___close	20405220	46	0	0	0.00%
___destroy_global…	20400510	40	0	0	0.00%
___div32	204043F0	80	0	0	0.00%
___doprnt	204008D0	5358	0	0	0.00%
___get_stderr	20401DE0	10	0	0	0.00%
___get_stdin	20401D…	10	0	0	0.00%
___get_stdout	20401D…	10	0	0	0.00%
___Qdiv32_uu	20404340	112	0	0	0.00%
___QFAbs	20403D90	8	0	0	0.00%
___QFAdd	204035C0	46	0	0	0.00%
___QFCmpeq	20402830	142	0	0	0.00%
___QFCmpge	20402AE0	178	0	0	0.00%
___QFCmpgt	20402BA0	188	0	0	0.00%
___QFCmple	20402960	178	0	0	0.00%
___QFCmplt	20402A20	188	0	0	0.00%
___QFCmpne	204028C0	148	0	0	0.00%
___QFloatToInt16s	20403F70	10	0	0	0.00%
___QFloatToInt16u	20403F80	10	0	0	0.00%
___QFloatToInt32s	20403D…	252	0	0	0.00%
___QFMul	20403230	900	0	0	0.00%
___QInt16sToFloat	204042B0	8	0	0	0.00%
___QInt16uToFloat	204042C0	8	0	0	0.00%
___Qmod32_uu	204042D0	112	0	0	0.00%
___register_global…	204004E0	34	0	0	0.00%
___sig_dfl	20404C20	18	0	0	0.00%
___sig_err	20404C40	8	0	0	0.00%
___sig_ign	20404C50	2	0	0	0.00%

Addresses	Misses	Hits	Miss rate	Section	Function
20403f90-2040411d	0	0	0.00	.text	__d_itof

The left side of the **Core** view lists each function included in the iCache data. For each function listed, these values are displayed:

- PC (program counter)
- Size
- Hits
- Misses
- Miss rate (shown as a percentage)

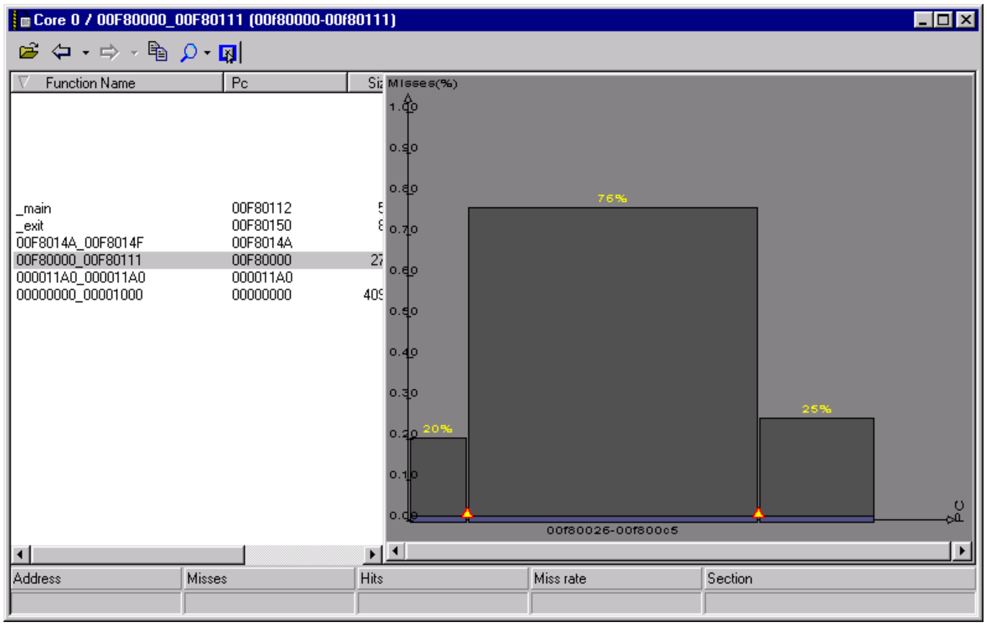
The bottom panel of the **Core** view displays useful information when the mouse cursor passes over different parts of the view.

Double-clicking on a bar displays the next view (Function).

Function View

The **Function** view (Figure 8.5) presents information for a single function. The view displays one bar for each interval (where an interval represents contiguous code between calls to other functions).

Figure 8.5 Function View of the ICache Performance Window



Core 0 / 00F80000_00F80111 (00f80000-00f80111)

Function Name	Pc	Si
_main	00F80112	5
_exit	00F80150	8
00F8014A_00F8014F	00F8014A	
00F80000_00F80111	00F80000	27
000011A0_000011A0	000011A0	
00000000_00001000	00000000	409

Address	Misses	Hits	Miss rate	Section

The triangles represent function calls. Double-clicking on a triangle (or above a triangle) displays the **Function** view of the called function.

The left side of the **Function** view lists all functions. For each function listed, these values are displayed:

- PC (program counter)
- Size
- Hits
- Misses
- Miss rate (shown as a percentage)

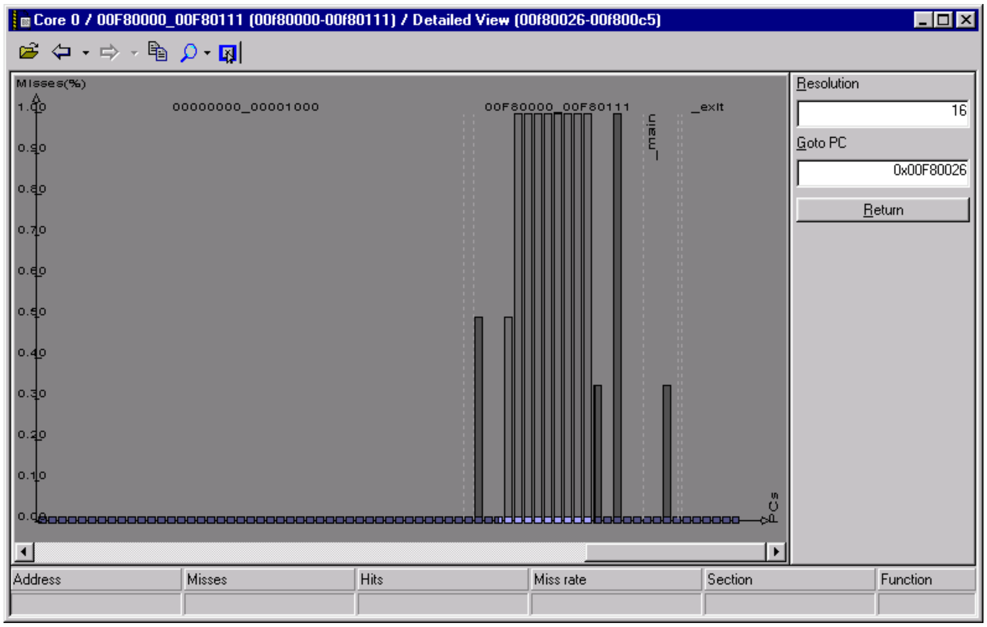
The bottom panel of the **Function** view displays useful information when a mouse passes over different parts of the view.

Double-clicking on a bar displays the next view (the **PC** view).

PC View

The **PC** view displays all code with the specified resolution. (See Figure 8.6.)

Figure 8.6 PC View of the ICache Performance Window



Core 0 / 00F80000_00F80111 (00f80000-00f80111) / Detailed View (00f80026-00f800c5)

Resolution: 16

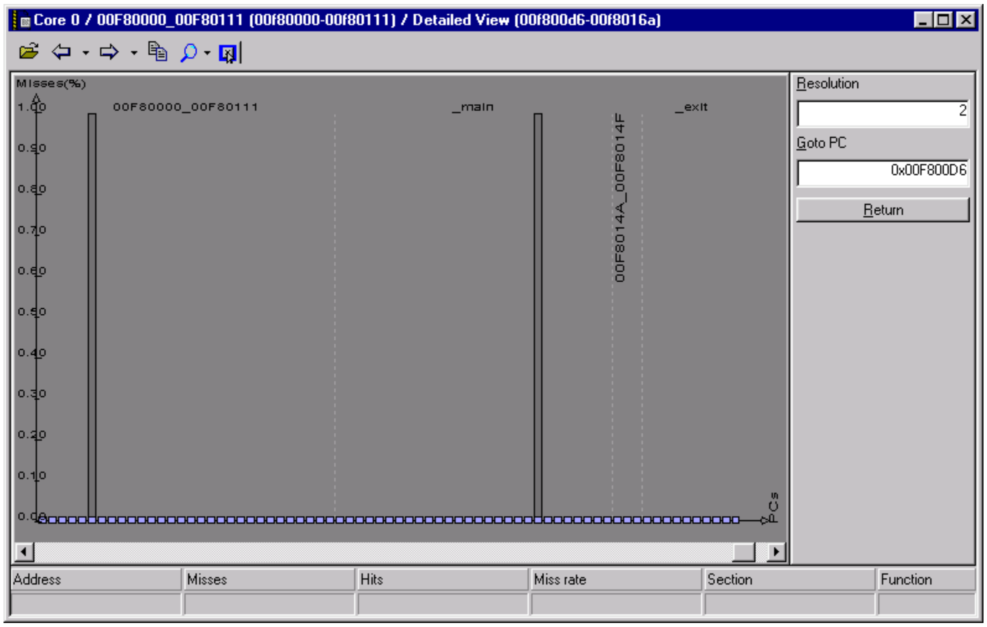
Goto PC: 0x00F80026

Address	Misses	Hits	Miss rate	Section	Function

You can select a different resolution in the Resolution text box and a different PC in the Goto PC text box. You can return to the starting position by clicking **Return**.

You can select an interval of the graph by pressing the left mouse button, dragging over the interval of interest, and releasing the left mouse button. Once you release the left mouse button, the **ICache Performance** window displays a new graph that shows the selected interval. (See Figure 8.7.)

Figure 8.7 New PC View After Selecting an Interval



Core 0 / 00F80000_00F80111 (00f80000-00f80111) / Detailed View (00f800d6-00f8016a)

Resolution: 2

Goto PC: 0x00F800D6

Address	Misses	Hits	Miss rate	Section	Function

Double-clicking on a bar in the **PC** view displays the **Function** view for the function that contains the PC value clicked.

9

Enhanced On-Chip Emulation (EOnCE)

This chapter explains how to use the CodeWarrior™ EOnCE Configurator.

The EOnCE Configurator lets you use the StarCore® DSP's on-chip EOnCE module from within the CodeWarrior IDE. The EOnCE module allows non-intrusive interaction with a StarCore chip's core and lets you examine the contents of registers, memory, and on-chip peripherals in a special debugging environment.

The sections of this chapter are:

- EOnCE Features
- EOnCE Configurator Panel Descriptions
- EOnCE Example: Counting Factorial Function Calls
- EOnCE Example: Using the Trace Buffer

EOnCE Features

With the EOnCE Configurator, you can keep a running trace of tasks and interrupts and 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 that with a regular software breakpoint, the program halts immediately *before* the breakpoint instruction; however, with an EOnCE breakpoint, execution 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 is accessible either directly by your software or by the CodeWarrior debugger
- Does not require that peripherals be halted during debug mode

EOnCE Trace Buffer Overview

The following information is pertinent when using the EOnCE trace buffer:

- The trace buffer is a circular buffer. When the buffer is full, if you continue to step through code, the buffer is overwritten from the beginning.
- You can determine whether the trace buffer is full by examining the TBFULL bit of the ESR (EOnCE Status Register) register. When the trace buffer is full, the TBFULL bit is set.
- You can trace up to 2048 bytes worth of addresses in the trace buffer.
- You must enable the trace buffer each time before getting new trace information.

EOnCE Configurator Panel Descriptions

This section describes each EOnCE Configurator panel. You use these panels to configure debugging using the on-chip EOnCE module.

NOTE When selecting settings in the EOnCE Configurator, configure the tabbed panels in the left-to-right order. For example, configure the **Address Event Detection Channel 0** panel before configuring the **Event Counter** panel. In addition, within a panel, configure your selected settings from the left-top position to the right-bottom position.

You can save settings that you specify in the EOnCE Configurator for your current debugging session only by clicking **OK** in the **EOnCE Configurator** window.

You can save an EOnCE configuration in a file for later reuse by choosing **Debug > EOnCE > Save EOnCE Configuration** and specifying the file name to save to.

You can open a previously saved EOnCE configuration file to use with a project by choosing **Debug > EOnCE > Open EOnCE Configuration** and navigating to the location of the EOnCE configuration file.

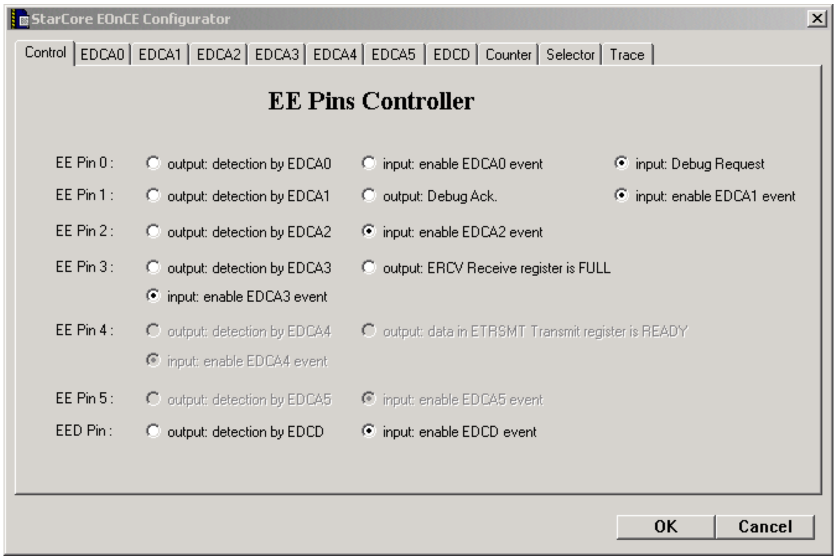
- EE Pins Controller Panel
- Address Event Detection Channel Panels

- Data Event Detection Channel Panel
- Event Counter Panel
- Event Selector Panel
- Trace Unit Panel

EE Pins Controller Panel

Figure 9.1 shows the **EE Pins Controller** panel. Use this panel 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 9.1 EE Pins Controller Panel



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 9.1 lists and defines the settings you can make on the **EE Pins Controller** panel of the EOnCE Configurator.

Table 9.1 EE Pins Controller Panel Description

Panel Item	Description	
EE Pin 0	Three possible settings exist:	
	Setting	**Description**
	output: detection by EDCA0	After an event is detected on EDCA0 (event detection channel 0), the signal on EE pin 0 is toggled.
	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 asserted to EE pin 0 during and after reset causes the core to enter debug mode. A signal asserted to EE pin 0 also causes an exit from stop or wait processing states of the core.
EE Pin 1	Three possible settings exist:	
	Setting	**Description**
	output: detection by EDCA1	After an event is detected on EDCA1 (event detection channel 1), the signal on EE pin 1 is toggled.
	output: Debug Ack.	A signal is asserted to EE pin 1 after the core enters debug mode. A signal is negated to EE pin 1 after the core exits from debug mode.
	input: enable EDCA1 event	An input signal from EE pin 1 enables an event on EDCA1 (event detection channel 1).
EE Pin 2	Two possible settings exist:	
	Setting	**Description**
	output: detection by EDCA2	After an event is detected on EDCA2 (event detection channel 2), the signal on EE pin 2 is toggled.
	input: enable EDCA2 event	An input signal from EE pin 2 enables an event on EDCA2 (event detection channel 2) and ECNT.

Table 9.1 EE Pins Controller Panel Description (*continued*)

Panel Item	Description	
EE Pin 3	Three possible settings exist:	
	Setting	**Description**
	output: detection by EDCA3	After an event is detected on EDCA3 (event detection channel 3), the signal on EE pin 3 is toggled.
	output: ERCV Receive register is full	A signal is asserted to EE pin 3 after the host finishes writing to the ERCV register. A signal is negated to EE pin 3 after the host finishes reading the ETRSMT register.
	input: enable EDCA3 event	An input signal from EE pin 3 enables an event on EDCA3 (event detection channel 3).
EE Pin 5	Not applicable.	
EE Pin 5	Not applicable.	
EED Pin	Two possible settings exist:	
	Setting	**Description**
	output: detection by EDCD	After an event is detected on the EDCD (Data Event Detection channel), the signal on the EED pin is toggled.
	input: enable EDCD event	An input signal from the EED pin enables an event on EDCD (the Data Event Detection channel).

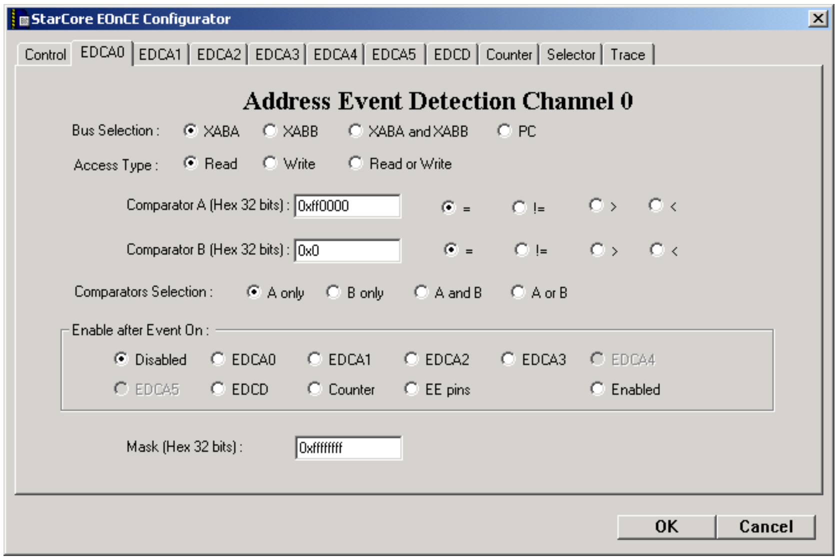
Address Event Detection Channel Panels

The EOnCE module includes several address event detection channels that can detect address values from an address bus according to the selections you choose. Each address event detection channel has a corresponding EOnCE Configurator panel:

- **Address Event Detection Channel 0** panel - (EDCA0)
- **Address Event Detection Channel 1** panel - (EDCA1)
- **Address Event Detection Channel 2** panel - (EDCA2)
- **Address Event Detection Channel 3** panel - (EDCA3)
- **Address Event Detection Channel 4** panel - (EDCA4)
- **Address Event Detection Channel 5** panel - (EDCA5)

Figure 9.2 shows an EOnCE Configurator's address event detection channel 0 panel.

Figure 9.2 Address Event Detection Channel 0 Panel



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) : 0xff0000 ● = ○ != ○ > ○ <

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

Table 9.2 lists and defines the settings you can make on the address channel panels of the EOnCE Configurator.

Table 9.2 Address Channel Panel Description

Panel Item	Description
Bus Selection	The bus on which to detect an address value. You can specify: • XABA • XABB • XABA and XABB • PC For example, to set a breakpoint on an instruction, specify PC, which indicates the value of the program counter.
Access Type	The type of access performed on the specified address. You can specify: • Read • Write • Read or write

Table 9.2 Address Channel Panel Description (*continued*)

Panel Item	Description
Comparator A	Specify a value (in hexadecimal, with a maximum length of 32 bits) with which to compare the detected address value. You can specify these comparison types: • = (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. You can specify the following types of comparisons: • = (equal) • != (not equal) • > (greater than) • < (less than)
Comparators Selection	Choose a value or values with which to compare the detected address value. You can specify one of the following: • A only • B only • A and B • A or B

Table 9.2 Address Channel Panel Description (*continued*)

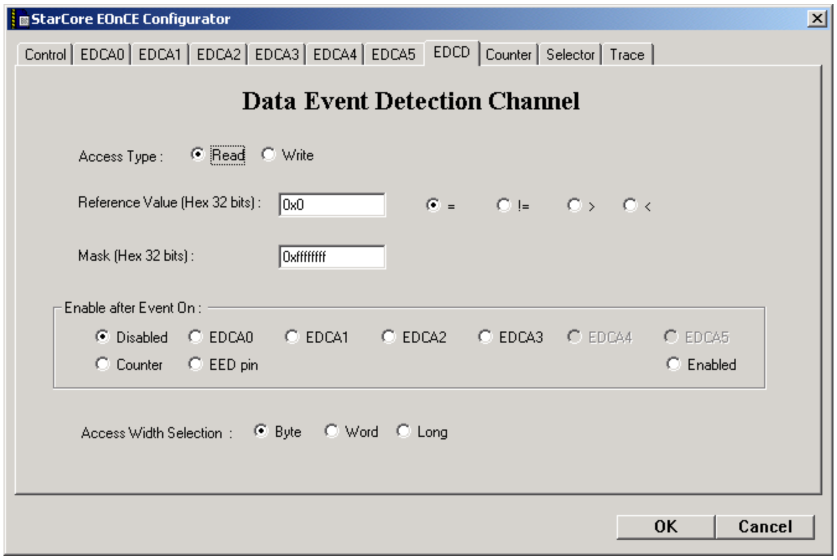
Panel Item	Description
Enable after Event On	Enable the comparison specified by this panel after an event on the specified item. You can specify one of the following: • Disabled • EDCA0, EDCA1, EDCA2, EDCA3 • EDCD • Counter • EE pins • Enabled If you select disabled, the IDE does not perform a comparison on the address. If you select enabled, the IDE performs the specified comparison if an event occurs on any of the items in the list.
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 Panel

You can use the **Data Event Detection Channel** panel to detect a particular data value.

Figure 9.3 shows the **Data Event Detection Channel** panel.

Figure 9.3 Data Event Detection Channel Panel



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 9.3 lists and defines the settings you can make on the **Data Event Detection Channel** panel of the EOnCE Configurator.

Table 9.3 Data Event Detection Channel Panel Description

Panel Item	Description
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 9.3 Data Event Detection Channel Panel Description (*continued*)

Panel Item	Description
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 panel 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.

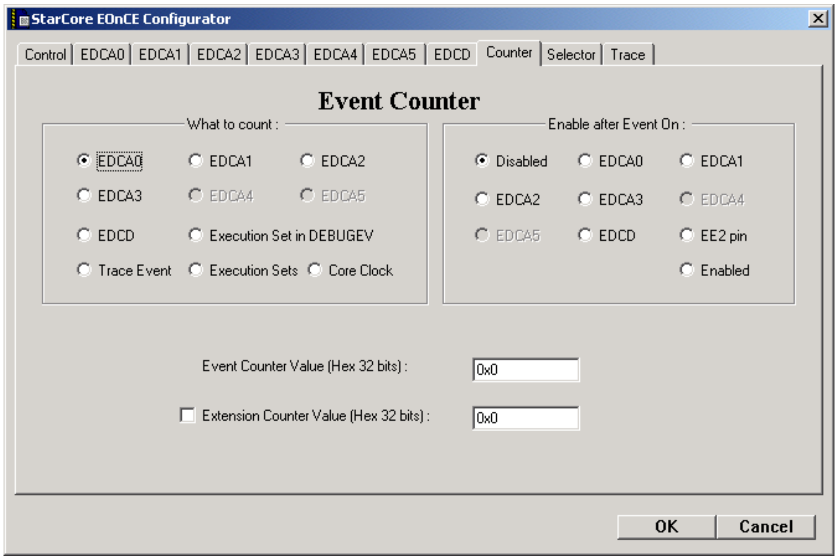
Event Counter Panel

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 9.4 shows the **Event Counter** panel of the EOnCE Configurator.

Figure 9.4 Event Counter Panel



StarCore EOnCE Configurator

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

Event Counter

What to count :

● EDCA0 ○ EDCA1 ○ EDCA2
○ EDCA3 ○ EDCA4 ○ EDCA5
○ EDCD ○ Execution Set in DEBUGEV
○ Trace Event ○ Execution Sets ○ Core Clock

Enable after Event On :

● Disabled ○ EDCA0 ○ EDCA1
○ EDCA2 ○ EDCA3 ○ EDCA4
○ EDCA5 ○ EDCD ○ EE2 pin
○ Enabled

Event Counter Value (Hex 32 bits) : 0x0

□ Extension Counter Value (Hex 32 bits) : 0x0

OK Cancel

Table 9.4 lists and defines the settings you can make on the **Event Counter** panel of the EOnCE Configurator.

Table 9.4 Event Counter Panel Description

Panel Item	Description
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

Table 9.4 Event Counter Panel Description (*continued*)

Panel Item	Description
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 Panel

The **Event Selector** panel 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

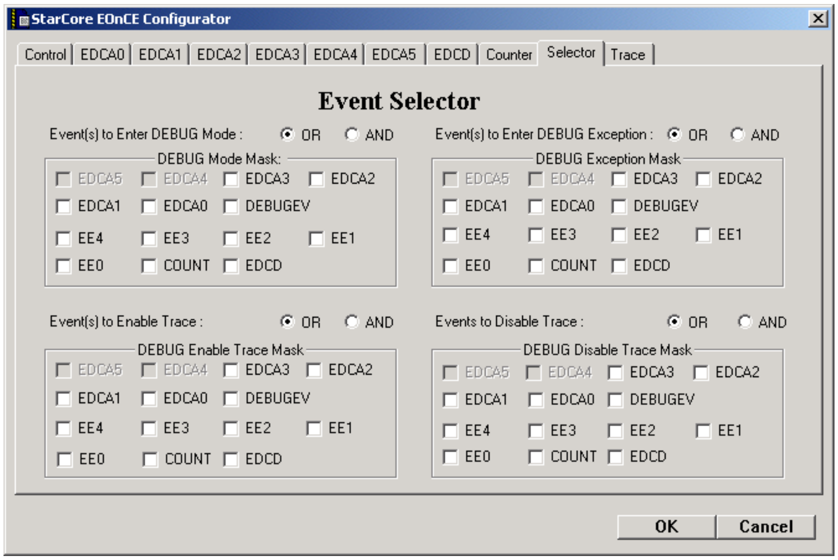
On the **Event Selector** panel, you can specify that after an event occurs on one of the following items, the corresponding debugging action occurs:

- 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 9.5 shows the **Event Selector** panel of the EOnCE Configurator.

Figure 9.5 Event Selector Panel



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 9.5 lists and defines the settings you can make on the **Event Selector** panel of the EOnCE Configurator.

Table 9.5 Event Selector Panel

Panel Item	Description
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.

Table 9.5 Event Selector Panel (*continued*)

Panel Item	Description
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.
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.

Table 9.5 Event Selector Panel (*continued*)

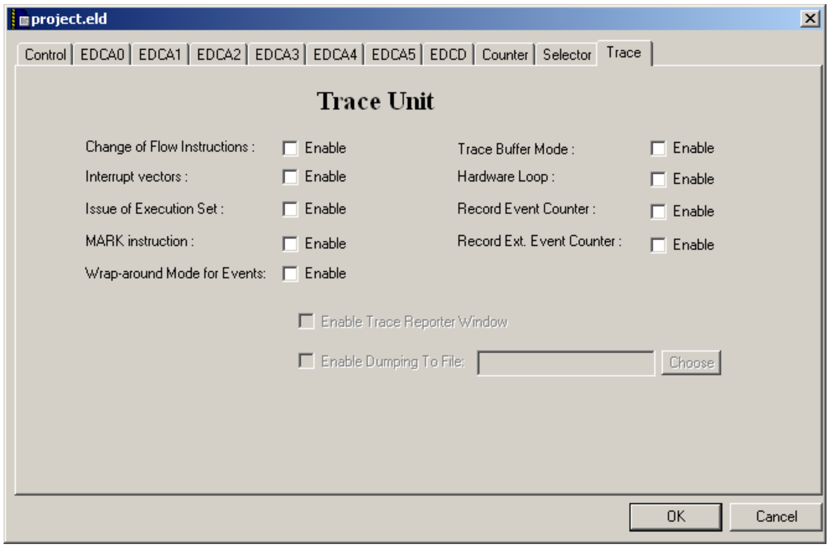
Panel Item	Description
DEBUG Disable Trace Mask	Disable tracing after an event on one or more specified items. You can specify the following items: • EDCA0, EDCA1, EDCA2, EDCA3 • DEBUGEV • EE4, EE3, EE2, EE1, EE0 • Counter • EDCD Click **Any** to specify one item or **All** to specify multiple items.

Trace Unit Panel

With EOnCE, you can collect data in a trace buffer as you debug a program. You can use the **Trace Unit** panel to choose the trace buffer settings.

Figure 9.6 shows the **Trace Unit** panel of the EOnCE Configurator.

Figure 9.6 EOnCE Configurator Trace Unit Panel



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

□ Enable Trace Reporter Window

□ Enable Dumping To File: Choose

Table 9.6 lists and defines the settings you can make on the **Trace Unit** panel of the EOnCE Configurator.

Table 9.6 Trace Unit Panel Description

Panel Item	Description
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.
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.

Table 9.6 Trace Unit Panel Description (*continued*)

Panel Item	Description
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.

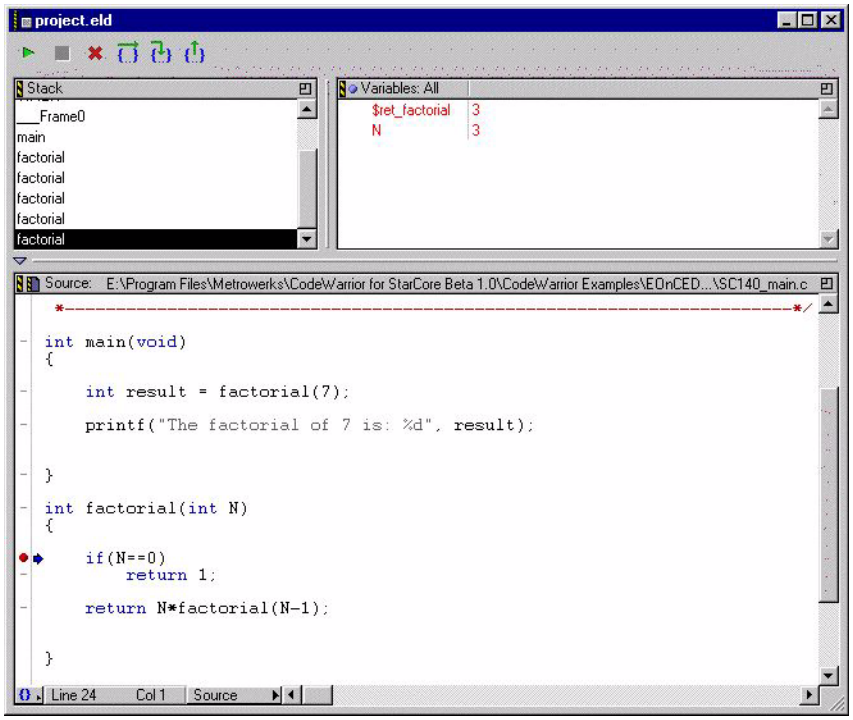
EOnCE Example: Counting Factorial Function Calls

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 9.7 shows the debugger window after counting the call to `factorial` five times using a regular software breakpoint.

Figure 9.7 Debugger Window: Counting with a Regular Software Breakpoint



Stack:

```
___Frame0
main
factorial
factorial
factorial
factorial
factorial
```

Variables: All

$ret_factorial	3
N	3

Source: E:\Program Files\Metrowerks\CodeWarrior for StarCore Beta 1.0\CodeWarrior Examples\EOnCED...\SC140_main.c

```
  *-----------------------------------------------------------------------------*/

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** panel.
6. Configure the **Event Counter** panel.
7. Configure the **Event Selector** panel.

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`

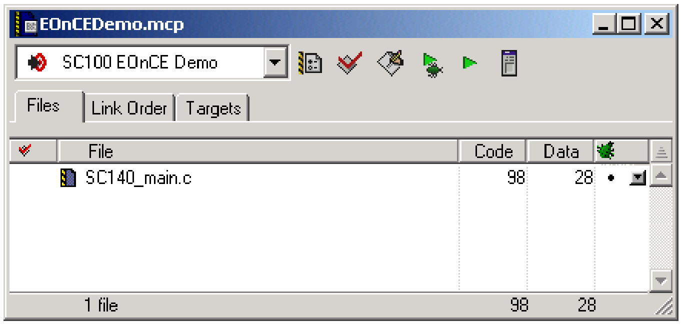
Solaris

`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 9.8.)

Figure 9.8 EOnCEDemo Project Window



File	Code	Data
SC140_main.c	98	28
1 file	98	28

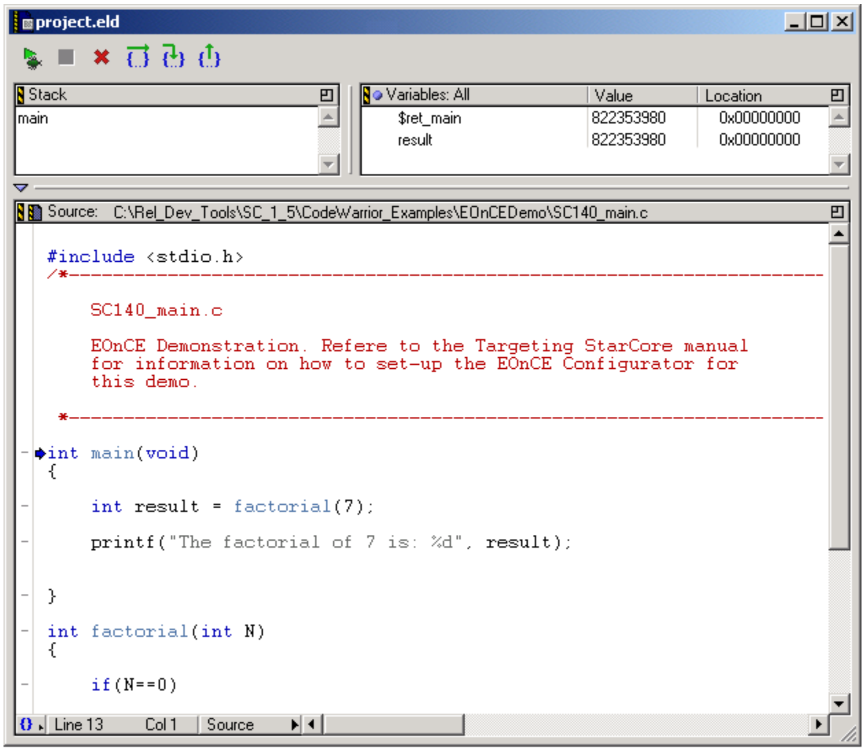
Download the EOnCEDemo Project

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 9.9.

NOTE You must download your program to the target board before configuring EOnCE debugging conditions.

Figure 9.9 Debugger Window for the EOnCEDemo Project



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)
```

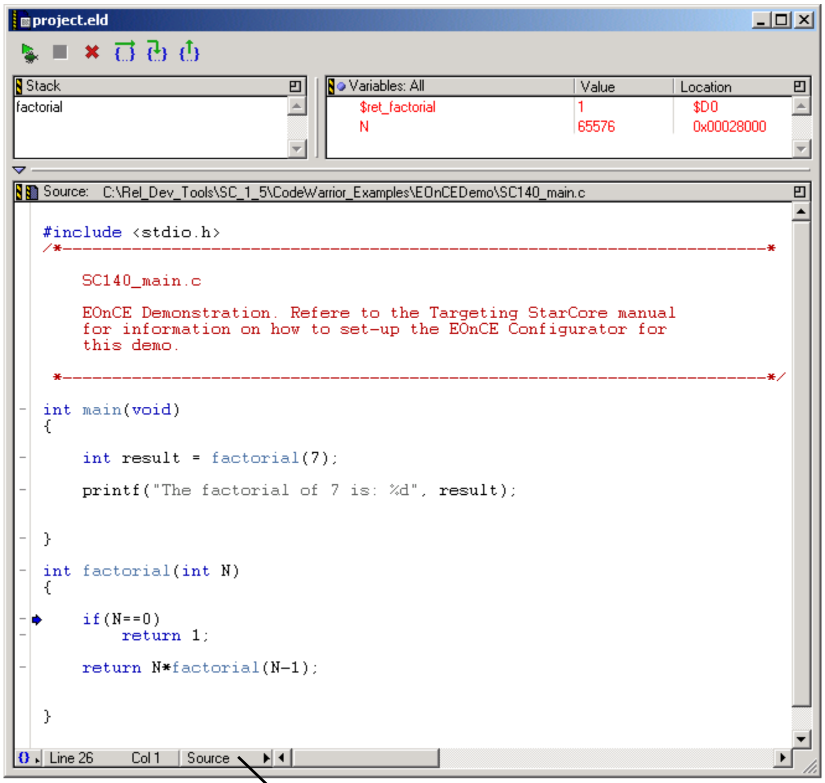
Get the Address of the Instruction

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 9.10 shows the debugger window after you move the arrow.

Figure 9.10 Current Statement Arrow: First Line of the Factorial Function



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);

}
```

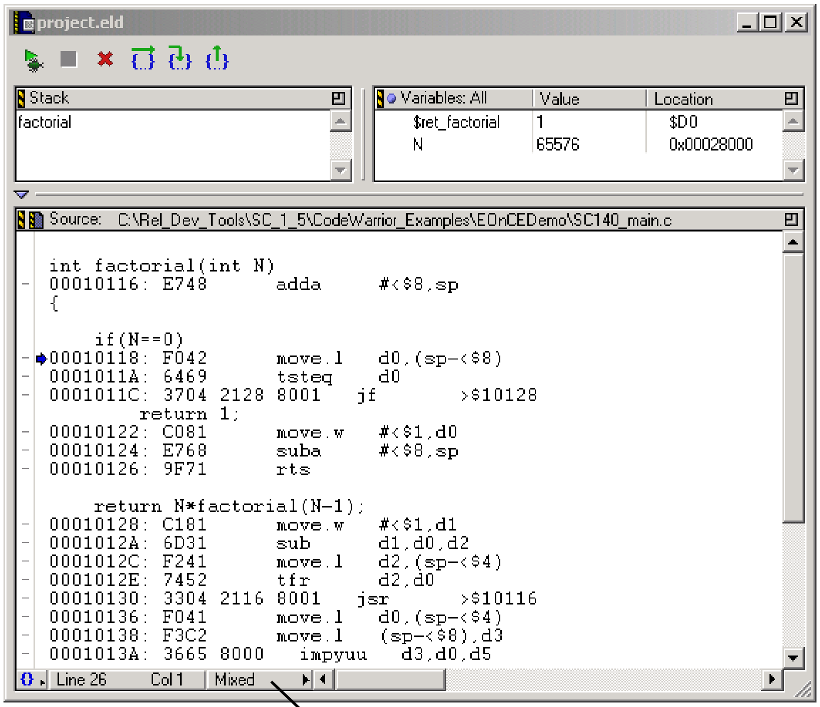
Source button

2. At the bottom of the debugger window, click the **Source** button, and 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 9.11.)

Figure 9.11 Debugger Window Displaying a Mixed Code View


project.eld
Stack
factorial
Source: C:\Rel_Dev_Tools\SC_1_5\CodeWarrior_Examples\EOnCEDemo\SC140_main.c
Line 26
Col 1
Mixed

Variables: All	Value	Location
$ret_factorial	1	$D0
N	65576	0x00028000

```
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
```

Mixed button

3. Write down the address value of the instruction immediately preceding the location to which the Current Statement arrow now points.

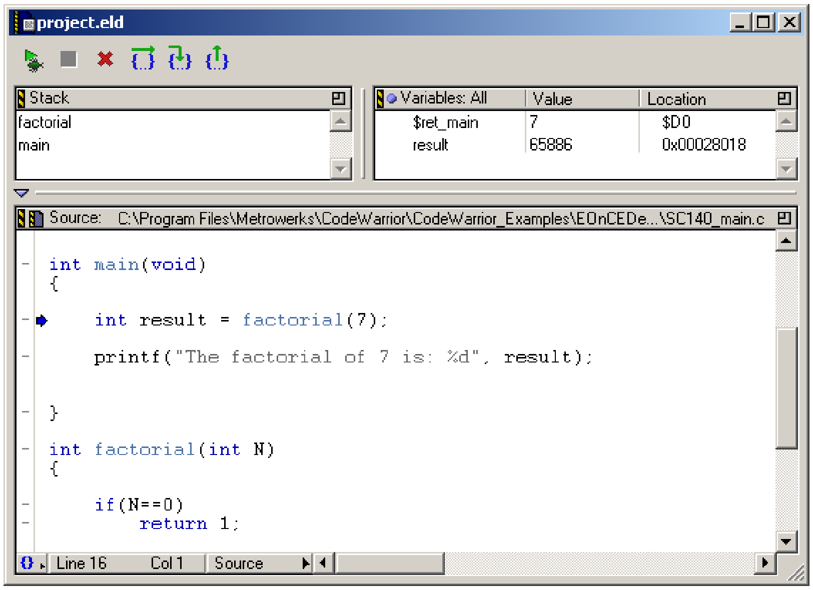
4. At the bottom of the debugger window, click the **Mixed** button, and choose Source from the menu that appears.

   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 9.12 shows the debugger window after you perform these actions.

Figure 9.12 Current Statement Arrow on the First Statement in main()



Stack
factorial
main

Variables: All	Value	Location
$ret_main	7	$D0
result	65886	0x00028018

Source: C:\Program Files\Metrowerks\CodeWarrior\CodeWarrior_Examples\EOnCEDe...\SC140_main.c

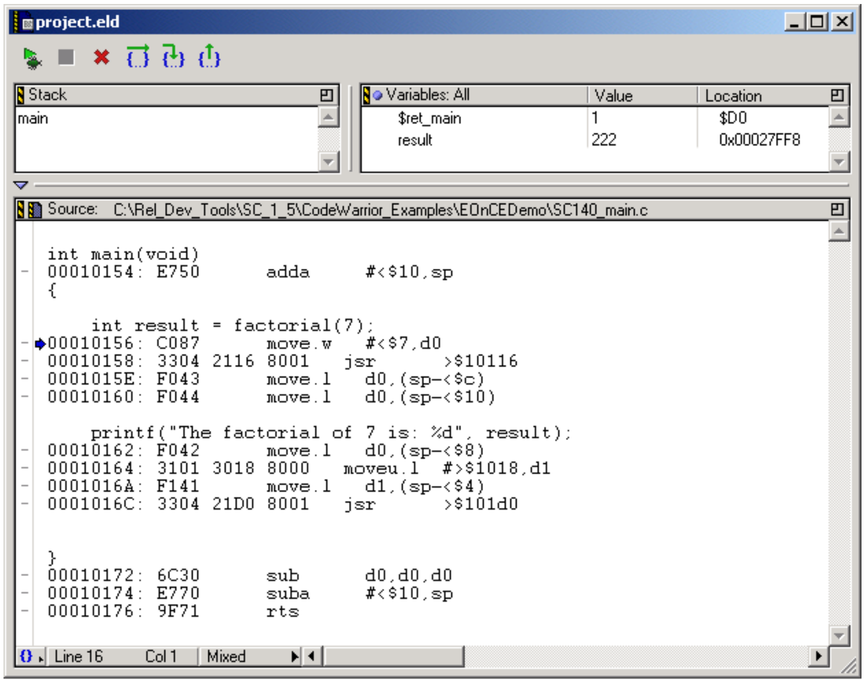
```
int main(void)
{
    int result = factorial(7);

    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, and 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 9.13 shows the debugger window after you perform these actions.

Figure 9.13 Debugger Window—Mixed Code View



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
```

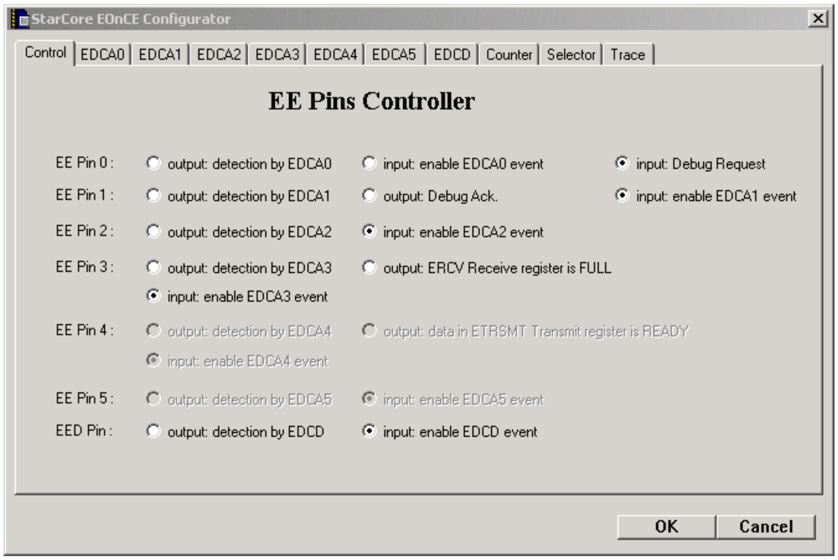
Open the EOnCE Configurator

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 9.14.)

Figure 9.14 EE Pins Controller Panel



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** panel, 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** panel are not relevant; do not change them.

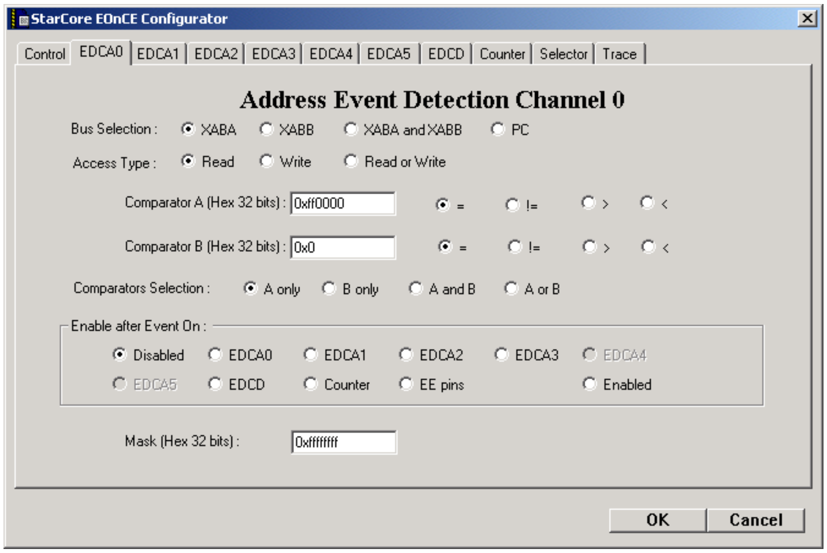
Configure the Address Event Detection Channel 0 Panel

To choose settings for the address event detection channel 0 by configuring the **Address Event Detection Channel 0** panel:

1. Click the **EDCA0** tab.

   The IDE displays the chosen panel. (See Figure 9.15).

Figure 9.15 Address Event Detection Channel 0 Panel



StarCore EOnCE Configurator

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) : 0xff0000 ● = ○ != ○ > ○ <

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.

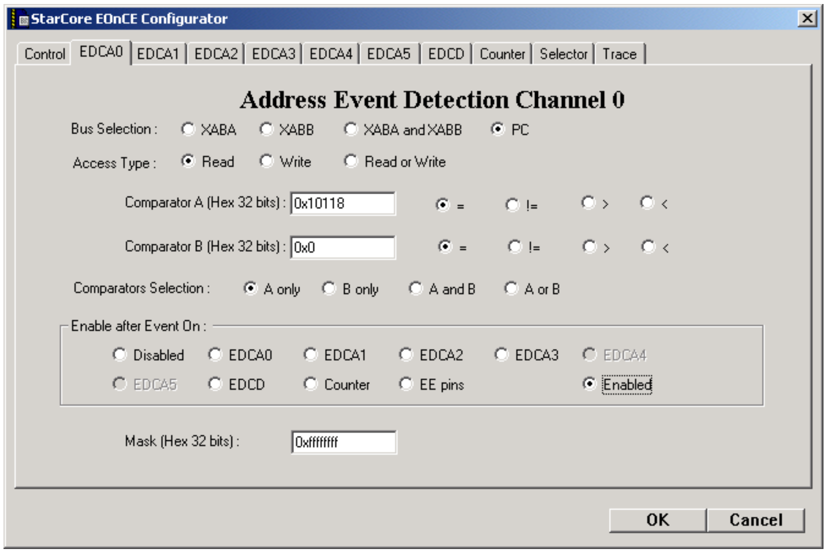
NOTE When selecting settings in the EOnCE Configurator, configure the tabbed panels in the left-to-right order of the tabs. For example, configure the **Address Event Detection Channel 0** panel before configuring the **Event Counter** panel. In addition, within a panel, configure your selected settings 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 panel now appears as shown in Figure 9.16.

Figure 9.16 Address Event Detection Channel 0 after Changing Settings



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

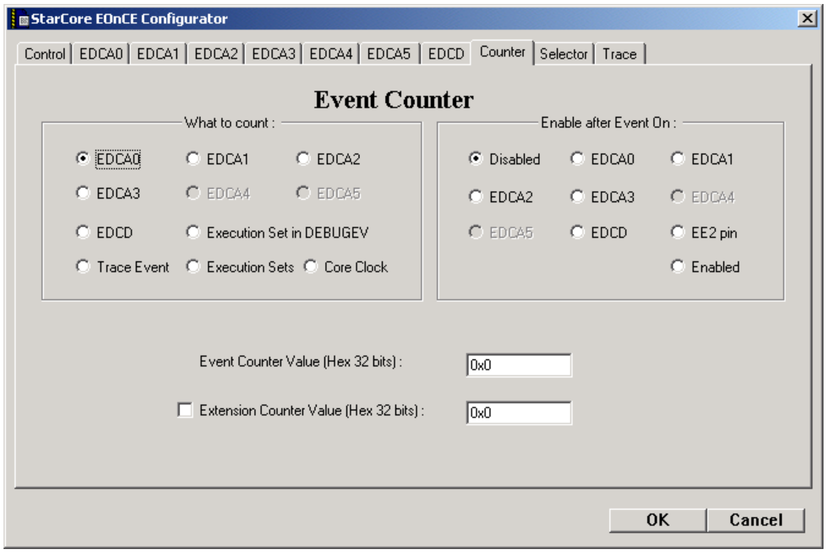
Configure the Event Counter Panel

To configure the **Event Counter** panel to count a particular event on address channel 0:

1. Click the **Counter** tab.

   The IDE displays the **Event Counter** panel. (See Figure 9.17.)

Figure 9.17 Event Counter Panel



Event Counter

What to count :
● EDCA0 ○ EDCA1 ○ EDCA2
○ EDCA3 ○ EDCA4 ○ EDCA5
○ EDCD ○ Execution Set in DEBUGEV
○ Trace Event ○ Execution Sets ○ Core Clock

Enable after Event On :
● Disabled ○ EDCA0 ○ EDCA1
○ EDCA2 ○ EDCA3 ○ EDCA4
○ EDCA5 ○ EDCD ○ EE2 pin
○ Enabled

Event Counter Value (Hex 32 bits) : 0x0

□ Extension Counter Value (Hex 32 bits) : 0x0

By default, the **Event Counter** panel specifies to count EDC0 (address channel 0, which corresponds with the **Address Event Detection Channel 0** panel that you just configured and is correct for this example).

2. For Enable after Event On, select Enabled.

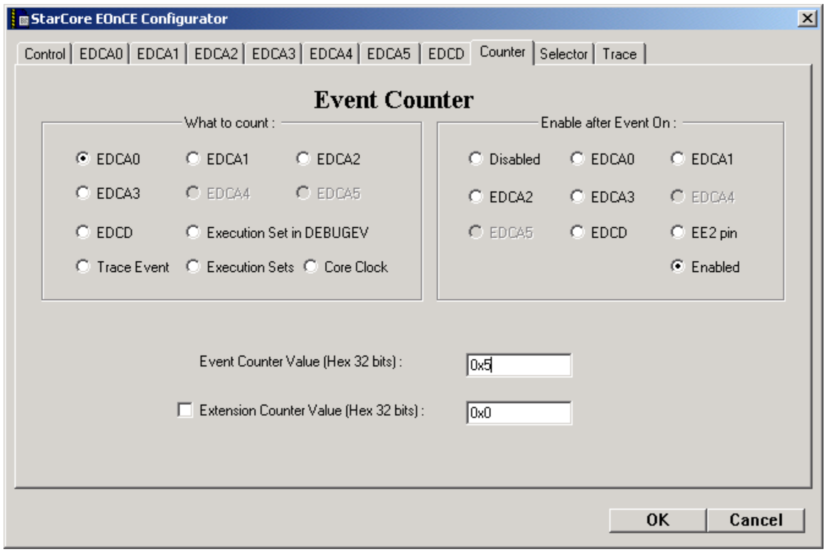
3. Type the value (in hexadecimal) for how many times you want to count in the Event Counter Value (Hex 32 bits) field.

   For this example, type:

   ```
   0x5
   ```

   Figure 9.18 shows the **Event Counter** panel after your changes.

Figure 9.18 Event Counter Panel after Changing Settings



Event Counter

What to count :

● EDCA0 ○ EDCA1 ○ EDCA2
○ EDCA3 ○ EDCA4 ○ EDCA5
○ EDCD ○ Execution Set in DEBUGEV
○ Trace Event ○ Execution Sets ○ Core Clock

Enable after Event On :

○ Disabled ○ EDCA0 ○ EDCA1
○ EDCA2 ○ EDCA3 ○ EDCA4
○ EDCA5 ○ EDCD ○ EE2 pin
● Enabled

Event Counter Value (Hex 32 bits) : 0x5

□ Extension Counter Value (Hex 32 bits) : 0x0

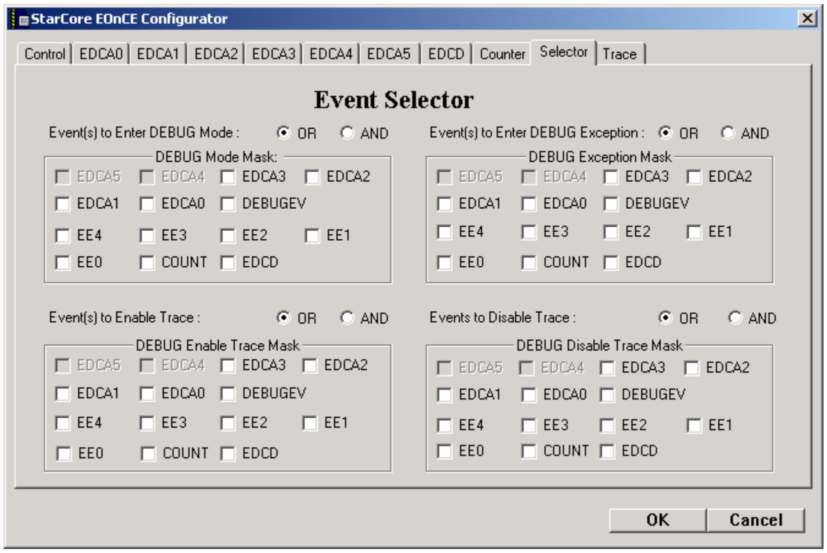
Configure the Event Selector Panel

To configure the **Event Selector** panel:

1. Click the **Selector** tab.

   The IDE displays the **Event Selector** panel. (See Figure 9.19.)

Figure 9.19 Event Selector Panel



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

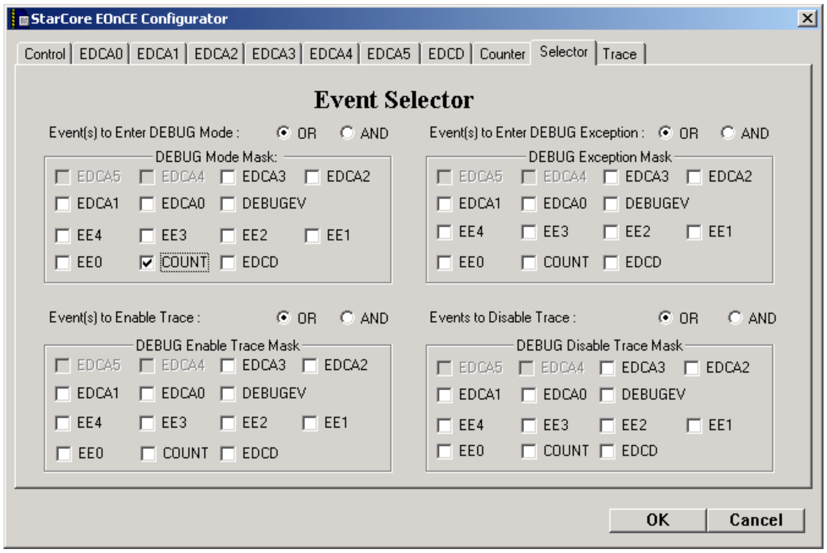
The default setting for Event(s) to Enter DEBUG Mode is OR, which is correct for this example.

2. In DEBUG Mode Mask, click the COUNT checkbox (to enable it).

   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 9.20 shows the appearance of the **Event Selector** panel after this change.

Figure 9.20 Event Selector Panel 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.

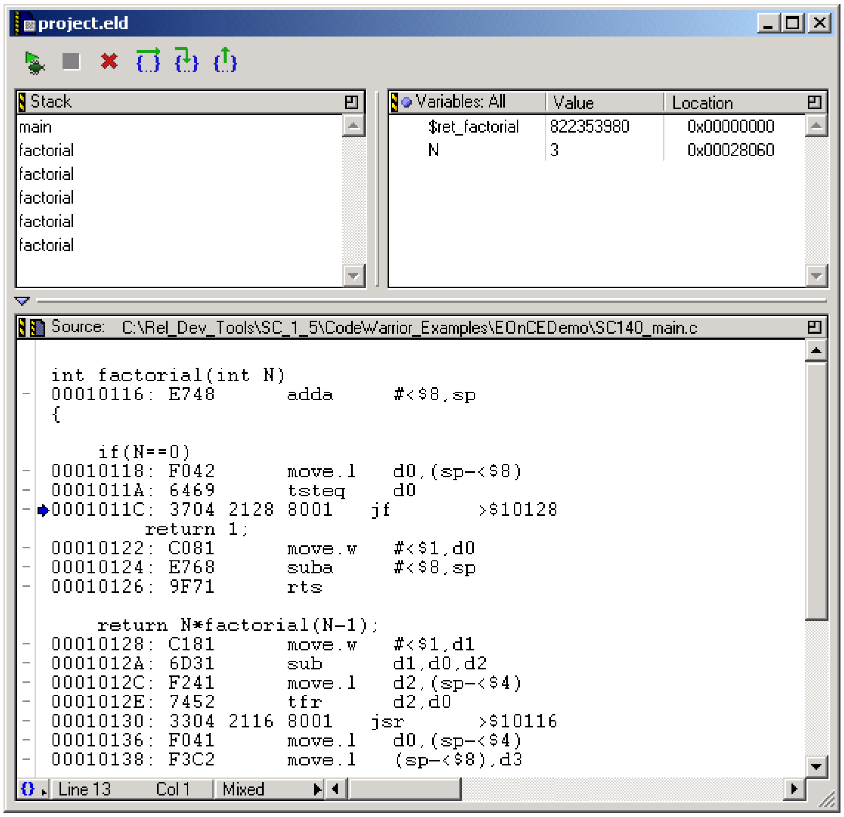
Run the EOnCE Factorial Count Debugging Example

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 9.21 shows the appearance of the debugger window after you run the debugging example.

Figure 9.21 Debugger Window after Running the Debugging Example



Stack
main
factorial
factorial
factorial
factorial
factorial

Variables: All	Value	Location
$ret_factorial	822353980	0x00000000
N	3	0x00028060

Source: C:\Rel_Dev_Tools\SC_1_5\CodeWarrior_Examples\EOnCEDemo\SC140_main.c

```
 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

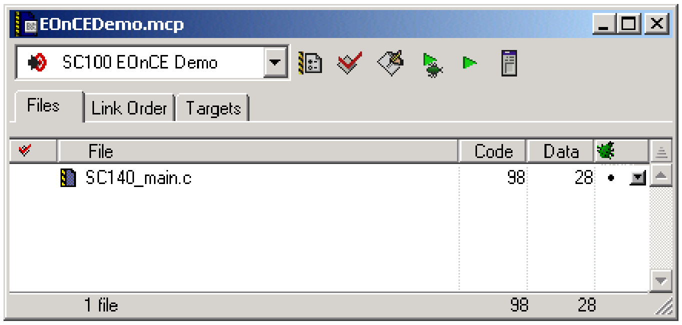
`*installDir*\Examples\StarCore\EOnCEDemo`

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 9.22.)

Figure 9.22 EOnCEDemo.mcp Project Window



File	Code	Data
SC140_main.c	98	28
1 file	98	28

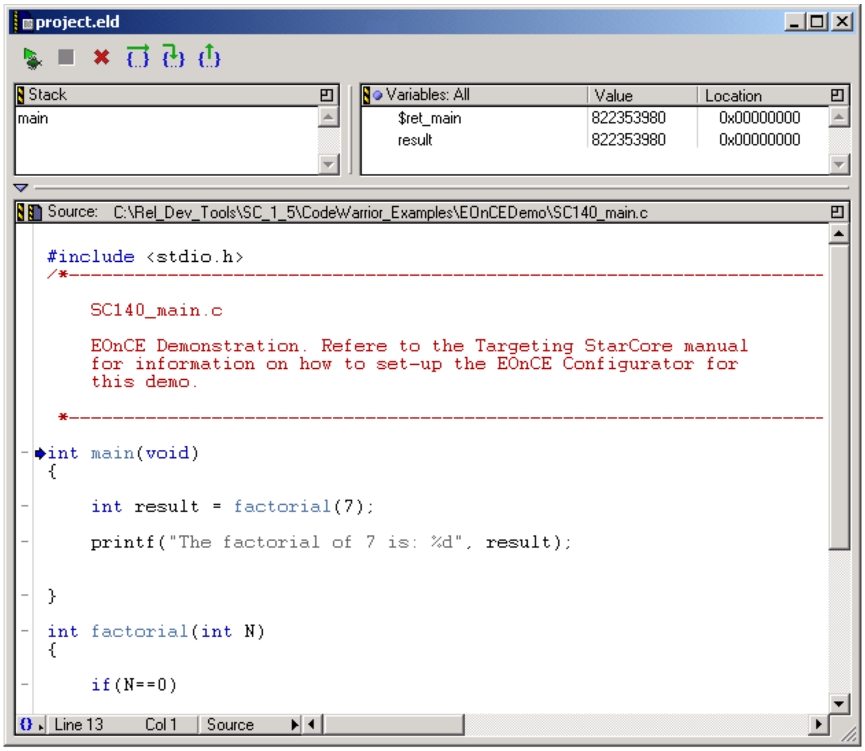
Download the EOnCEDemo Project

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 9.23.

NOTE You must download your program to the target board before configuring EOnCE debugging conditions.

Figure 9.23 Debugger Window for the EOnCEDemo Project



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 13 Col 1 Source

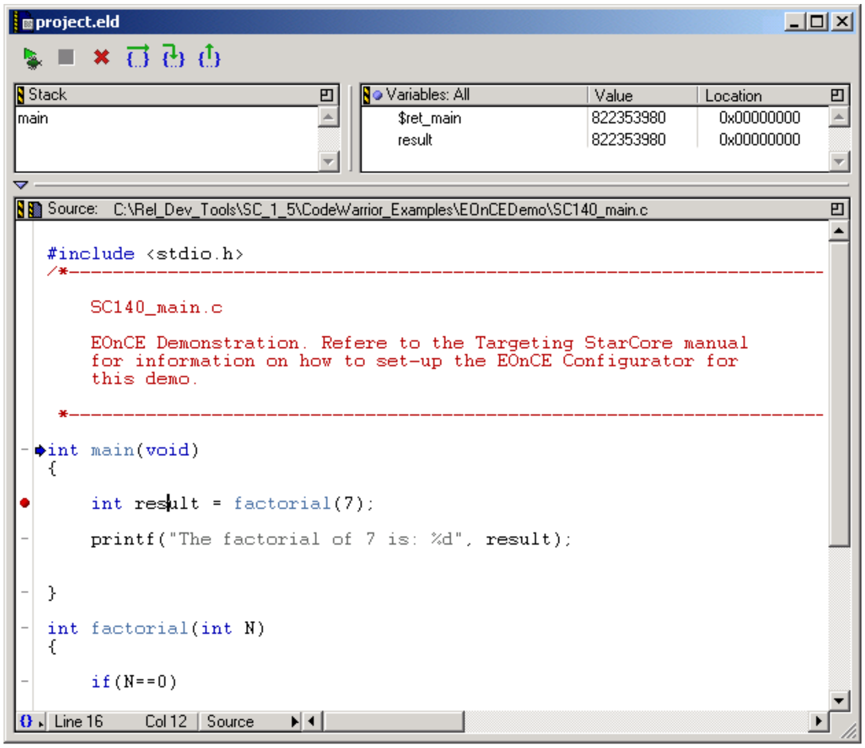
Set a Breakpoint

To set a breakpoint, click the gray dash next to the this statement in the debugger window:

```
int result = factorial (7);
```

Figure 9.24 shows the debugger window after you set the breakpoint.

Figure 9.24 Debugger Window after Setting the Breakpoint



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

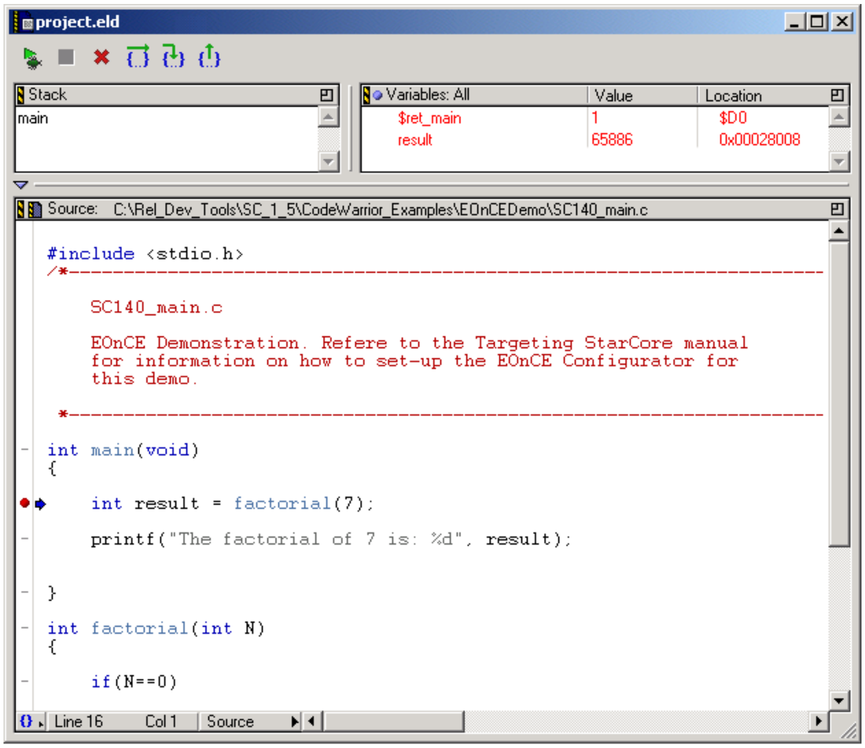
```
#include <stdio.h>
/*-----------------------------------------------------------------------

    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)
```

Run to the Breakpoint

Choose **Project > Run** to run to the breakpoint you previously set.

Figure 9.25 shows the debugger window after running to the breakpoint.

Figure 9.25 Debugger Window after Running to the Breakpoint



project.eld

Stack
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)
```

Line 16 Col 1 Source

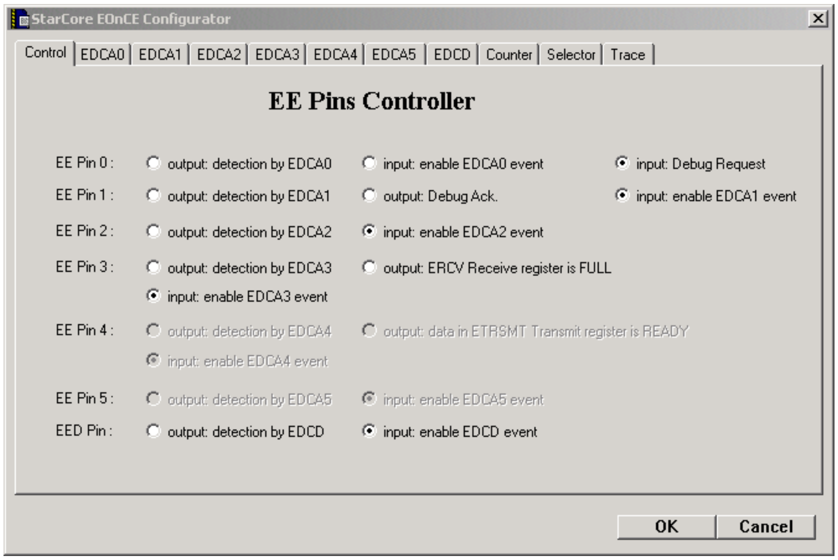
Open the EOnCE Configurator

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 9.26.)

Figure 9.26 EE Pins Controller Panel



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** panel, 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** panel shown in Figure 9.26 are correct; do not change them.

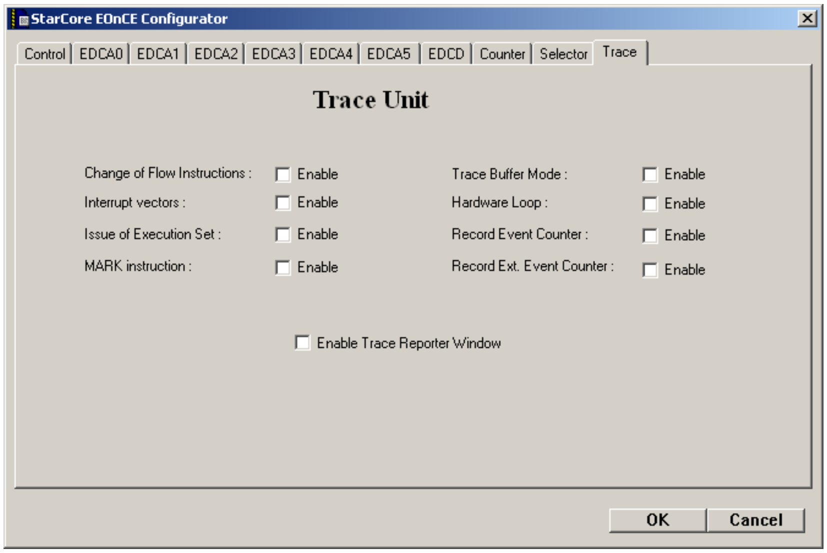
Configure a Trace

To configure the trace for this example:

1. Click the **Trace** tab.

   The IDE displays the **Trace Unit** panel. (See Figure 9.27.)

Figure 9.27 EOnCE Configurator Trace Unit Panel



StarCore EOnCE Configurator

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

□ Enable Trace Reporter Window

2. Check the Change of Flow Instructions checkbox.

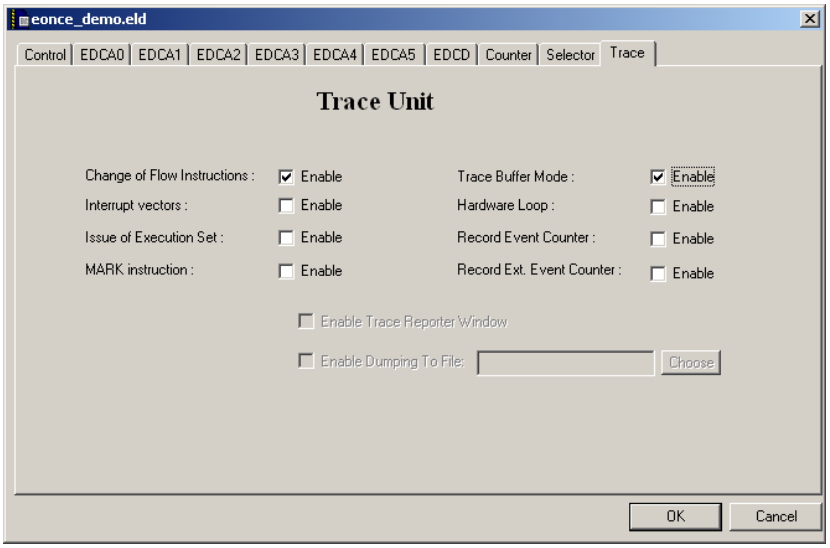
   This enables a trace on anything that changes the instruction flow (for example, a jump, branch, or return to subroutine instruction).

3. Check the Trace Buffer Mode checkbox.

4. Check the Enable Trace Reporter Window checkbox.

   Figure 9.28 shows the **Trace Unit** panel after configuration.

Figure 9.28 EOnCE Configurator Trace Unit Panel after Configuration



Trace Unit

Change of Flow Instructions : ☑ Enable

Interrupt vectors : □ Enable

Issue of Execution Set : □ Enable

MARK instruction : □ Enable

Trace Buffer Mode : ☑ Enable

Hardware Loop : □ Enable

Record Event Counter : □ Enable

Record Ext. Event Counter : □ Enable

□ Enable Trace Reporter Window

□ Enable Dumping To File:

Save the EOnCE Configurator Settings

To save your changes, click **OK** in the **EOnCE Configurator** window.

Run the EOnCE Trace Buffer Debugging Example

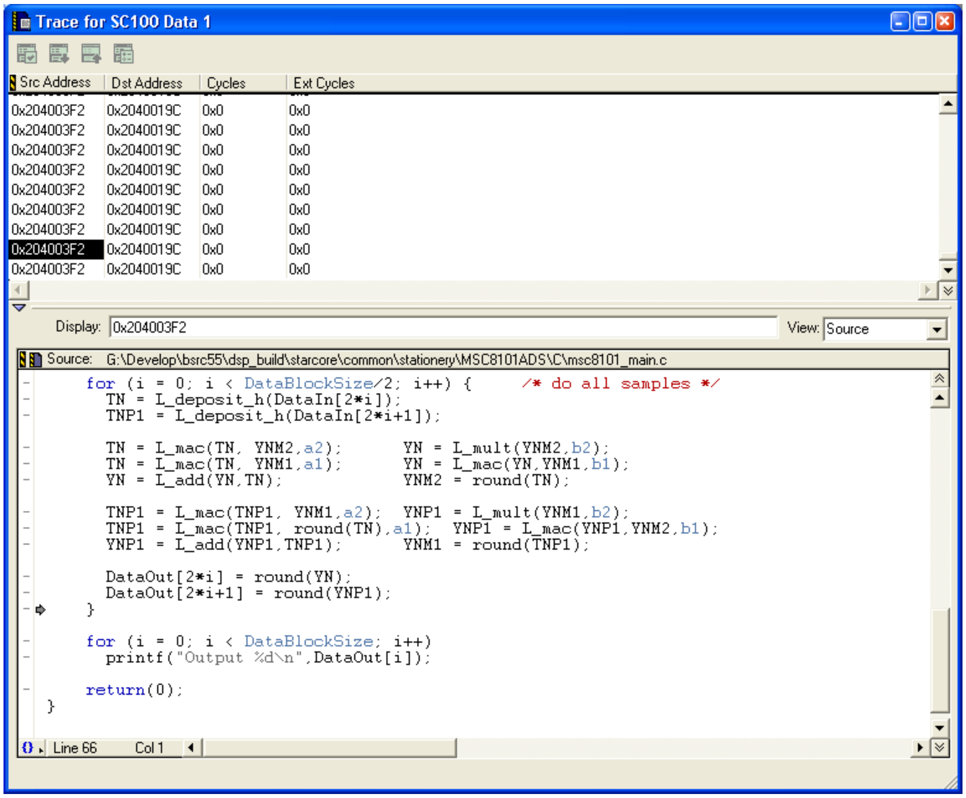
1. To run the EOnCE trace buffer debugging example, choose **Debug > Step Over**.

   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 9.29.)

Figure 9.29 EOnCE Trace View Window



Src Address	Dst Address	Cycles	Ext Cycles
0x204003F2	0x2040019C	0x0	0x0
0x204003F2	0x2040019C	0x0	0x0
0x204003F2	0x2040019C	0x0	0x0
0x204003F2	0x2040019C	0x0	0x0
0x204003F2	0x2040019C	0x0	0x0
0x204003F2	0x2040019C	0x0	0x0
0x204003F2	0x2040019C	0x0	0x0
0x204003F2	0x2040019C	0x0	0x0
0x204003F2	0x2040019C	0x0	0x0

Display: 0x204003F2 View: Source

Source: G:\Develop\bsrc55\dsp_build\starcore\common\stationery\MSC8101ADS\C\msc8101_main.c

```
    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);
}
```

Line 66 Col 1

10

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 code that are not being executed or that are taking 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.

The sections of this chapter are:

- Profiler Types
- Profiler Points
- Profiler Examples
- Launching the Profiler
- Opening the Profiler Sessions Window
- Removing a Profiler Session
- Removing All Profiler Sessions
- View a List of Functions
- View an Instruction-Level Report
- View Function Details
- View a Function Call Tree
- View Source Files Information
- View Profile Information Line-by-Line
- Save a Profile
- Load a Profile
- Generate a Tab-Delimited Profile Report
- Generate an HTML Profile Report
- Generate an XML Profiling Report
- Set Up to Profile Assembly Language Programs

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 select On Host from the Profiler Type listbox of the Profiler target settings panel.

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 with program execution quite a bit.

On Chip

As its name implies, the code the 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. The on-chip ISR technique interferes with program execution less than does the on host approach.

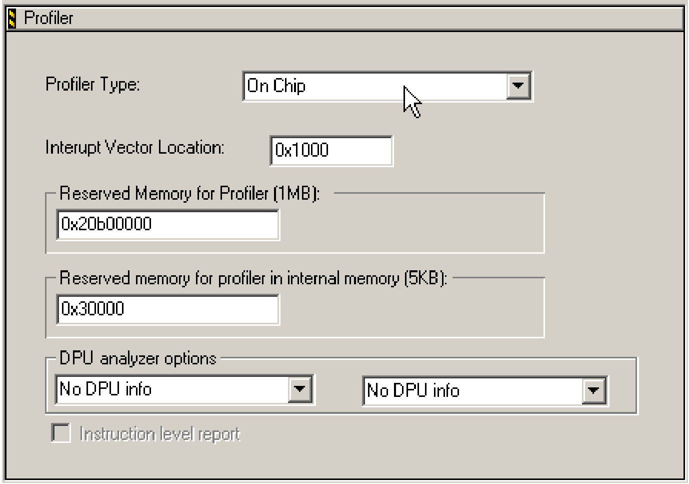
Unlike on host profiling, set up for on chip profiling is complex. To configure a build target for on chip profiling, follow these steps:

1. Open the project for which you want to use on chip profiling.
2. Press **Alt-F7**

   The **Target Settings** window appears.
3. In the pane on the left of this window, click Profiler.

   The Profiler panel appears on the right of **Target Settings** window. (See Figure 10.1.)

Figure 10.1 Profiler Target Settings Panel



Profiler

Profiler Type: On Chip

Interupt Vector Location: 0x1000

Reserved Memory for Profiler (1MB): 0x20b00000

Reserved memory for profiler in internal memory (5KB): 0x30000

DPU analyzer options: No DPU info | No DPU info

□ Instruction level report

4. From the Profiler Type listbox, select On Chip.
5. In the Interrupt Vector Location text box, type the address (in hexadecimal) of the vector to the interrupt service routine used by the on chip profiler.
6. In the Reserve Memory for Profiler (1MB) text box, type the base address of the 1MB 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 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 Reserve memory for profiler in internal memory (5KB) text box, type the base address of the 5KB 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 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).

    To do this, follow these steps:

    a. In an assembly language file, add the code shown in Figure 10.1, *or* in a C language file, add the code shown in Figure 10.2.

Listing 10.1 Assembly Language Declaration of a Buffer for the On Chip Profiler's ISR

```
org p:$C00
    dup $40
    nop
    endm
```

Figure 10.2 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 0, 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:

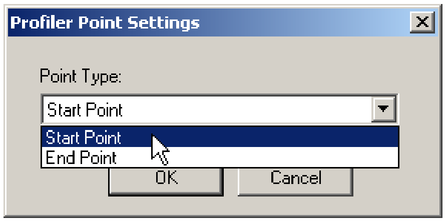
   a. Open the source file in which you want to set a pair of profiler points in an editor window.

   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 10.3.)

Figure 10.3 Profiler Point Settings Dialog Box

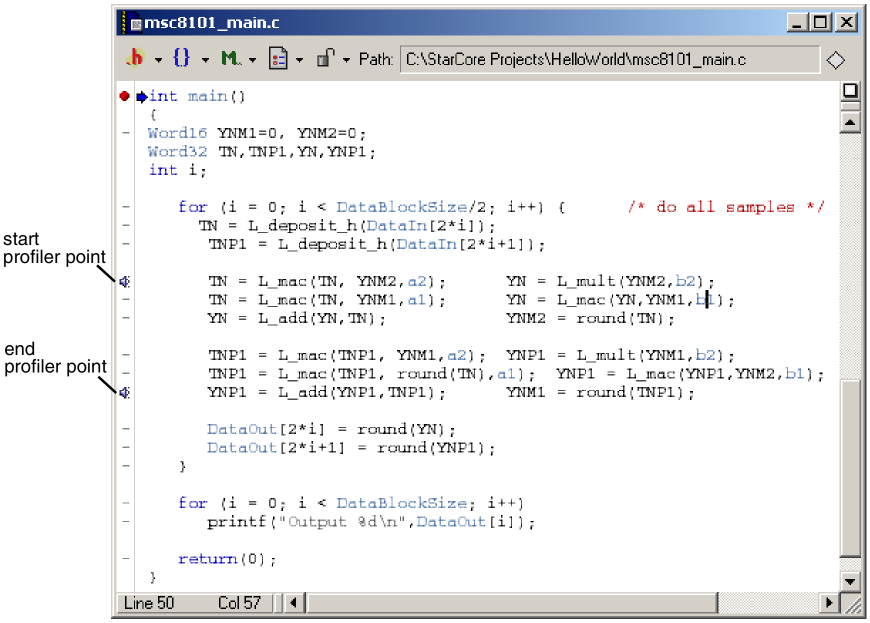


d. From the Point Type listbox, select Start Point.

e. Click **OK**

The debugger assigns a start profiler point to the selected source code statement. (See Figure 10.4.)

Figure 10.4 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 10.4.)

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.

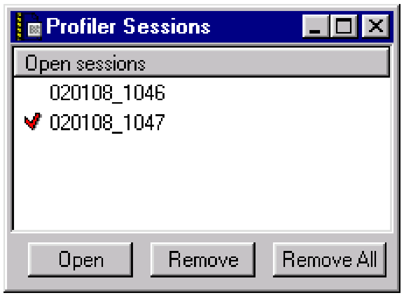
NOTE If your program makes `printf` calls to `stdout`, the output appears in the IDE’s I/O window. This can be useful for marking the progress of your profiling execution.

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 10.5.)

Figure 10.5 Profiler Sessions Window



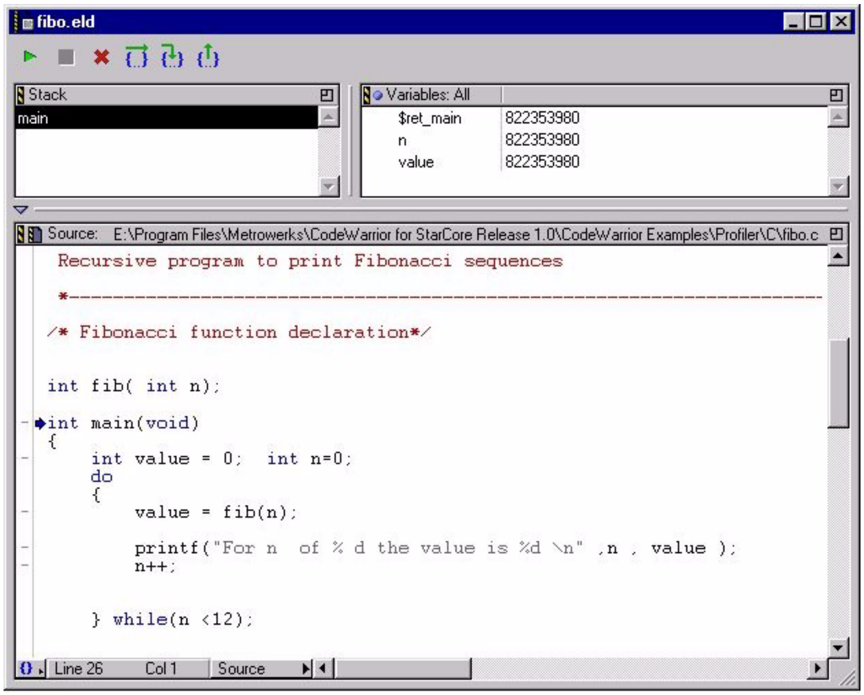
020108_1046
✓ 020108_1047

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 10.6.)

NOTE You can use any debugging commands once the debugger window appears.

Figure 10.6 Debugger Window—Upon Starting the Profiler



Stack
main

Variables: All	
$ret_main	822353980
n	822353980
value	822353980

Source: E:\Program Files\Metrowerks\CodeWarrior for StarCore Release 1.0\CodeWarrior Examples\Profiler\C\fibo.c

```
 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.
If you are debugging a multi-core project, downloading the master project may cause the other projects in the multi-core project to be downloaded as well. In this case, if you are using 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**.

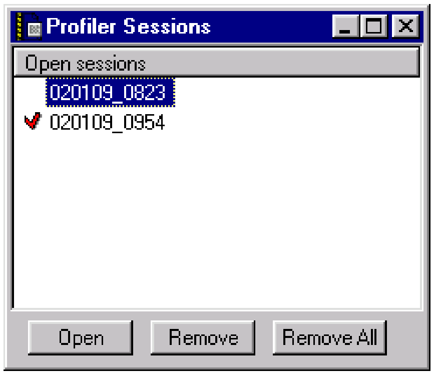
Removing a Profiler Session

To remove a profile from the **Profiler Sessions** window, perform these steps:

1. Click the name of any profiler session to highlight it.

   Figure 10.5 shows a **Profiler Sessions** window after highlighting a session.

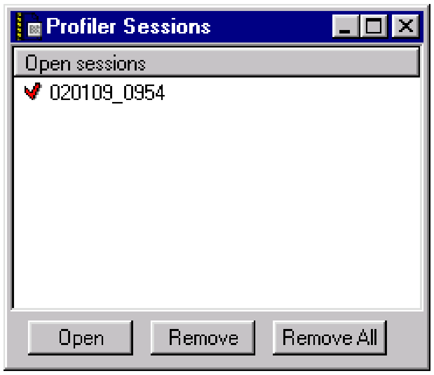
Figure 10.7 Profiler Sessions Window with Session Name Highlighted



2. Click **Remove**

   The IDE removes the session from the **Profiler Sessions** window and closes all other profiler windows related to that session. Figure 10.8 shows the **Profiler Sessions** window after you delete the chosen session.

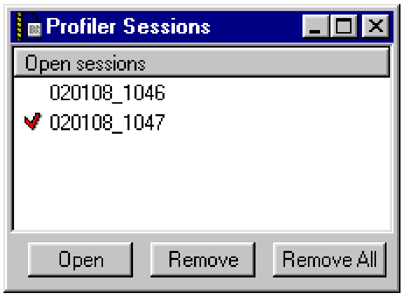
Figure 10.8 Profiler Sessions Window After Removing a Session



Removing All Profiler Sessions

To remove all profiler sessions from a **Profiler Sessions** window, click **Remove All**. (See Figure 10.9.)

Figure 10.9 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:

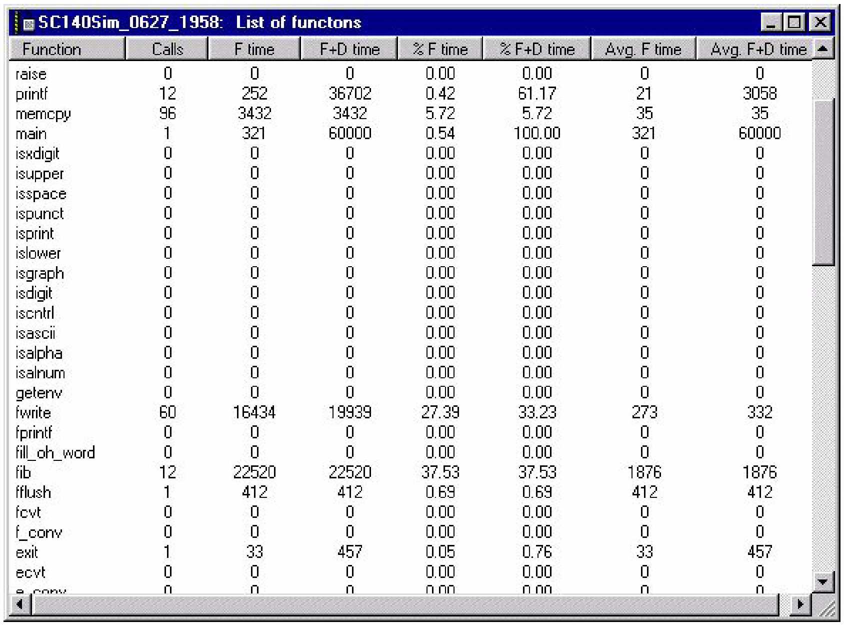
- Choose **Profiler > Functions**

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 10.18.)

Figure 10.10 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**

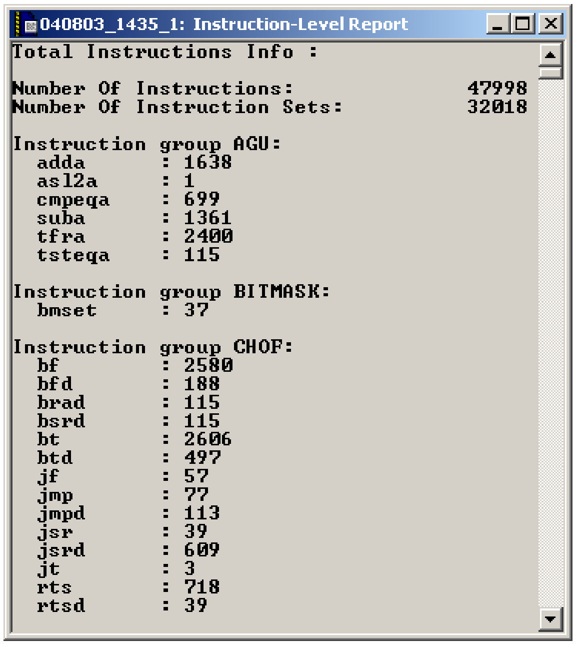
   The debugger downloads your build target to the target device and then displays the debugger window and the **Profiler Sessions** window.
8. In the **Profiler Sessions** window, select the profiler session to use and click **Open**
9. Use the debugger window to execute your program under control of the 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 10.11.)

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 10.11 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:

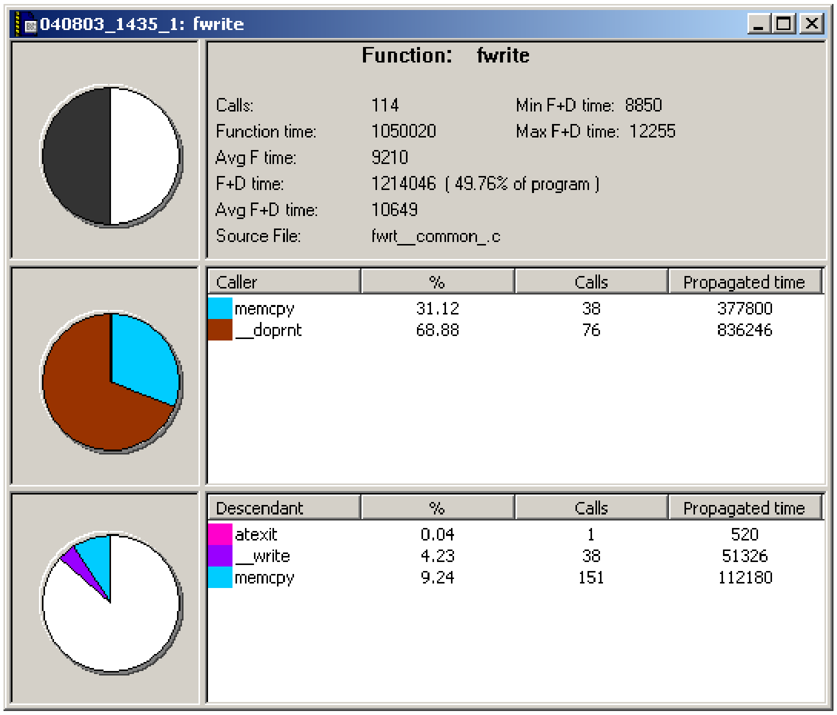
- The number of instructions executed
- The number of instruction sets executed
- A list of executed instructions grouped by category

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 10.12.)

Figure 10.12 The Function Details Window



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.

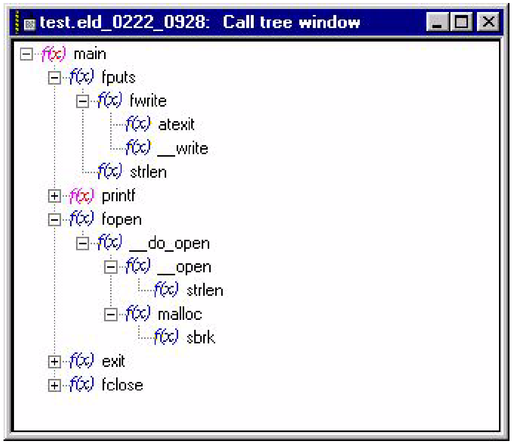
View a Function Call Tree

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 10.13.)

Figure 10.13 The Function Call Tree Window



test.eld_0222_0928: Call tree window

```
⊟ f(x) main
   ⊟ f(x) fputs
      ⊟ f(x) fwrite
            f(x) atexit
            f(x) __write
         f(x) strlen
   ⊞ f(x) printf
   ⊟ f(x) fopen
      ⊟ f(x) __do_open
         ⊟ f(x) __open
               f(x) strlen
         ⊟ f(x) malloc
               f(x) sbrk
   ⊞ f(x) exit
   ⊞ f(x) fclose
```

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.

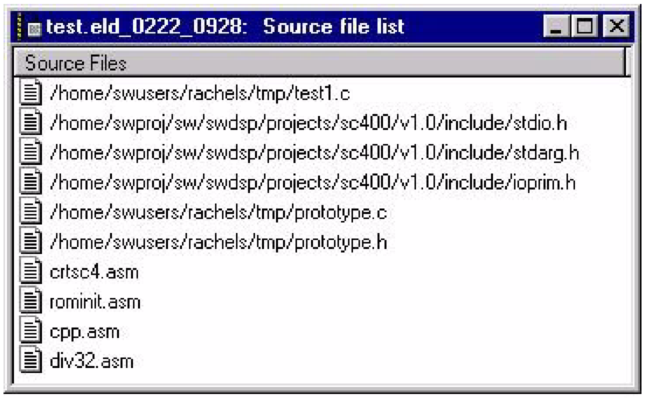
View Source Files Information

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 10.14.)

Figure 10.14 The Source Files Window



/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.

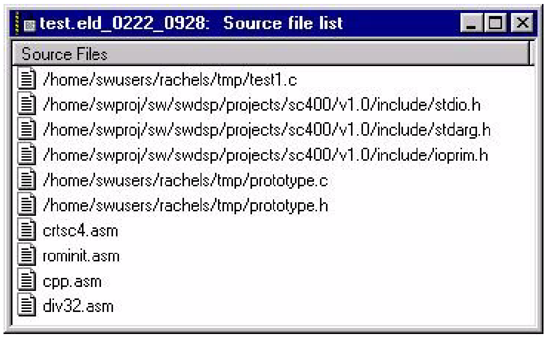
NOTE You can view profile information line-by-line by double-clicking on a source file in the **Source Files** window.

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 10.15 shows the **Source Files** window.

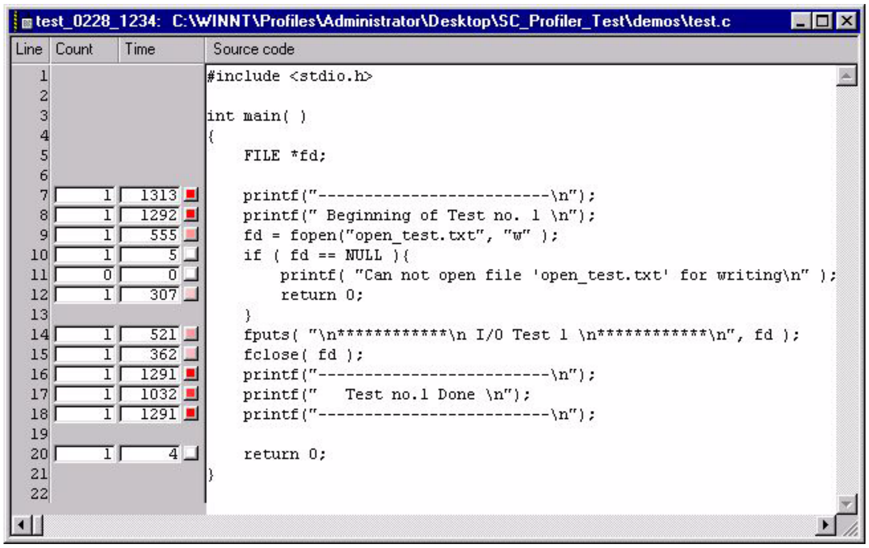
Figure 10.15 The Source Files Window



/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 10.16.)

Figure 10.16 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.

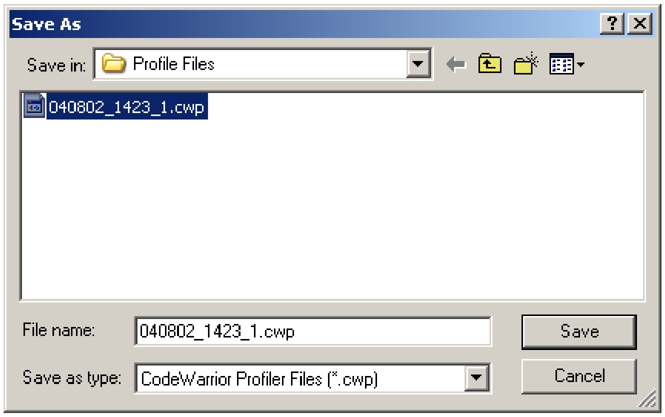
Save a Profile

To save a profile, follow these steps:

1. Choose **Profiler > Save**

   The **Save As** dialog box appears. (See Figure 10.17.)

Figure 10.17 Save As Dialog Box



Save in: Profile Files

040802_1423_1.cwp

File name: 040802_1423_1.cwp

Save as type: CodeWarrior Profiler Files (*.cwp)

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.

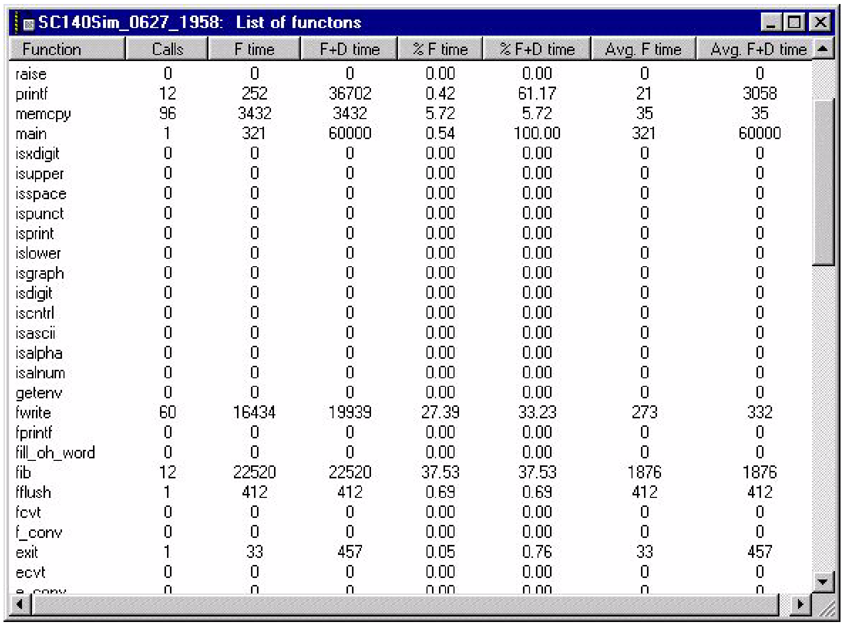
2. If needed, use the dialog box to navigate to the directory that contains the profile.
3. Select the profile and click **Open**

   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 10.18.)

Figure 10.18 The List of Functions Window



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

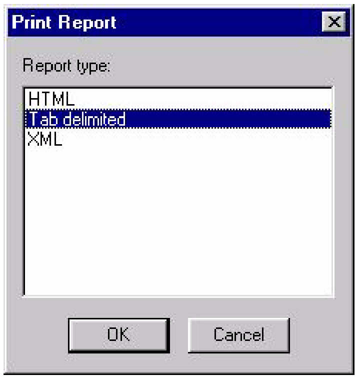
To generate a tab-delimited profile report, follow these steps:

1. Choose **Profiler > Export**

   The **Print Report** window appears.

2. Select Tab delimited in the **Print Report** window. (See Figure 10.19.)

Figure 10.19 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.

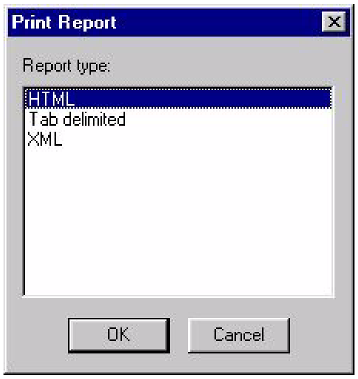
Generate an HTML Profile Report

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 10.20.)

Figure 10.20 Print Report Window with HTML Selected

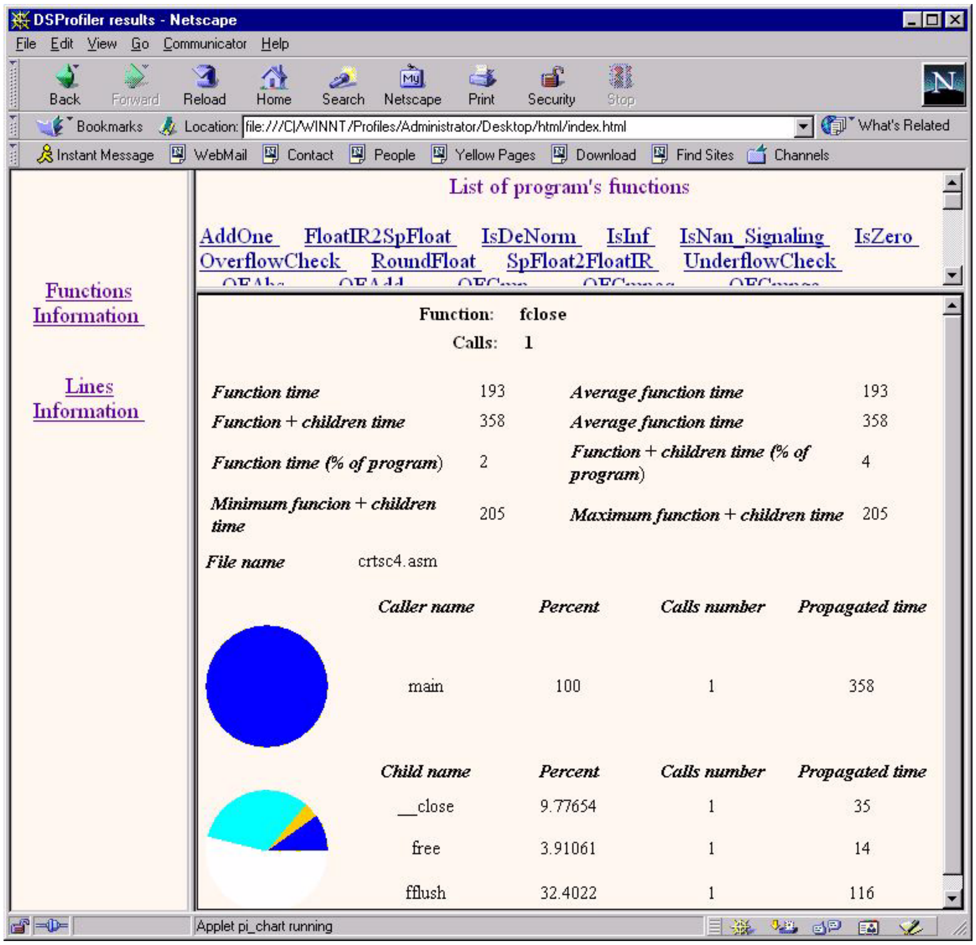


3. Click **OK**
4. Choose a location to save the file.

   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 10.21 shows an example HTML report.

Figure 10.21 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
Applet pi_chart running

List of program's functions

AddOne FloatIR2SpFloat IsDeNorm IsInf IsNan_Signaling IsZero
OverflowCheck RoundFloat SpFloat2FloatIR UnderflowCheck

Functions Information

Lines Information

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

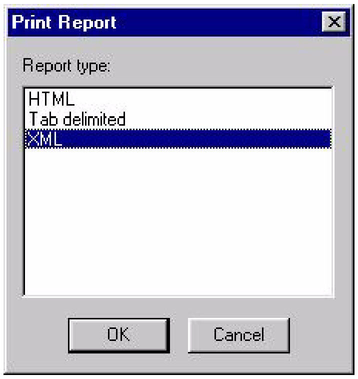
To generate a profiling report in XML:

1. Choose **Profiler > Export**

   The **Print Report** window appears.

2. Select XML in the **Print Report** window. (See Figure 10.22.)

Figure 10.22 Print Report Window with XML Selected



3. Click **OK**

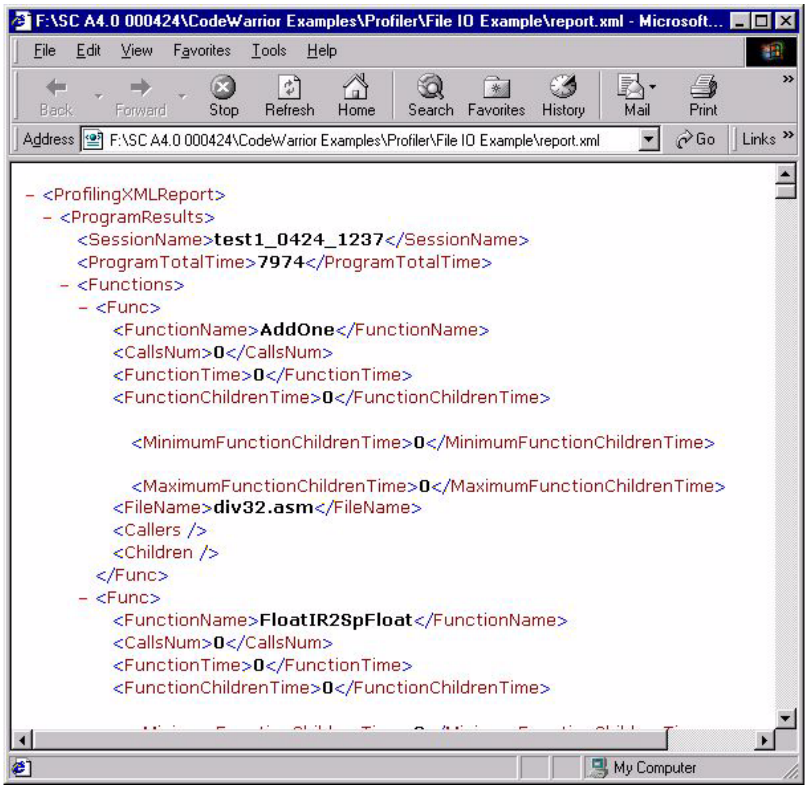
4. Choose a location to save the file.

   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 10.23 shows an example XML report.

Figure 10.23 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.

11

Debugging Optimized Code

This chapter explains how to use the CodeWarrior™ optimized code debugger.

The optimized code debugger can map executable code to its corresponding source code, even if the executable code has been heavily optimized. As a result, you can debug the version of your software intended for release (the optimized version) instead of a test version generated with optimizations disabled.

This sections of this chapter are:

- Code Mapping View Window
- Run Control for Optimized Code

Code Mapping View Window

The code mapping view (CMV) window is a debugging tool that displays a side-by-side view of the disassembly of machine instructions and the original source statements. This window is available only if you are debugging. To display it, select **View > Code Mapping View** during a debug session.

NOTE Although the CMV and debugger windows are synchronized, the debugger window lacks certain CMV run-control and breakpoint options. Examples are optimized code step evaluators, multiple program counter arrows, and various optimized code breakpoints. Furthermore, you cannot use the debugger window to debug optimized code if the view is mixed source and code. Accordingly, you should use the CMV window instead of the debugger window to debug optimized code.

This section covers these topics:

- Viewing the Code Mapping Window
- Code Mapping Window—User Interface
- Analyzing Optimized Code

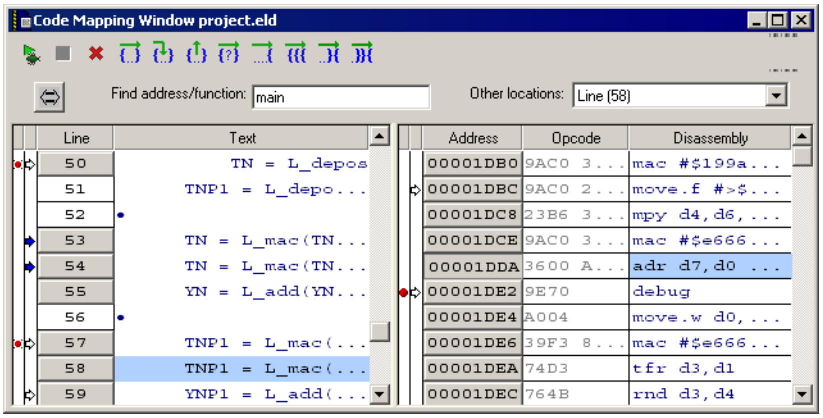
Viewing the Code Mapping Window

To begin using the **Code Mapping** window, follow these steps:

1. Choose **Project > Debug** to start a debugging session.
2. Choose **View > Code Mapping View**.

   The **Code Mapping** window appears. (See Figure 11.1.)

Figure 11.1 Code Mapping Window



Code Mapping Window project.eld

Find address/function: main

Other locations: Line (58)

Line	Text
50	TN = L_depos
51	TNP1 = L_depo...
52	•
53	TN = L_mac(TN...
54	TN = L_mac(TN...
55	YN = L_add(YN...
56	•
57	TNP1 = L_mac(...
58	TNP1 = L_mac(...
59	YNP1 = L_add(...

Address	Opcode	Disassembly
00001DB0	9AC0 3...	mac #$199a...
00001DBC	9AC0 2...	move.f #>$...
00001DC8	23B6 3...	mpy d4,d6,...
00001DCE	9AC0 3...	mac #$e666...
00001DDA	3600 A...	adr d7,d0 ...
00001DE2	9E70	debug
00001DE4	A004	move.w d0,...
00001DE6	39F3 8...	mac #$e666...
00001DEA	74D3	tfr d3,d1
00001DEC	764B	rnd d3,d4

Code Mapping Window—User Interface

The sections below discuss the user interface elements of the **Code Mapping** window**.**

Expanded Step Controls

Run Control options include the standard debugger run control commands, plus expanded step controls.

Table 11.1 Expanded Step Controls

{?}	Step Naive
....{	Step Next

Table 11.1 Expanded Step Controls (*continued*)

→ {{{	Step Forward
→ ..}{	Step After End of Statement
→ .}}{	Step After All Previous

Address Bar

The address bar contains these interface elements:

- Swap Panes

  The Swap Panes button ⇔ toggles between displaying source code in the right pane and displaying source code in the left pane.

- Find address/function

  Type a function name or address location to locate the corresponding side-by-side view. If you enter an address, it must be in hexadecimal (using the C language notation).

  For example, `0xFFFF`

- Other locations

  If an address is currently selected, this listbox displays the complete list of source code line numbers that correspond to the current address, if any.

  If a source code line is currently selected, this listbox displays the complete list of addresses that correspond to the current source code line, if any.

Pane Controls

The source and instruction view panes (Figure 11.2) contain these interface elements:

- Breakpoint

  Displays the breakpoints associated with a line of source or assembly.

- Program Counter

  Displays the program counters associated with a line of source or assembly.

- Line

  The line number of the original source.
- Text

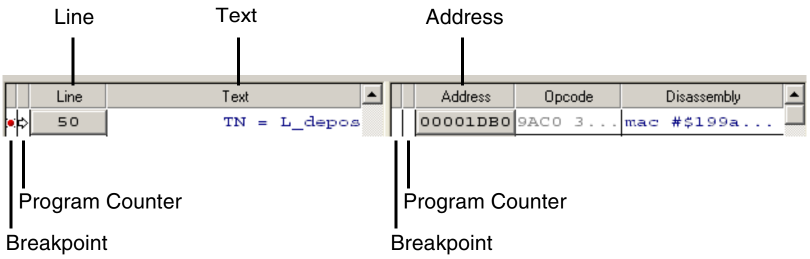
  The source code from the original project.
- Address

  The address (in hexadecimal) of the generated opcode.
- Opcode

  The opcode generated from the source file. Several opcodes often correspond to each line of source code.
- Disassembly

  The disassembled instruction of the corresponding opcode.

Figure 11.2 View Pane Elements



Line	Text
50	TN = L_depos

Address	Opcode	Disassembly
00001DB0	9AC0 3...	mac #$199a...

Analyzing Optimized Code

The Code Mapping View lets you analyze optimized code in several ways:

View Corresponding Statements

Click a statement in the source pane to highlight the corresponding disassembly in green.

Click an instruction address in the assembly pane to highlight the corresponding source instructions in blue.

The Other Locations listbox lets you browse the complete list of corresponding statement lines or disassembly addresses.

Evaluate Run Control

Right-click a statement line to open the Evaluate Run Control drop-down menu. The options in the run control menu let you preview the results of step commands. Unlike the regular program counter ➡, the preview program counters appear as a hollow arrow ⇨.

Run Control for Optimized Code

Optimized code does not follow the same flow as your source code. To help us navigate optimized code, we use special breakpoints and step functions.

- Breakpoints
- Step Functions

Breakpoints

There are four different breakpoint that help you debug optimized code in the code mapping view.

- Break Naive
- Break Begin of Statement
- Break End of Statement
- Break After All Previous

There are also two new representations for breakpoints.

- Shadow Breakpoints
- Half-Shaded Breakpoints

Break Naive

A break naive breakpoint sets breaks on all the assembly instructions or source statements that correspond to the current line.

Break Begin of Statement

A break-begin-of-statement breakpoint sets a break on the first instruction that the compiler generated for the current statement.

Break End of Statement

A break-end-of-statement breakpoint sets a break on the first instruction that is the beginning of a source statement.

Break After All Previous

A break-after-all-previous breakpoint sets a break at a location that is free from any side effects of statements that are still executing. The OCD engine places the break at the nearest location where the current statement and all its predecessors will have completed execution.

Shadow Breakpoints

When you set a breakpoint, the debugger sometimes creates shadow breakpoints elsewhere in the source. The debugger creates shadow breakpoints in two situations:

- If you set a break on an instruction, the shadow breakpoints appear on the corresponding source statements.
- If you set a breakpoint on a statement that generates instructions that also correspond to other source statements, the shadow breakpoints appear on the corresponding source statements.

The debugger represents shadow breakpoints with the ● icon, a red dot with small black dots in the corners.

You can remove a shadow breakpoint by clicking it, but doing so also removes the original breakpoint and any other related shadow breakpoints. The debugger prompts you for confirmation before removing a shadow breakpoint.

Half-Shaded Breakpoints

The code mapping view lets you set breakpoints on lines that do not have an equivalent in the source pane of the debugger window. Such breakpoints are represented by the ◐ icon, a half-shaded red dot. As there are no source pane equivalents, these breakpoints will not appear in the debugger window. Half-shaded breakpoints can only be cleared from the assembly pane of the CMV window.

Step Functions

All step commands can have the potential to reach the end of the current function. If this function is `main()`, the step command may end the program. Using the step evaluation functions of the code mapping view can help you identify such instances.

- Step Naive
- Step Next
- Step Forward
- Step After End of Statement
- Step After All Previous

Step Naive

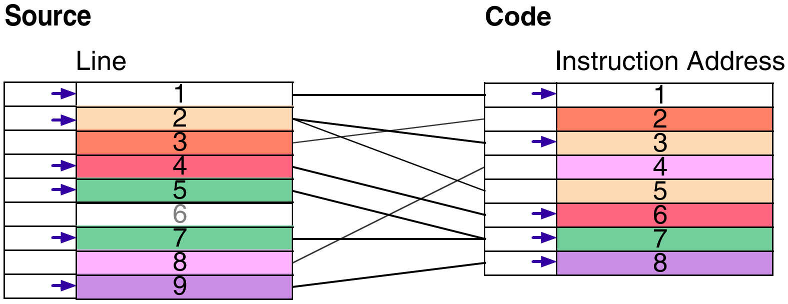
The Step Naive command steps to the first instruction of the next statement in the source code.

NOTE Figure 11.3, Figure 11.4, and Figure 11.5 show a hypothetical relationship between source code and object code:
— Execution of source line 1 executes the instruction at address 1,
— Execution of source line 2 executes the instructions at addresses 3 and 5,
— Execution of source line 3 executes the instruction at address 2,
— Execution of source line 4 executes the instruction at address 6,
— and so on ...

Step Next

The Step Next command steps to the next source statement whose instructions follow the current instruction. Figure 11.3 shows the results of several Step Next commands on the hypothetical code.

Figure 11.3 Examples of Step Next



Execution starts with the code at **address 1**. This means that source **line 1** has been executed.

1. Step Next command #1:

   a. Does not stop at **address 2**.

      The **address 2** code comes from source **line 3**, which is not the closest unexecuted line to **line 1**. Program execution continues with code at **address 2**, which means that source **line 2** has been executed.

   b. Stops on **address 3**.

      The **address 3** code comes from source **line 2**, which is the closest unexecuted line to **line 1**.

2. Step Next command #2:
    a. Executes code at **address 3**, which means that source **line 2** has been executed.
    b. Does not stop at **address 4** or **address 5**.

       The **address 4** code comes from source **line 8**, which is not the closest unexecuted line to **line 3**. The **address 5** code comes from **line 2**, which already has been executed. Program execution continues with code at **address 4**, and **address 5**, which means that source **line 8** has been executed.
    c. Stops on **address 6**.

       The **address 6** code comes from source **line 4**, which is the closest unexecuted line to **line 2**.
3. Step Next command #3:
    a. Executes code at **address 6**, which means that source **line 4** has been executed.
    b. Stops on **address 7**.

       The **address 7** code comes from source **line 5** and source **line 7**; **line 5** is the closest unexecuted line to **line 4**.
4. Step Next command #4:
    a. Executes code at **address 7**, which means that source **line 5** has been executed.
    b. Stops on **address 8**.

       The **address 8** code comes from source **line 9**, which is the closest unexecuted line to **line5**.

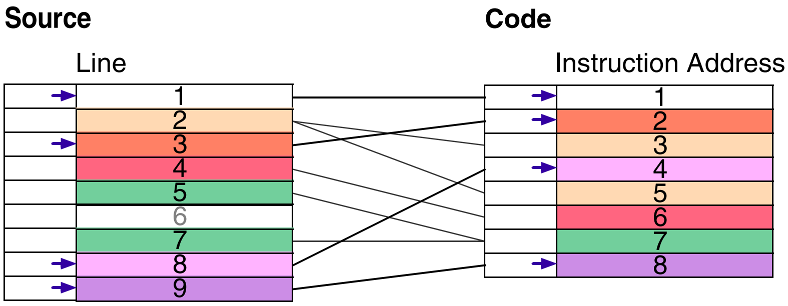
NOTE In this example, stepping from either **line5** or **line7** would stop at **address 8**, because **line6** would be optimized out. In other cases, however, the distinction between **line 5** and **line 7** could be important. The OCD algorithm selects the lowest line number unless you specify otherwise. To specify a different line, click its number in the CMV source pane.

Summary: The four Step Next commands step to addresses 3, 6, 7, and 8.

Step Forward

The Step Forward command steps to the next instruction address whose statement follows the current statement. Figure 11.4 depicts the results of several Step Forward commands on the hypothetical code.

Figure 11.4 Examples of Step Forward



Execution starts with the code at **address 1**. This means that source **line 1** has been executed.

1. Step Forward command #1 stops at **address 2**.

   The **address 2 code** comes from source **line3**, which follows **line1**.

2. Step Forward command #2:

   a. Executes code at **address 2**, which means that source **line 3** has been executed.

   b. Does not stop at **address 3**.

   The **address 3** code comes from source **line 2**, which does not follow **line 3**. Program execution continues with code at **address 3**, which means that source **line 2** has been executed.

   c. Stops at **address 4**.

   The **address 4** code comes from **line 8**, which does follow **line 3**.

3. Step Forward command #3:

   a. Executes code at **address 4**, which means that source **line 8** has been executed.

   b. Does not stop at **address 5**, **address 6**, or **address 7**.

   The **address 5** code comes from source **line 2**, which already has been executed. The **address 6** code comes from source **line 4**, which does not follow **line 8**. The **address 7** code comes from source **line 5** and source **line 7**, which do not follow **line 8**. Program execution continues with code at **address 5**., **address 6**, and **address 7**. This means that source **line 4**, **line 5**, and **line 7** have been executed.

   c. Stops at **address 8**.

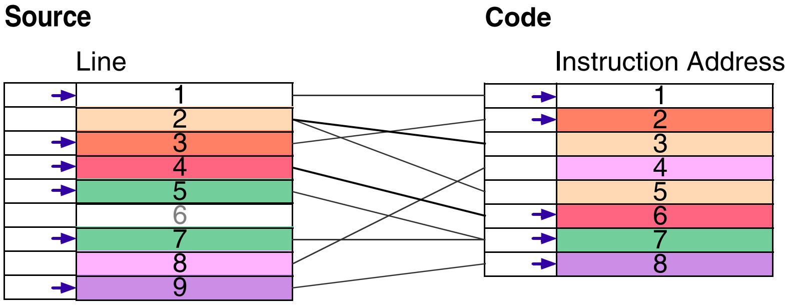
   The **address 8** code comes from source **line 9**, which does follow **line 8**.

Summary: The three Step Forward commands step to addresses 2, 4, and 8.

Step After End of Statement

The Step After End of Statement command steps to the first instruction of the next source statement whose instruction has not yet been executed. Figure 11.5 depicts the results of several Step After End of Statement commands on the hypothetical code.

Figure 11.5 Examples of Step After End of Statement



Execution starts with the code at **address 1**. This means that source **line 1** has been executed.

1. Step After End of Statement command #1 stops at **address 2**.

   Address 2 starts the first code that follows **line 1**'s instructions. Furthermore, the **address 2** code comes from **line 3**, which follows **line 1**.

2. Step After End of Statement command #2.

   a. Executes code at **address 2**, which means that source **line 3** has been executed.

   b. Does not stop at **address 3**, **address 4**, or **address 5**.

   The **address 3** and **address 5** code comes from source **line 2**, which does not follow **line 3**. The **address 4** code comes from source **line 8**, which does follow source **line 3**, but the **address 4** code does not follow all the **line 2** code. Program execution continues with code at **address 3**, **address 4**, and **address 5**. This means that source **line 2** and source **line 8** have been executed.

   c. Stops at **address 6**.

   Address 6 starts the first code that follows all of **line3**'s instructions. Furthermore, the **address 6** code comes from source **line 4** , which follows **line 3**.

3. Step After End of Statement command #3:

   a. Executes code at **address 6**, which means that source **line 4** has been executed.

   b. Stops at **address 7**.

   Address 7 starts the first code that follows **line 4**'s instructions. Furthermore, the **address 7** code comes from source **line 5** and **line 7**, which follow **line 4**.

4. Step After End of Statement command #4:
   a. Executes code at **address 7**, which means that source **line 5** and **line 7** have been executed.
   b. Stops at **address 8**.

      Address 8 starts the first code that follows **line 5**'s instructions. Furthermore, the **address 8** code comes from **line 9**, which follows **line 5**.)

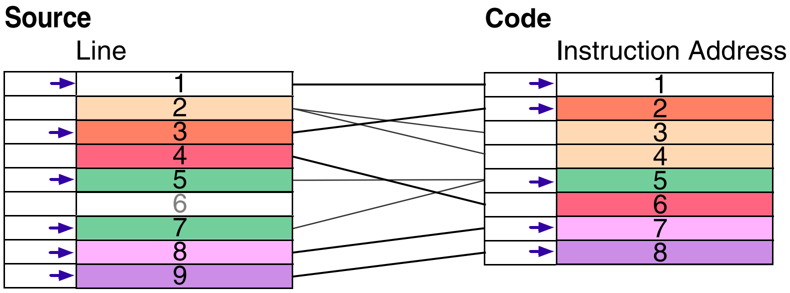
Summary: The four Step After End of Statement commands step to addresses 2, 6, 7, and 8.

Step After All Previous

The Step After All Previous command steps to the next statement that is free from the effects of still-executing instructions. Figure 11.6 depicts the results of several Step After All Previous commands on hypothetical code.

NOTE The hypothetical code relationship of Figure 11.6 differs from that of the earlier figures:
- Execution of source line 2 executes the instructions at addresses 3 and 4,
- Execution of source lines 5 and 7 executes the instructions at address 5, and
- Execution of source line 8 executes the instruction at address 7.

Figure 11.6 Examples of Step After All Previous



Execution starts with the code at **address 1**. This means that source **line 1** has been executed.

1. Step After All Previous command #1 stops at **address 2**.

   Address 2 starts the first code that follows **line 1**'s instructions. Furthermore, the **address 2** code comes from **line 3**, which follows **line 1**.

2. Step After All Previous command #2:
   a. Executes code at **address 2**, which means that source **line 3** has been executed.
   b. Does not stop at **address 3** or **address 4**.

      The **address 3** and **address 4** code comes from source **line 2**, which does not follow **line 3**. Furthermore, **line 2's** instructions still are executing. Program execution continues with code at **address 3** and **address 4**; only then have all of source **line 2**'s instructions been executed.
   c. Stops at **address 5**.

      Address 5 starts the first code that follows all the instructions of **line 3** and **line 2**. Furthermore, the **address 5** code comes from source **line 5** and **line 7**, which follow **line 3**.
3. Step After All Previous command #3:
   a. Executes code at **address 5**, which means that source **line 5** and **line 7** have been executed,
   b. Does not stop at **address 6**.

      The **address 6** code comes from source **line 4**, which does not follow **line 5** and **line 7**. Furthermore, **line 4**'s instructions still are executing. Program execution continues with code at **address 6**; only then have all of source **line 4**'s instructions been executed.
   c. Stops at **address 7**.

      Address 7 starts the first code that follows all the instructions of **line 3**, **line 2**, and **line 4**. Furthermore, the **address 7** code comes from source **line 8**, which follows **line 7**.
4. Step After All Previous command #4:
   a. Executes code at **address 7**, which means that source **line 8** has been executed.
   b. Stops at **address 8.**

      Address 8 starts the first code that follows all the instructions of **line 3**, **line 2**, **line 4**, and **line 8**. Furthermore, the **address 8** code comes from **line 9**, which follows **line 8**.

Summary: The four Step After All Previous commands step to addresses 2, 5, 7, and 8.

12

High-Speed Simultaneous Transfer and Data Visualization

This chapter explains how to use the CodeWarrior™ High-Speed Simultaneous Transfer (HSST) and Data Visualization features.

The sections are:

- HSST
- Data Visualization

HSST

High-Speed Simultaneous Transfer (HSST) facilitates data transfer between low-level targets (hardware or simulator) and host-side client applications. The data transfer occurs without stopping the core. The host-side client application must be an IDE plug-in or a script run from the command-line debugger.

To use HSST, launch the target side application from the debugger. The debugger automatically enables HSST communications as required.

- Host-Side Client Interface
- Target Library Interface

Host-Side Client Interface

This section documents the API calls for using High-Speed Simultaneous Transfer (HSST) from your host-side client application.

hsst_open

A host-side client application uses this function to open a communication channel with the low-level target. Trying to open a channel that already is open results in the same channel ID being returned.

```
HRESULT hsst_open(
     const char*       channel_name,
     size_t*           cid);
```

Parameters

`channel_name`

Communication channel name.

`cid`

Identifier associated with the communication channel.

Returns

`S_OK` if the call succeeds; `S_FALSE` if the call fails.

hsst_close

A host-side client application uses this function to close a communication channel with the low-level target.

```
HRESULT hsst_close( size_t channel_id );
```

Parameter

`channel_id`

Communication channel identifier.

Returns

`S_OK` if the call succeeds; `S_FALSE` if the call fails.

hsst_read

A host-side client application uses this function to read data sent by the target application without stopping the core.

```
HRESULT hsst_read(
     void*       data,
     size_t      size,
     size_t      nmemb,
     size_t      channel_id,
     size_t*     read);
```

Parameters

`data`

Data buffer that receives the data.

`size`

Size of the individual data elements to be read.

`nmemb`

Number of data elements to be read.

`channel_id`

Identifier of the communication channel from which data is read.

`read`

Number of data elements read so far.

Returns

`S_OK` if the call succeeds; `S_FALSE` if the call fails.

hsst_write

A host-side client application uses this function to write data that the target application can read without stopping the core.

```
HRESULT hsst_write(
     void*     data,
     size_t    size,
     size_t    nmemb,
     size_t    channel_id,
      size_t*   written);
```

Parameters

`data`

Buffer that holds the data to be written.

`size`

Size of the individual data elements to be written.

`nmemb`

Number of data elements to be written.

`channel_id`

Identifier of the communication channel to receive the written data.

`written`

Number of data elements written so far.

Returns

`S_OK` if the call succeeds; `S_FALSE` if the call fails.

hsst_size

A host-side client application uses this function to determine the size of unread data (in bytes) in the communication channel.

```
HRESULT hsst_size(
      size_t       channel_id,
      size_t*      unread);
```

Parameters

`channel_id`

Identifier of the applicable communication channel.

`unread`

Number of bytes of unread data.

Returns

`S_OK` if the call succeeds; `S_FALSE` if the call fails.

hsst_block_mode

A host-side client application uses this function to set blocked mode for the specified communication channel. This delays implementation of calls to read from this channel until the requested amount of data is available. Blocked mode is the default setting.

```
HRESULT hsst_block_mode( size_t channel_id );
```

Parameter

`channel_id`

Identifier of the communication channel.

Returns

`S_OK` if the call succeeds; `S_FALSE` if the call fails.

hsst_noblock_mode

A host-side client application uses this function to set non-blocked mode for the specified communication channel. This means that, for calls to read from this channel, there is no delay for data availability.

```
HRESULT hsst_noblock_mode( size_t channel_id );
```

Parameter

`channel_id`

Identifier of the communication channel.

Returns

`S_OK` if the call succeeds; `S_FALSE` if the call fails.

hsst_attach_listener

A host-side client application uses this function to attach itself as a listener to the specified communication channel. The client application receives notification any time this channel has data available to be read.

```
HRESULT hsst_attach_listener(
     size_t                cid,
    NotifiableHSSTClient* subscriber);
```

Parameters

`cid`

Identifier of the communication channel.

`subscriber`

Pointer to an instance of class `NotifiableHSSTClient`.

Remarks

To notify the client application that data is available on the specified channel, HSST calls the `Update` function:

```
void NotifiableHSSTClient::Update(
    size_t    descriptor,
    size_t    size,
    size_t    nmemb);
```

Returns

The `hsst_attach_listener` function returns `S_OK` if the call succeeds; `S_FALSE` if the call fails.

hsst_detach_listener

A listener host-side client application uses this function to detach itself from the specified communication channel.

```
HRESULT hsst_detach_listener( size_t cid );
```

Parameter

`cid`

Identifier of the communication channel.

Returns

`S_OK` if the call succeeds; `S_FALSE` if the call fails.

hsst_set_log_dir

A host-side client application uses this function to set a log directory for the specified communication channel.

```
HRESULT hsst_set_log_dir(
     size_t         cid,
     const char*    log_directory);
```

Parameter

`cid`

Identifier of the communication channel.

`log_directory`

Path to the directory in which to store temporary log files.

Remarks

This function lets the host-side client application use data logged from a previous High-Speed Simultaneous Transfer (HSST) session, instead of reading data directly from the board.

After the initial call to `hsst_set_log_dir`, the IDE examines the specified directory for logged data associated with the relevant channel. Only after all the data has been read from the file, do additional reads come from the board.

To stop reading logged data, the host-side client application calls `hsst_set_log_dir` with `NULL` as its argument. This call only affects host-side reading.

Returns

`S_OK` if the call succeeds; `S_FALSE` if the call fails.

Target Library Interface

This section describes the API calls for using High-Speed Simultaneous Transfer (HSST) from your target application.

HSST_open

A target application uses this function to open a bidirectional communication channel with the host. The default setting is for the function to open an output channel in buffered mode. Trying to open a channel that already is open results in the same channel ID being returned.

```
HSST_STREAM* HSST_open( const char* stream );
```

Parameter

`stream`

Communication channel name.

Returns

The stream associated with the opened channel.

HSST_close

A target application uses this function to close a communication channel with the host.

```
int    HSST_close( HSST_STREAM* stream );
```

Parameter

`stream`

Pointer to the communication channel.

Returns

`0` if the call succeeds; `1` if the call fails.

HSST_setvbuf

A target application uses this function for any of these actions:

- Specifying buffered mode for a channel opened in write mode. (This can improve performance greatly.)
- Resizing the buffer in a buffered channel opened in write mode
- Providing an external buffer for a channel opened in write mode
- Changing buffering to unbuffered mode.

```
int HSST_setvbuf(
     HSST_STREAM*     rs,
     unsigned char*   buf,
     int              mode,
     size_t           size);
```

Parameters

`rs`

Pointer to the communication channel.

`buf`

Passes a pointer to an external buffer.

`mode`

Passes the buffering mode: buffered (`HSSTFBUF`) or unbuffered (`HSSTNBUF`).

`size`

Passes the size of the buffer.

Remarks

You can use this function only for an open the channel. Contents of internal or external buffers at any time are indeterminate.

Returns

`0` if the call succeeds; `1` if the call fails.

HSST_write

A target application uses this function to write data for the host-side client application to read.

```
size_t HSST_write(
     void*         data,
     size_t        size,
     size_t        nmemb,
     HSST_STREAM*  stream);
```

Parameters

`data`

Passes a pointer to the data buffer holding the data to be written.

`size`

Passes the size of the individual data elements to be written.

`nmemb`

Passes the number of data elements to be written.

`stream`

Passes a pointer to the communication channel.

Returns

The number of data elements written.

HSST_read

A target application uses this function to read data sent by the host.

```
size_t HSST_read(
     void*        data,
     size_t       size,
     size_t       nmemb,
     HSST_STREAM* stream);
```

Parameters

`data`

Passes a pointer to the data buffer into which to read the data.

`size`

Passes the size of the individual data elements to be read.

`nmemb`

Passes the number of data elements to be read.

`stream`

Passes a pointer to the communication channel.

Returns

The number of data elements read.

HSST_flush

A target application uses this function to flush out data buffered in a buffered output channel.

```
int HSST_flush( HSST_STREAM *stream );
```

Parameter

`stream`

Passes a pointer to the communication channel. (To flush all open buffered communication channels, enter the value null for this parameter.)

Returns

`0` if this call succeeds; `1` if the call fails.

HSST_size

A target application uses this function to determine the number of bytes of unread data for the specified communication channel.

```
size_t HSST_size( HSST_STREAM* stream );
```

Parameter

`stream`

Passes a pointer to the communication channel.

Returns

The number of bytes of unread data.

HSST_raw_read

A target application uses this function to write raw data to a communication channel, without any automatic endianess conversion during communication.

```
size_t HSST_raw_read(
     void*        ptr,
     size_t       length,
     HSST_STREAM* rs);
```

Parameters

`ptr`

Pointer to the destination buffer for the data.

`length`

Buffer size, in bytes.

`rs`

Pointer to the communication channel.

Returns

The number of bytes of raw data read.

HSST_raw_write

A target application uses this function to read raw data from a communication channel, without any automatic endianess conversion during communication).

```
size_t HSST_raw_write(
     void*        ptr,
     size_t       length,
     HSST_STREAM* rs);
```

Parameters

`ptr`

Pointer to the buffer that holds the data to be written.

`length`

Buffer size, in bytes.

`rs`

Pointer to the communication channel.

Returns

The number of data elements written.

HSST_set_log_dir

A target application uses this function to set the host-side directory for storing temporary log files.

```
int HSST_set_log_dir(
     HSST_STREAM*    stream,
     char*           dir_name);
```

Parameters

`stream`

Passes a pointer to the communication channel.

`dir_name`

Passes a pointer to the path to the directory in which to store temporary log files.

Remarks

Calling this function overwrites previously existing logs. Logging stops when you close channel call this function with a null parameter value. The host-side function `HSST_set_log_dir` can use these logs.

Returns

`S_OK` if the call succeeds; `S_FALSE` if the call fails.

Data Visualization

Data visualization lets you graph variables, registers, regions of memory, and HSST data streams as they change over time.

The Data Visualization tools can plot memory data, register data, global variable data, and HSST data.

- Starting Data Visualization
- Data Target Dialog Boxes
- Graph Window Properties

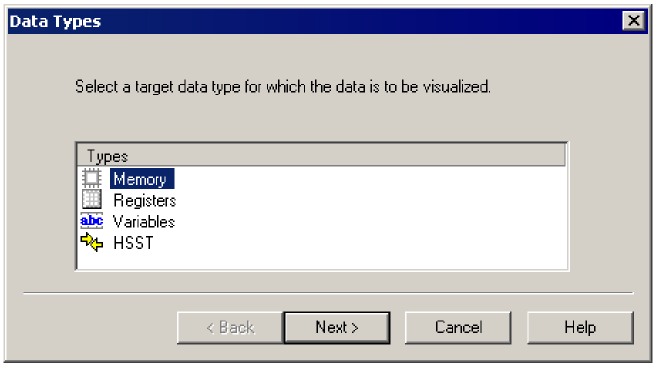
Starting Data Visualization

To start the Data Visualization tool:

1. Start a debug session.
2. Select **Data Visualization > Configurator**

   The **Data Types** window appears. (See Figure 12.1.)

Figure 12.1 Data Types Window



Data Types

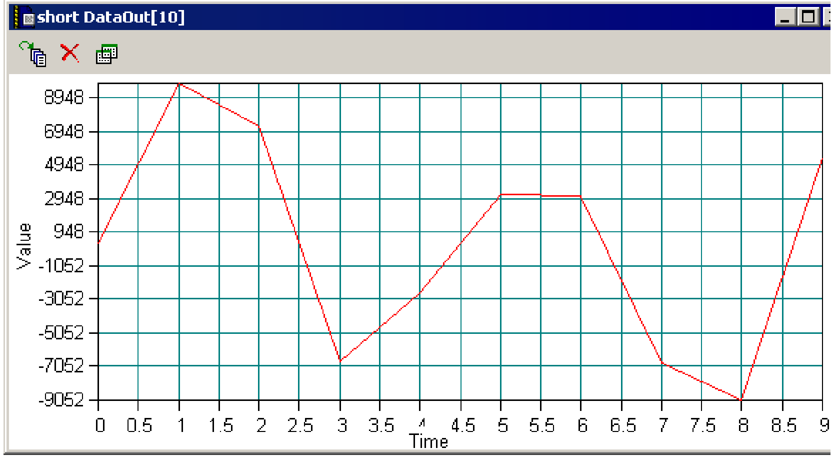
Select a target data type for which the data is to be visualized.

Types
- Memory
- Registers
- Variables
- HSST

3. Select a data target type
4. Click **Next**
5. Configure the data target dialog box and filter dialog box.
6. Run your program to display the data.

   See Figure 12.2.

Figure 12.2 Graph Window



Data Target Dialog Boxes

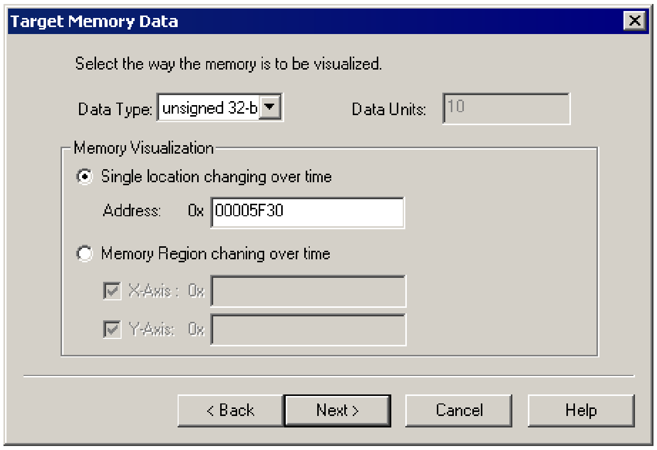
There are four possible data targets. Each target has its own configuration dialog box.

- Memory
- Registers
- Variables
- HSST

Memory

The **Target Memory** dialog box lets you graph memory contents in real-time.

Figure 12.3 Target Memory Dialog Box



Target Memory Data

Select the way the memory is to be visualized.

Data Type: unsigned 32-b

Data Units: 10

Memory Visualization

● Single location changing over time

Address: 0x 00005F30

○ Memory Region chaning over time

☑ X-Axis : 0x

☑ Y-Axis: 0x

Data Type

The Data Type listbox lets you select the type of data to be plotted.

Data Unit

The Data Units text box lets you enter a value for number of data units to be plotted. This option is only available when you select Memory Region Changing Over Time.

Single Location Changing Over Time

The Single Location Changing Over Time option lets you graph the value of a single memory address. Enter this memory address in the Address text box.

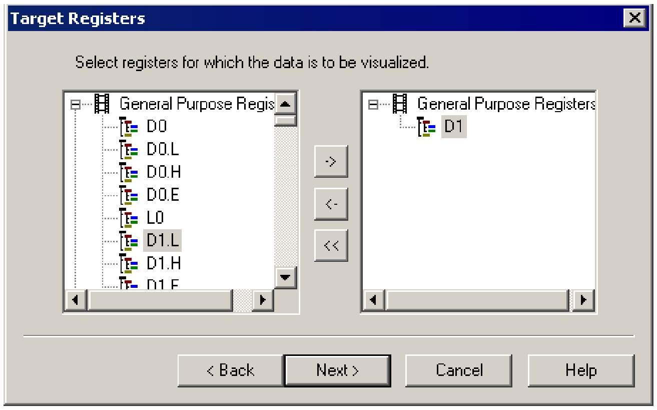
Memory Region Changing Over Time

The Memory Region Changing Over Time options lets you graph the values of a memory region. Enter the memory addresses for the region in the X-Axis and Y-Axis text boxes.

Registers

The **Target Registers** dialog box lets you graph the value of registers in real-time.

Figure 12.4 Target Registers Dialog Box



Target Registers

Select registers for which the data is to be visualized.

General Purpose Regis
- D0
- D0.L
- D0.H
- D0.E
- L0
- D1.L
- D1.H
- D1.E

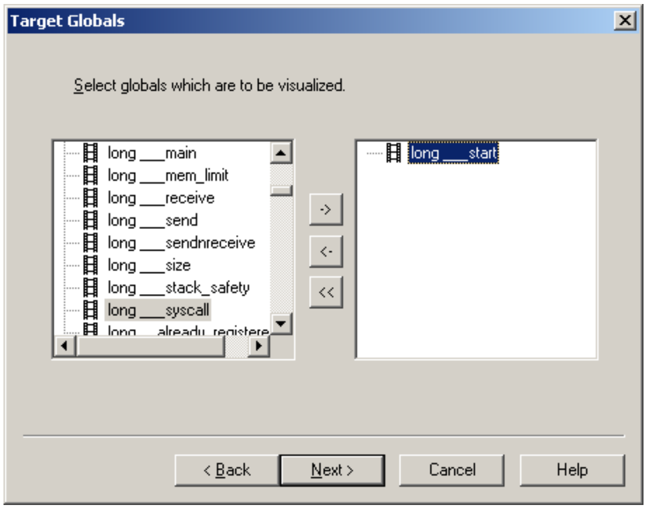
General Purpose Registers
- D1

Select registers from the left column, and click the **->** button to add them to the list of registers to be plotted.

Variables

The **Target Globals** dialog box lets you graph the value of global variables in real-time. (See Figure 12.5.)

Figure 12.5 Target Globals Dialog Box



Target Globals

Select globals which are to be visualized.

- long ___main
- long ___mem_limit
- long ___receive
- long ___send
- long ___sendreceive
- long ___size
- long ___stack_safety
- long ___syscall
- long ___already_registere

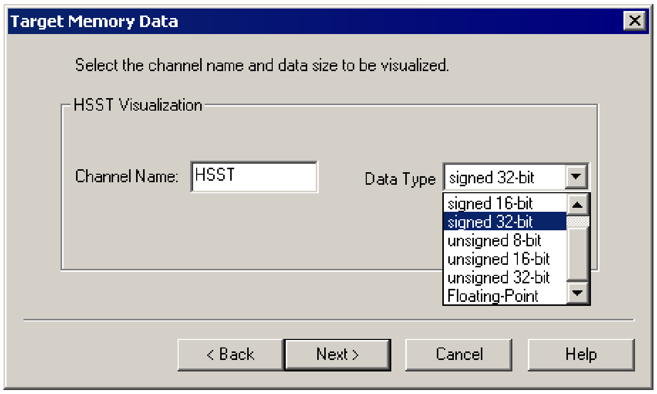
- long ___start

Select global variables from the left column, and click the **->** button to add them to the list of variables to be plotted.

HSST

The **Target HSST** dialog box lets you graph the value of an HSST stream in real-time. (See Figure 12.6.)

Figure 12.6 Target HSST Dialog Box



Target Memory Data

Select the channel name and data size to be visualized.

HSST Visualization

Channel Name: HSST

Data Type: signed 32-bit

Channel Name

The Channel Name text box lets you specify the name of the HSST stream to be plotted.

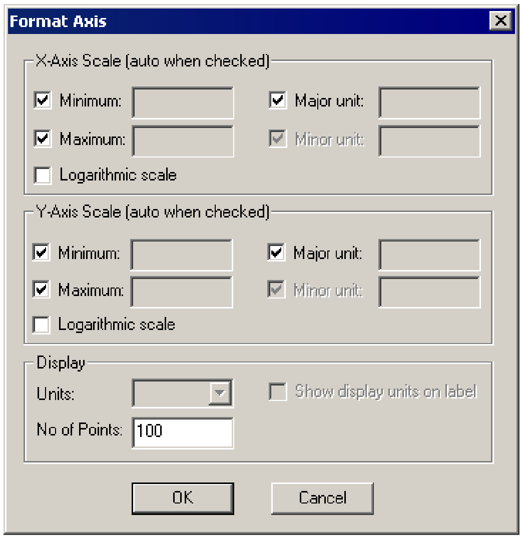
Data Type

The Data Type listbox lets you select the type of data to be plotted.

Graph Window Properties

To change the look of the graph window, click the graph properties button to open the **Format Axis** dialog box.

Figure 12.7 Format Axis Dialog Box



Format Axis

X-Axis Scale (auto when checked)

☑ Minimum: ☑ Major unit:

☑ Maximum: ☑ Minor unit:

□ Logarithmic scale

Y-Axis Scale (auto when checked)

☑ Minimum: ☑ Major unit:

☑ Maximum: ☑ Minor unit:

□ Logarithmic scale

Display

Units: □ Show display units on label

No of Points: 100

Scaling

The default scaling settings of the data visualization tools automatically scale the graph window to fit the existing data points.

To override automatic scaling, uncheck a scaling checkbox to enable the text box and enter your own value.

To scale either axis logarithmically, enable the Logarithmic Scale option of the corresponding axis.

Display

The Display settings let you change the maximum number of data points that are plotted on the graph.

13

Debugger Communications Protocols

The CodeWarrior™ debugger can communicate with StarCore® devices in several ways.

Table 13.1 lists each StarCore device along with the communications protocol the debugger supports for the device.

Table 13.1 Communication Protocols by Target Device

StarCore® Device	CCS	MetroTRK	Simulator
SC140	x		
SC140 Simulator			x
MSC8101	x	x	
MSC8102	x		
MSC8102 Simulator			x

This chapter describes the following communications protocols.

- Command Converter Server
- Metrowerks™ Target Resident Kernel
- Simulator

Command Converter Server

The command converter server (CCS) provides a TCP/IP connection point for debugger communications. Running a CCS on your host computer lets you share access to your target board with remote users of the CodeWarrior debugger. Conversely, you have access to the target board of any remote computer running a CCS instance, provided that you know its IP address and CCS port number.

- Creating a CCS Remote Connection
- Running CCS

- The CCS Console
- Configuring a CCS Connection

Creating a CCS Remote Connection

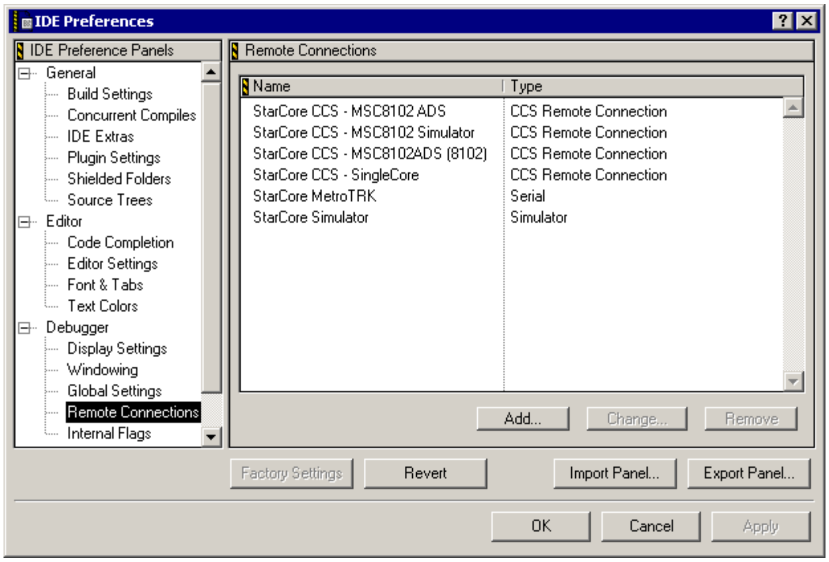
Before you can debug programs using CCS, you must have a remote connection for CCS in the CodeWarrior IDE. The CodeWarrior installer creates several CCS remote connections that you may edit as necessary. Or, if you do not wish to change the default connections, you may add a new remote connection.

To add a new remote connection:

1. Select **Edit > Preferences** from the IDE main menu.

   The **IDE Preferences** window appears. (See Figure 13.1.)

Figure 13.1 IDE Preferences window



Name	Type
StarCore CCS - MSC8102 ADS	CCS Remote Connection
StarCore CCS - MSC8102 Simulator	CCS Remote Connection
StarCore CCS - MSC8102ADS (8102)	CCS Remote Connection
StarCore CCS - SingleCore	CCS Remote Connection
StarCore MetroTRK	Serial
StarCore Simulator	Simulator

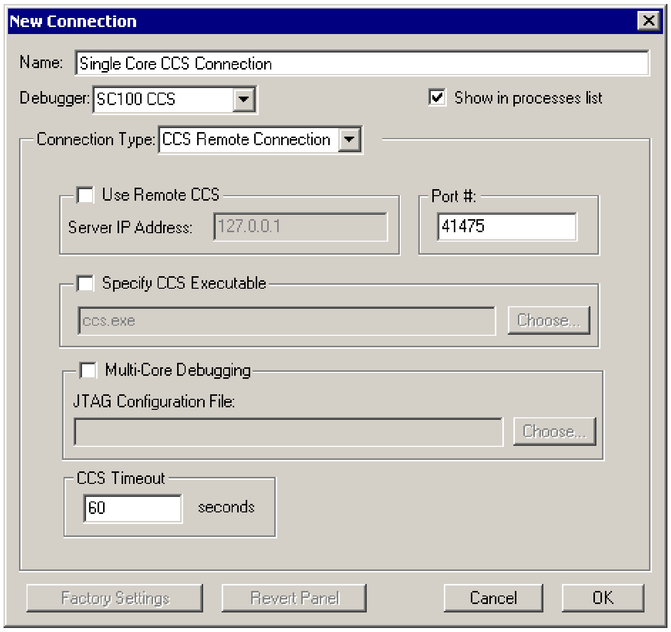
2. From the list on the left side of the window, select Remote Connections.

   The **Remote Connections** preference panel appears.

3. Click **Add**

   The **New Connection** dialog box appears.

Figure 13.2 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, type the name of the new connection.
5. From the Debugger listbox, select to SC100 CCS.
6. From the Connection Type listbox, select CCS Remote Connection.
7. In the Port # text box, type the port number on which CCS listens for messages.

NOTE If you are not sure of this port number, try the default, 41475.

8. If you are connecting to a CCS instance on a different machine, check the Use Remote CCS box and specify the IP address of the remote machine.
9. If CCS is connected to a multi-core target (such as the MSC8102), check the Multi-Core Debugging box and select the appropriate JTAG configuration file.

Running CCS

The CodeWarrior IDE automatically starts CCS if it is not running when you try to debug using a local CCS connection. You can also run CCS yourself—the executable is here:

```
installDir\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. This menu provides these options:

- Show Console

  Displays the CCS console.

- Hide console

  Hides the CCS console.

- About CCS

  Displays version information.

- Quit CCS

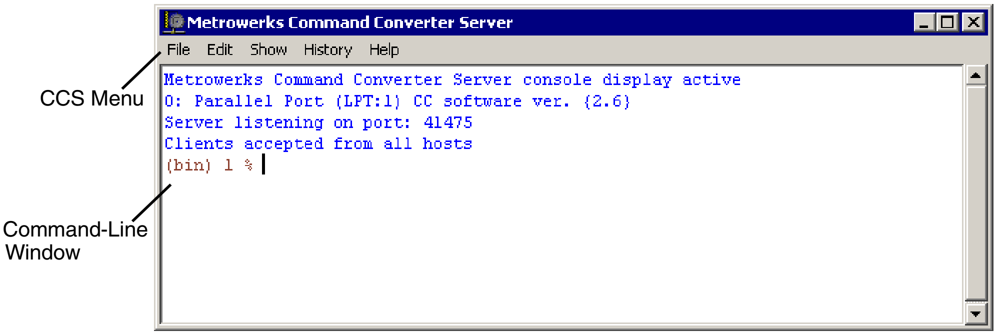
  Terminates CCS.

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 CCS menu.

Figure 13.3 shows the CCS console.

Figure 13.3 The Command Converter Server Console



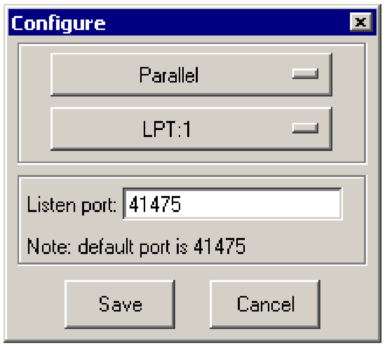
```
Metrowerks Command Converter Server console display active
0: Parallel Port (LPT:1) CC software ver. {2.6}
Server listening on port: 41475
Clients accepted from all hosts
(bin) 1 %
```

Configuring a CCS Connection

CCS is initially configured according to the options you specified during the installation procedure. You can change the properties of the connection between the host computer and the target from either the menu or the command line.

To configure the connection, select **File > Configure**. The **Configure** dialog box appears. (See Figure 13.4.)

Figure 13.4 CCS Configure Dialog Box



Listen port: 41475

Note: default port is 41475

To configure the connection from the command line, use the config command to set the listen port and command converter. Before doing this, you may have to delete the existing configuration with the `delete all` command.

- Server listen port

  ```
  config port 41475
  ```

- Parallel port LPT1 JTAG command converter

  ```
  config cc lpt:1
  ```

- HTI command converter

  ```
  config cc hti:10.1.0.1
  ```

Metrowerks™ Target Resident Kernel

The Metrowerks Target Resident Kernel (MetroTRK) is a debug protocol for the MSC8101 board that allows run control through a serial connection from the host computer to the target board. MetroTRK is provided as an S-Record file, which can be flashed to the MSC8101 board.

- MetroTRK Limitations and Restrictions
- Downloading MetroTRK to the MSC8101 Board
- Remote Debugger Settings for MetroTRK

MetroTRK Limitations and Restrictions

The MetroTRK protocol has some limitations compared to the CCS protocol:

- The MetroTRK protocol does not support HSST.
- The MetroTRK protocol cannot be used to program the flash.
- The MetroTRK protocol always loads the Memory Window in increments of 64 bytes regardless of the word size you select. However, you can still view and modify the Memory Window in any of the selectable word sizes.
- Using the profiler with the MetroTRK protocol significantly slows down debugging.
- The MetroTRK protocol does not support multi-core debugging.

There are several restrictions regarding the type of programs that MetroTRK can debug.

- The user program should not modify the memory used by MetroTRK.
- The interrupt vectors used by MetroTRK should not be accessed by the user program.
- The user program should not execute the instructions that change the status of the core such as: halt, stop, wait, debug. However, a TRAP instruction can be used to stop the core.
- The user program must not execute an initialization file. MetroTRK for the MSC8101 requires its own initialization.
- The user program should not change the clock configuration, including the pctl0 and pctl1 registers of core.

If the user program needs to use other SIC interrupts such as SCC interrupts, the user program must save the original interrupt vector and insure that the SMC interrupt is routed to the original interrupt handler.

Downloading MetroTRK to the MSC8101 Board

There are two MetroTRK S-Record files from which to choose:

- ROM1_Version\metroTRK1.s — smaller RAM footprint, slower speed
- ROM2_Version\metroTRK2.s — larger RAM footprint, faster speed

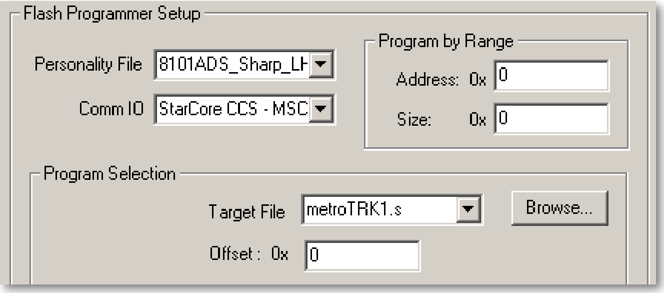
These files are in this directory:

`installDir\StarCore_Tools\MetroTRK\S_Records`

ROM1_Version\metroTRK1.s

The `.text` segment of this S-Record file resides in flash memory. It uses less RAM than `metroTRK2.s`, but supports a lower maximum communications speed. To load this S-Record file:

1. Program the flash with `metroTRK1.s` at offset 0x0.



Flash Programmer Setup
Personality File: 8101ADS_Sharp_LH
Comm IO: StarCore CCS - MSC
Program by Range
Address: 0x 0
Size: 0x 0
Program Selection
Target File: metroTRK1.s
Offset : 0x 0

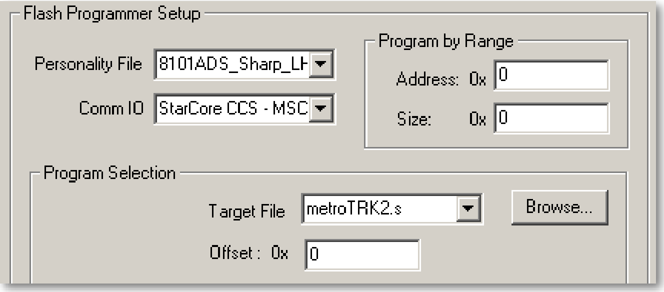
2. Disconnect power from the board.
3. Disconnect from the JTAG header.
4. Connect to the upper serial port RS232 (-2).
5. Set Switch 10-1 to OFF
6. Set Switch 9-7 to ON
7. Connect power to the board.

   LD11 and LD17 should light.

ROM2_Version\metroTRK2.s

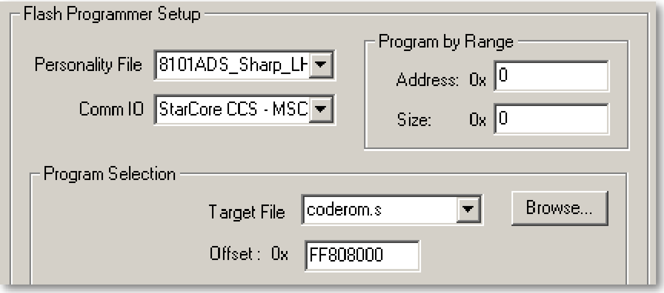
The `.text` segment of this S-Record file resides in SRAM. It uses more RAM than `metroTRK1.s`, but supports a faster communications speed. To load this S-Record file:

1. Program the flash with `metroTRK2.s` at offset `0x0`.



Flash Programmer Setup
Personality File: 8101ADS_Sharp_LH
Comm IO: StarCore CCS - MSC
Program by Range
Address: 0x 0
Size: 0x 0
Program Selection
Target File: metroTRK2.s
Offset : 0x 0

2. Program the flash with `coderom.s` at offset `0xFF808000`.



Flash Programmer Setup

Personality File: 8101ADS_Sharp_LH

Comm IO: StarCore CCS - MSC

Program by Range

Address: 0x 0

Size: 0x 0

Program Selection

Target File: coderom.s

Offset : 0x FF808000

3. Disconnect power from the board.
4. Disconnect from the JTAG header
5. Connect to the upper serial port RS232 (-2).
6. Set Switch 10-1 to OFF
7. Set Switch 9-7 to ON
8. Connect power to the board.

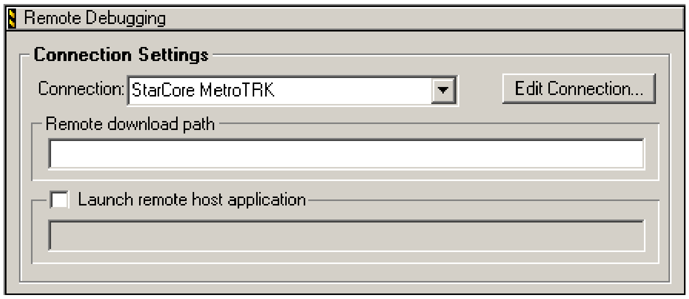
   LD11 and LD17 should light.

Remote Debugger Settings for MetroTRK

To use MetroTRK as the debug protocol, you must create a remote debugger setting for it.

1. Select Remote Debugging in the **Target Settings** window.

   The **Remote Debugging** target settings panel appears.



Remote Debugging

Connection Settings

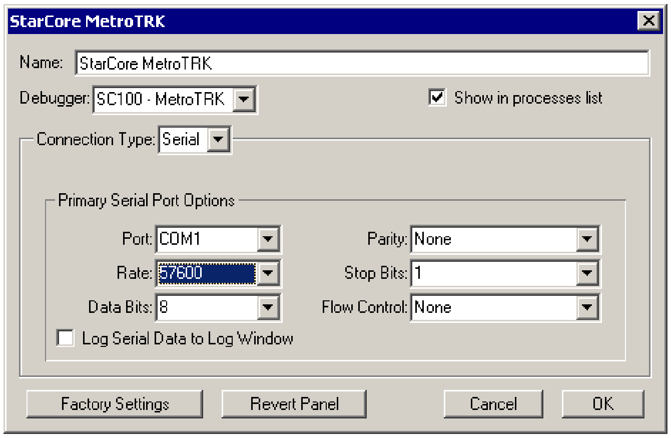
Connection: StarCore MetroTRK

Remote download path

☐ Launch remote host application

2. Check Enable Remote Debugging box and select the StarCore MetroTRK connection.
3. On the **Remote Debugging** panel, click **Edit Connection**

   The **StarCore MetroTRK** remote connection dialog box appears.



StarCore MetroTRK

Name: StarCore MetroTRK

Debugger: SC100 - MetroTRK

☑ Show in processes list

Connection Type: Serial

Primary Serial Port Options

Port: COM1

Rate: 57600

Data Bits: 8

Parity: None

Stop Bits: 1

Flow Control: None

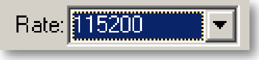
☐ Log Serial Data to Log Window

4. In the **StarCore MetroTRK** dialog box:

   - If you are running `metroTRK1.s`, set Rate to 57600.

Rate: 57600

   - If you are running `metroTRK2.s`, set Rate to 115200.



Rate: 115200

5. You can now debug or run your program using the MetroTRK protocol connection.

Simulator

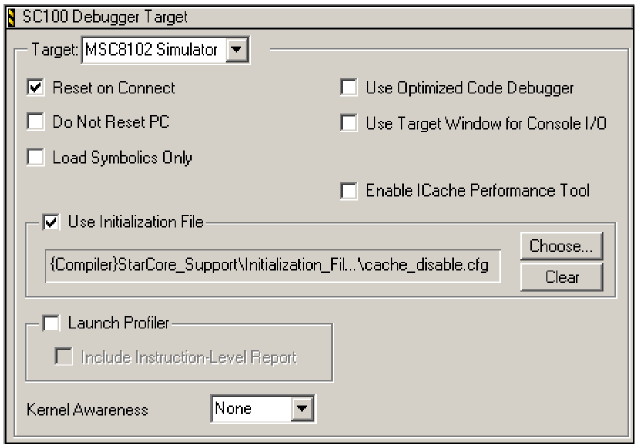
In the absence of a hardware target, you can debug using either the SC100 or the MSC8102 simulator. The CodeWarrior IDE ships with two preconfigured remote connections for these simulators.

- MSC8102 Simulator
- SC100 Simulator

MSC8102 Simulator

The MSC8102 simulator simulates the multi-core environment of the MSC8102 ADS evaluation board. If you are using the MSC8102 simulator as the target (Figure 13.5), the CCS options defined in the remote connection (Figure 13.6) are ignored in favor of hard coded values. The hard coded values run CCSSim (the simulator version of CCS) on port 41476.

Figure 13.5 SC100 Debugger Target for MSC8102 Simulator



SC100 Debugger Target

Target: MSC8102 Simulator

☑ Reset on Connect

☐ Do Not Reset PC

☐ Load Symbolics Only

☐ Use Optimized Code Debugger

☐ Use Target Window for Console I/O

☐ Enable ICache Performance Tool

☑ Use Initialization File

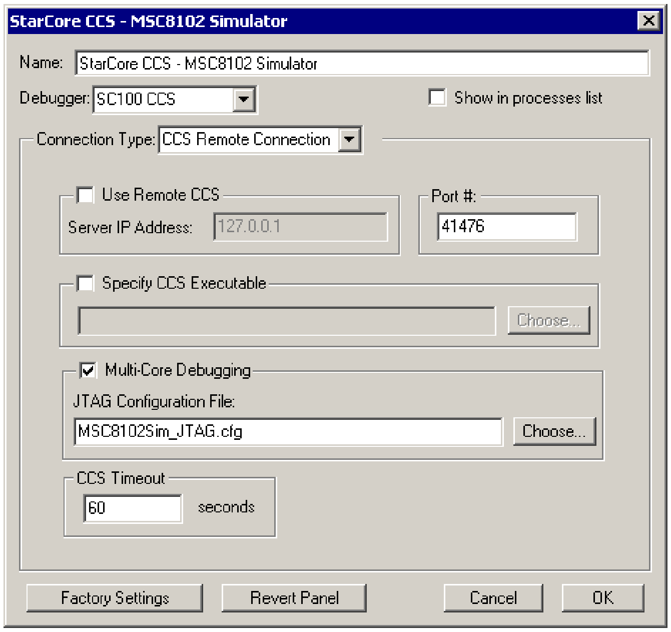
{Compiler}StarCore_Support\Initialization_Fil...\cache_disable.cfg

☐ Launch Profiler

☐ Include Instruction-Level Report

Kernel Awareness: None

Figure 13.6 Remote Connection for MSC8102 Simulator



StarCore CCS - MSC8102 Simulator

Name: StarCore CCS - MSC8102 Simulator

Debugger: SC100 CCS

☐ Show in processes list

Connection Type: CCS Remote Connection

☐ Use Remote CCS

Server IP Address: 127.0.0.1

Port #: 41476

☐ Specify CCS Executable

☑ Multi-Core Debugging

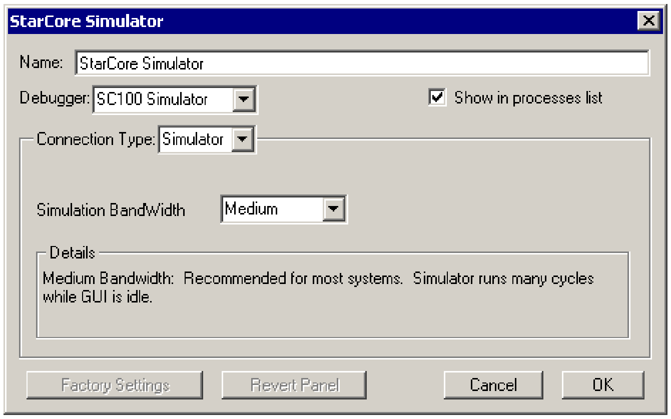
JTAG Configuration File: MSC8102Sim_JTAG.cfg

CCS Timeout: 60 seconds

SC100 Simulator

The SC100 simulator simulates a single core SC100 chip. The remote connection settings panel shown in Figure 13.7 let you change the CPU priority of the simulator.

Figure 13.7 Remote Connection for SC100 Simulator



StarCore Simulator

Name: StarCore Simulator

Debugger: SC100 Simulator

☑ Show in processes list

Connection Type: Simulator

Simulation BandWidth Medium

Details

Medium Bandwidth: Recommended for most systems. Simulator runs many cycles while GUI is idle.

14

StarCore® DSP Utilities

This chapter explains how to use the utility programs included in CodeWarrior™ Development Studio for StarCore DSP Architectures product.

The sections are:

- Flash Programmer
- ELF Dump Utility
- ELF to LOD Utility
- ELF to S-Record Utility
- Archiver Utility
- disasmsc100 Disassembler
- Name Utility
- Size Utility
- Statistics Utility

Flash Programmer

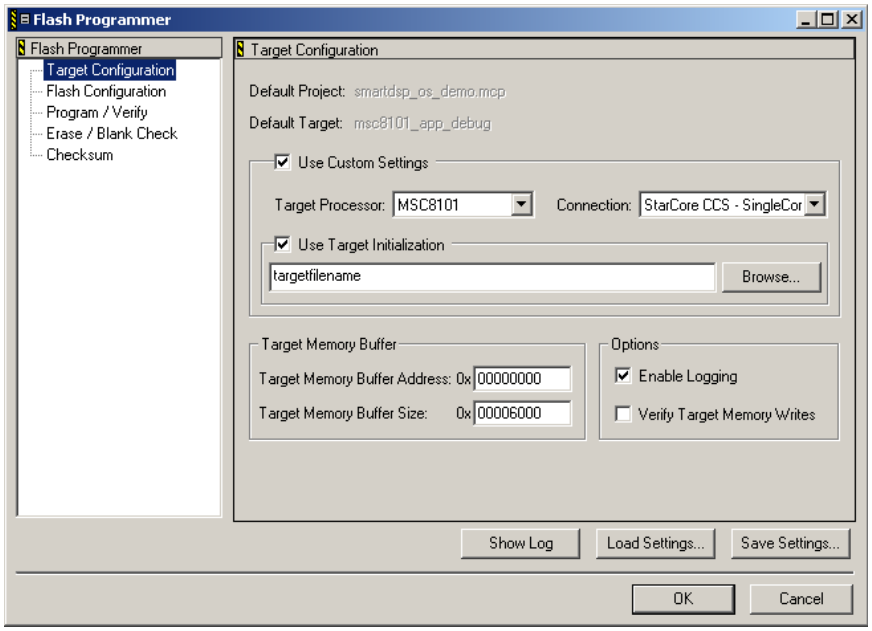
The CodeWarrior flash programmer lets you manipulate the flash memory of a StarCore board from within the CodeWarrior IDE. Specifically, the flash programmer can perform these functions:

- Program
- Erase
- Blank Check
- Verify
- Checksum

NOTE Common flash programmer features (such as view/modify memory, view/modify register, and save memory range to a file) are provided by the CodeWarrior debugger. As a result, the CodeWarrior flash programmer does not include these features.

To display the **Flash Programmer** window, select **Tools > Flash Programmer**. (See Figure 14.1.)

Figure 14.1 Flash Programmer Window



Default Project: smartdsp_os_demo.mcp
Default Target: msc8101_app_debug

☑ Use Custom Settings

Target Processor: MSC8101
Connection: StarCore CCS - SingleCor

☑ Use Target Initialization

targetfilename

Target Memory Buffer

Target Memory Buffer Address: 0x 00000000
Target Memory Buffer Size: 0x 00006000

Options

☑ Enable Logging
☐ Verify Target Memory Writes

For general instructions that explain how to use the CodeWarrior flash programmer, refer to the *IDE User Guide*.

StarCore-Specific Flash Programmer Information

The following sections provide the StarCore-specific information needed to use the flash programmer with the supported StarCore evaluation boards.

- Supported Board/Flash Module Combinations
- Flash Programmer/Board Setup for the 8101ADS
- Flash Programmer/Board Setup for the 8102ADS—Host Side
- Flash Programmer/Board Setup for the 8102ADS—Slave Side
- 8102ADS Eval Board—Boot Sector Restrictions

Supported Board/Flash Module Combinations

Table 14.1 lists the StarCore board/flash module combinations with which you can use the CodeWarrior flash programmer.

Table 14.1 Supported StarCore® Board/Flash Module Combinations

Board	Flash Module
8101ADS	LH28F016SCZ4
8102ADS	AM29LV320DB

Flash Programmer/Board Setup for the 8101ADS

To program flash memory on an 8101ADS evaluation board, use this setup:

- Target Configuration page of the **Flash Programmer** window:
  - Target Processor: MSC8101
  - Connection: StarCore CCS - Single Core
  - Target Initialization File: `*installDir*\StarCore_Support\Initialization_Files\RegisterConfigFiles\MSC8101\8101_Initialization.cfg`
- Board configuration: no board-level adjustments required

Flash Programmer/Board Setup for the 8102ADS—Host Side

To program the flash memory of the 8101 chip on the 8102ADS evaluation board, use this setup:

- Target Configuration page of the **Flash Programmer** window:
  - Target Processor: MSC8101
  - Connection: StarCore CCS - Single Core
  - Target Initialization File: `*installDir*\StarCore_Support\Initialization_Files\RegisterConfigFiles\MSC8102ADS\8101Only_32bit\8102ADS_DSI32_Host_Init.cfg`

- Board configuration:
  - JTAG connector: P14
  - Switch settings:
    SW7.1 ON
    SW7.2 ON
    SW7.3 ON

Flash Programmer/Board Setup for the 8102ADS—Slave Side

To program the flash memory of the 8102 chip on the 8102ADS evaluation board, use this setup:

- Target Configuration page of the **Flash Programmer** window:
  - Target Processor: MSC8102
  - Connection: StarCore CCS - MSC8102ADS (8102)
  - Target Initialization File:
    `*installDir*\StarCore_Support\Initialization_Files\RegisterConfigFiles\MSC8102ADS\8102Only_32bitSysMode\8102ADS_DSI32_Slave_Init.cfg`
- Board configuration:
  - JTAG connector: P15
  - Switch settings:
    SW7.1 ON
    SW7.2 ON
    SW7.3 ON
    SW4.3 OFF
  - Remove 8101 chip from socket on board

8102ADS Eval Board—Boot Sector Restrictions

By default, the flash programmer cannot program the boot sectors of either the host or slave side of the 8102ADS board because these sectors are protected by the BCSR6 and BCSR0 registers.

For the host side of the board, however, there is a workaround: You can unprotect the boot sectors of the host side by changing the values of registers BCSR6 and BCSR0. To do this, add these commands to your target initialization file:

```
writemem32 0x14500018 0x00000000
writemem32 0x14500000 0xFFFFFFFF
```

NOTE There is no workaround with which you can program the boot sectors of the slave side of the 8102ADS board.

ELF Dump Utility

The ELF (executable and linking format) file dump utility outputs the headers of absolute and linkable object files in a human-readable form.

The information produced by the utility depends on the type of ELF object file:

- Absolute (executable) object file

  The default output is the ELF header, all program headers, and all sections headers.

- Linkable (relocatable) object file

  The default output is the ELF header and all section headers.

Running the ELF Dump Utility

You can run the ELF dump utility from within the CodeWarrior IDE by selecting SC100 ELF Dump from the Post-linker listbox of the Target Settings panel.

You can also invoke the ELF dump utility from a command-line prompt using this command line:

```
sc100-elfdump [option ...] file ...
```

NOTE The ELF dump utility is in this directory: `installDir\StarCore_Support\compiler\bin`

Parameters

option

One or more of the command-line options listed in Table 14.2. 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 14.2 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)
`-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.
-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 14.2 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.)

ELF File Dump Output

Listing 14.1 shows the output of the ELF file dump utility. Note these things:

- The file name is `hello.eld`.
- The ELF header extends from line `e_ident` through line `e_shstrndx`.
- The program headers comprise lines `Segment 0`, `Segment 1` and their subordinate lines.
- The section headers comprise the remaining lines.

Listing 14.1 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 listbox of the Target Settings panel.

You can also invoke the ELF to LOD utility from a command-line prompt using this command line:

```
elflod [option ...] file ...
```

NOTE The ELF to LOD utility is in this directory: *installDir*\StarCore_Support\compiler\bin

Parameters

option

One or more of the command-line options listed in Table 14.3.

NOTE elflod utility command-line options are case-sensitive.

file

The filename of the ELF object file to use to produce a LOD file.

Table 14.3 elflod Utility—Command-Line Options

Option	Description
-d *file*	Redirects the utility's output to the specified file.

ELF To LOD Output

Listing 14.2 shows an example LOD file.

Listing 14.2 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 Motorola S-Record format files.

The S-Record format, which is a standard Motorola 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.

You can also invoke the ELF to S-Record utility from a command prompt using this command line:

```
elfsrec [option ...] file ...
```

NOTE The ELF to S-Record utility is in this directory: `installDir\StarCore_Support\compiler\bin`

Parameters

option

One or more of the command-line options listed in Table 14.4.

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 14.4 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 DSP software:

- `-l`
- `-d`
- `-o`

NOTE For CodeWarrior for StarCore DSPs, the elfsrec utility's default option is `-b`

Archiver Utility

The 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.

To invoke the archiver, use one of the command lines listed below. Table 14.5 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 The `sc100-ar` utility is in this directory:
`installDir\StarCore_Support\compiler\bin`

NOTE 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 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 archiver command-line options. The syntax rules for an argument file are listed below.

- Begin a comment line with the # character.
- Each line must end with the backslash (\) character.

Table 14.5 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.

Table 14.5 Archiver Utility—Command-Line Options (*continued*)

Option	Effect
`-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.
`-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. `sc100-dis` is included in the product for backward compatibility.

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>
```

NOTE The `disasmsc100` utility is in this directory:
`*installDir*\StarCore_Support\compiler\bin`

Table 14.6 lists and defines each option you can pass to the `disasmsc100` utility.

NOTE disasmsc100 utility command-line options are case-sensitive.

Table 14.6 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 endiannes (l) or big endianness (b).
`-k<file>`	Reads options from the specified configuration file.
`-q`	Suppresses banner display.

Table 14.6 disasmsc100 Disassembler—Command-Line Options (*continued*)

Option	Effect
-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 14.3 shows the output produced by the `disasmsc100` utility if you pass the `-z` (display statistics) command-line option.

Listing 14.3 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
-------------------------------------------------------
```

Name Utility

The name utility displays the symbolic information in each object file and library passed on the command line. If a file contains no symbolics, the utility reports this fact.

Running the Name Utility

Use this command line to invoke the name utility:

```
sc100-nm [-option ...] file ...
```

NOTE The `sc100-nm` utility is in this directory:
`installDir\StarCore_Support\compiler\bin`

Parameters

option

One or more of the options listed in Table 14.7.

NOTE Name utility command-line options are case-sensitive.

file

Name of the file to process.

Table 14.7 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.)

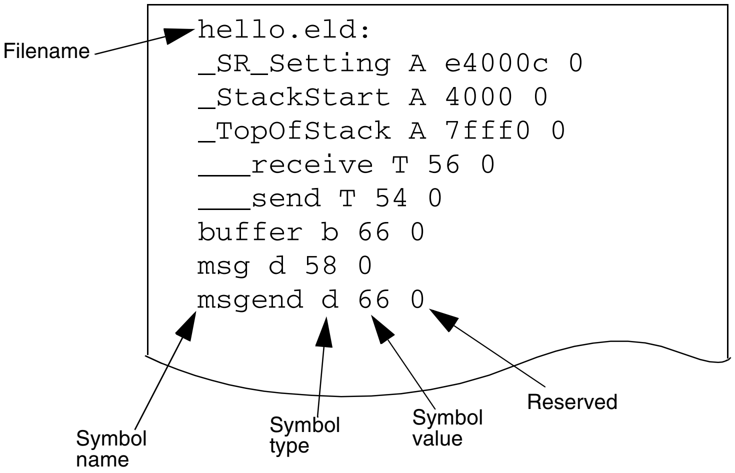
Table 14.7 Name Utility—Command-Line Options (*continued*)

Option	Effect
-V	Displays the version of the name utility.
-v	Sorts output by value, instead of by name.

Name Utility Output

Figure 14.2 shows the output generated by the name utility.

Figure 14.2 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 14.8 provides a key to the name utility's output. Note that uppercase letters indicate global symbols, while lowercase letters indicate local symbols.

Table 14.8 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

Table 14.8 Name Utility—Output Key (*continued*)

Character	Symbol Type
`D` or `d`	Data symbol
`R` or `r`	Read-only data symbol
`N`	Debug symbol
`?`	Unknown symbol type or binding

Size Utility

The size utility outputs the size (in bytes) of each section of each ELF object file passed on the command line. The default output lists totals for all `.text`, `.rodata`, `.data`, and `.bss` sections.

Running the Size Utility

Use this command line to invoke the size utility:

```
sc100-size [-option ...] file ...
```

NOTE The `sc100-size` utility is in this directory:
`installDir``\StarCore_Support\compiler\bin`

Parameters

`option`

One or more of the options listed in Table 14.9.

NOTE Size utility command-line options are case-sensitive.

`file`

Name of an ELF file.

Table 14.9 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.

Table 14.9 Size Utility—Command-Line Options (*continued*)

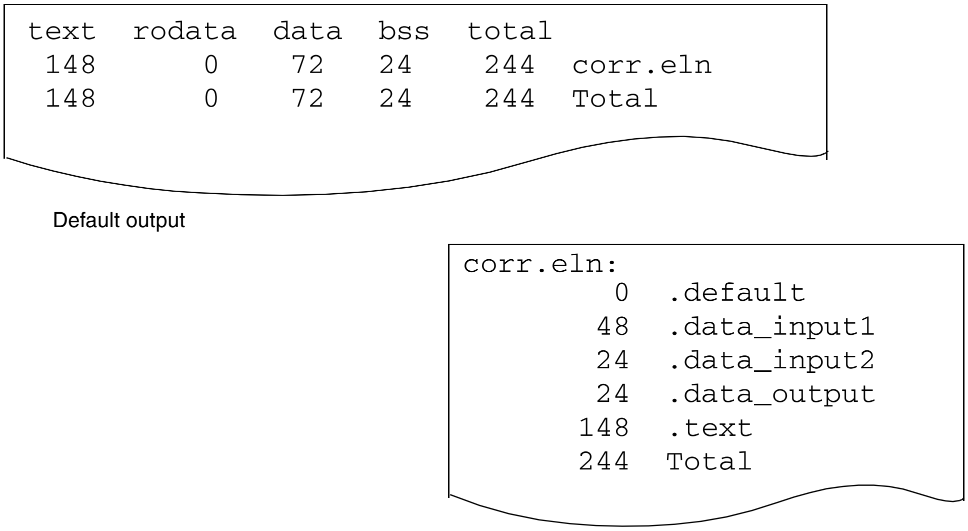
Option	Effect
-p	Outputs the size of all loadable segments (program view).
-V	Displays the version of the size utility.

Size Utility Output

Listing 14.1 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 14.3 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.

The sc100-stat 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 syntax for `sc100-stat` follows:

```
sc100-stat .eld_filename [section_name...] [-d]
```

NOTE The `sc100-stat` utility is in this directory:
`installDir\StarCore_Support\compiler\bin`

Table 14.10 lists and defines the options for the `sc100-stat` utility.

NOTE Statistics utility command-line options are case-sensitive.

Table 14.10 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.

15 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 15.1.)

Figure 15.1 Link Commander Window

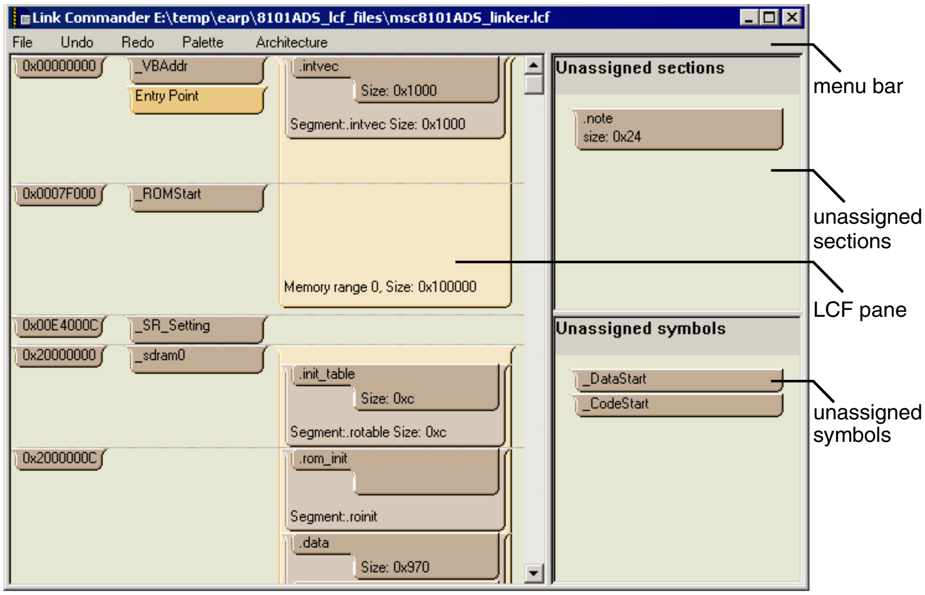


Table 15.1 lists and defines each option in the Link Commander menu bar.

Table 15.1 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

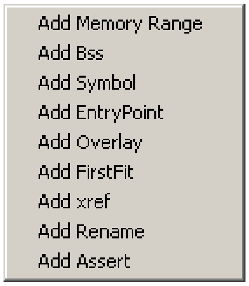
  This pane depicts the linker command file graphically.

  – 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 15.2.)

  – Select from this menu to add a memory range, a symbol, or any of several other LCF objects.

Figure 15.2 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.

16

C and Assembly Language Benchmarks

Your CodeWarrior™ software includes source code for common DSP benchmarks. Use these benchmarks to:

- Evaluate the performance of the Metrowerks™ 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

Windows® Version

The Windows versions of the C language benchmark source code files are here:

installDir\Examples\StarCore\Benchmark\c\

Solaris™ Version

The Solaris versions of the C language benchmark source code files are here:

installDir/CodeWarrior_ver_dir/
CodeWarrior_Examples/Benchmark/c

Table 16.1 describes the benchmarks in each directory.

Table 16.1 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:

   Windows versions:

   - *`installDir`*`\Examples\StarCore\Benchmark\c\efr\efr.mcp`
   - *`installDir`*`\Examples\StarCore\Benchmark\c\msample\msample.mcp`

   Solaris versions:

   - *`installDir`*`/CodeWarrior_ver_dir/CodeWarrior_Examples/Benchmark/c/efr/efr.mcp`
   - *`installDir`*`/CodeWarrior_ver_dir/CodeWarrior_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.

Windows

- *`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`

Solaris

- `*installDir*/CodeWarrior_ver_dir/CodeWarrior_Examples/c_asm_mix`
- `*installDir*/CodeWarrior_ver_dir/CodeWarrior_Examples/FileIO`
- `*installDir*/CodeWarrior_ver_dir/CodeWarrior_Examples/Sc140`
- `*installDir*/CodeWarrior_ver_dir/CodeWarrior_Examples/Command_Line_Script_Debug`

Assembly Language Benchmarks

Windows® Version

The Windows versions of the assembly language benchmark source code files are here:

installDir\Examples\StarCore\Benchmark\asm

Solaris™ Version

The Solaris versions of the assembly language benchmark source code files are here:

installDir/CodeWarrior_ver_dir/
CodeWarrior_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:

Windows®

- Absolute ASM
- Relocatable ASM

Solaris™

- 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 16.2 lists the assembly language benchmarks.

Table 16.2 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