



CodeWarrior® Targeting Dreamcast



Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes for the latest up-to-date information.

Revised: 990129 rw

Metrowerks CodeWarrior copyright ©1993–1999 by Metrowerks Inc. and its licensors. All rights reserved.

Documentation stored on the compact disk(s) may be printed by licensee for personal use. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from Metrowerks Inc.

Metrowerks, the Metrowerks logo, CodeWarrior, and Software at Work are registered trademarks of Metrowerks Inc. PowerPlant and PowerPlant Constructor are trademarks of Metrowerks Inc.

All other trademarks and registered trademarks are the property of their respective owners.

ALL SOFTWARE AND DOCUMENTATION ON THE COMPACT DISK(S) ARE SUBJECT TO THE LICENSE AGREEMENT IN THE CD BOOKLET.

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation 9801 Metric, Suite 100 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Ordering	Voice: (800) 377-5416 Fax: (512) 873-4901
International Ordering	Voice: +1 512 873-4724 Fax: +1 512 873 4901
World Wide Web	http://www.metrowerks.com
Registration information	mailto:register@metrowerks.com
Technical support	mailto:support@metrowerks.com
Sales, marketing, & licensing	mailto:sales@metrowerks.com
International sales, marketing, & licensing	intlsls@metrowerks.com
CompuServe	goto Metrowerks

Table of Contents

1 Introduction	7
Read the Release Notes!	7
CodeWarrior and Its Documentation	8
What's in This Manual	9
Where To Go from Here	10
2 Getting Started	13
System Requirements	13
Installing CodeWarrior for Dreamcast	14
Installing the CodeWarrior for Dreamcast Software	14
Installing the Dreamcast Runtime Library	15
Making Sure Your Dreamcast Development System Works.	15
3 The Dreamcast Tools	19
Introduction to the Dreamcast Tools.	19
CodeWarrior IDE.	20
CodeWarrior Compiler for Dreamcast.	20
CodeWarrior Assembler for Dreamcast	21
CodeWarrior Linker for Dreamcast	21
CodeWarrior Debugger for Dreamcast	21
Codescape Debugger for Dreamcast	21
The Development Process with CodeWarrior	22
4 Creating Applications	25
Creating an Application	25
5 Creating Static Libraries	35
About Static Libraries	35
Creating a Static Library.	35
6 Converting SH Projects	37
Steps for Converting SH Projects	37
7 Debugging For Dreamcast	43
Debugging with CodeWarrior	43

Table of Contents

Using <code>printf()</code>	44
Debugging Static Libraries	44
8 Debugging With Codescape	45
Debugging with the Codescape debugger	45
Using <code>printf()</code>	47
9 Target Settings for Dreamcast	49
Target Settings Overview	49
Settings Panels for Dreamcast	51
Target Settings	51
SH Target	54
SH Processor	57
Global Optimizations	58
Section Mappings	60
SH Linker	61
Debugger Settings	63
10 C and C++ for Dreamcast	65
Number Formats for Dreamcast	66
Dreamcast Integer Formats	67
Dreamcast Floating-Point Formats	67
Calling Conventions for Dreamcast	68
Variable Allocation for Dreamcast.	68
Optimizing Code for Dreamcast	68
Pragmas for Dreamcast	72
Linker Issues for Dreamcast	72
Linker Command File.	73
Deadstripping Unused Code and Data	80
Link Order	80
C++ issues for Dreamcast	81
11 Inline Assembler and Intrinsics for Dreamcast	83
Working with Inline Assembly	83
Inline Assembler Syntax.	84
Using Labels.	87

Table of Contents

Using Comments.	87
Using Registers	87
Assembler Directives	89
Intrinsic Functions	90
List of Intrinsic Functions	91
Mnemonics for Inline Assembly	93
Special Instructions for Inline Assembly	94
Complete List of Inline Assembly Mnemonics	95
12 Libraries and Runtime Code for Dreamcast	107
Runtime Libraries for Dreamcast	107
Allocating Memory and Heaps for Dreamcast	108
13 Troubleshooting for Dreamcast	109
Hardware Communications	109
Compiler Problems	110
Debugger Problems	110
Index	111

Table of Contents



Introduction

This manual describes how to use CodeWarrior to develop code targeted at the Dreamcast platform. This includes stand-alone application programs and static libraries.

The manual also shows how to set Dreamcast project options, and describes CodeWarrior's Dreamcast specific run-time libraries.

The introduction includes the following sections:

- [Read the Release Notes!](#)—where to go for critical, last-second details
- [CodeWarrior and Its Documentation](#)—a general description of the CodeWarrior architecture and documentation
- [What's in This Manual](#)—a description of the contents of this manual
- [Where To Go from Here](#)—recommendations for further reading

Read the Release Notes!

Before you use the CodeWarrior IDE or a particular tool, you should read the release notes. They contain important last-minute information about new features, bug fixes, and incompatibilities that *may not be included in the documentation*.

The release notes folder is always included as part of a standard CodeWarrior installation. The release notes folder is also located at the top level of the CodeWarrior CD.

CodeWarrior and Its Documentation

CodeWarrior is a multi-host, multi-language, multi-target development environment. What does that mean?

Multiple hosts CodeWarrior runs on several different operating systems including Windows, Solaris, and Mac OS. The features, human interface, and operation of CodeWarrior is very similar on all hosts.

Multiple languages You can use CodeWarrior to program in several languages, including C/C++, Pascal, and Java. Third-party compilers provide support for other languages such as Fortran. Which languages are available to you depend upon the target for which you are developing software.

Multiple targets You can use CodeWarrior to write software for several different chips or operating systems. CodeWarrior products support programming for game consoles, embedded processors, real-time operating systems, the Java Virtual Machine, and desktop operating systems such as Windows and Mac OS.

Most features of CodeWarrior apply regardless of your preferred host, language, or target. General features of CodeWarrior are described in other manuals, such as the *IDE User Guide* and *Debugger User Guide*.

However, each target has its own unique features. This manual describes those unique features.

For a complete understanding of CodeWarrior, you must refer to both the general documentation and the documentation that is specific to your particular target, such as this manual.

The documentation is organized so that various chapters in this manual are extensions of particular generic manuals, as shown in [Table 1.1](#). For a complete discussion of a particular subject, you may need to look in both the generic manual and the corresponding chapter in this Targeting manual.

Table 1.1 CodeWarrior documentation organization

This chapter...	Extends...
Creating Applications Creating Static Libraries	<i>Core Tutorials</i>
The Dreamcast Tools Target Settings for Dreamcast	<i>IDE User Guide</i>
“Debugging For Dreamcast”	<i>Debugger User Guide</i>
C and C++ for Dreamcast	<i>C Compilers Reference</i>

For example, to completely understand the C/C++ compiler, you need to know information in the *C Compilers Reference* (which covers the C/C++ front-end compiler) and the information in the C and C++ for Dreamcast chapter in this manual, which covers the back-end compiler that generates your Dreamcast specific code.

What's in This Manual

[Table 1.2](#) lists every chapter in this manual, and describes the information contained in each. However, this manual only contains information specific to Dreamcast software development. See [“CodeWarrior and Its Documentation” on page 8](#) for a discussion of how these chapters relate to other CodeWarrior documentation.

Table 1.2 Contents of chapters

Chapter	Description
Introduction	this chapter
Installing CodeWarrior for Dreamcast	how to install CodeWarrior for Dreamcast
The Dreamcast Tools	describes the tools for Dreamcast
Creating Applications	how to build applications for Dreamcast

Chapter	Description
Creating Static Libraries	how to build libraries for Dreamcast
Converting SH Projects	how to convert existing projects into CodeWarrior projects
Debugging For Dreamcast	how to debug your Dreamcast applications with CodeWarrior
Debugging With Codescape	how to interface CodeWarrior with the external Codescape debugger
Target Settings for Dreamcast	how to control the compiler and linker for Dreamcast
C and C++ for Dreamcast	details of the back-end C/C++ compiler for Dreamcast development.
Inline Assembler and Intrinsics for Dreamcast	details support for inline assembly and intrinsic functions
Libraries and Runtime Code for Dreamcast	libraries provided with CodeWarrior for Dreamcast
Troubleshooting for Dreamcast	troubleshooting information specific to Dreamcast development

Where To Go from Here

The manuals mentioned in this section are all on the CodeWarrior CD.

For everyone:

- For complete information about the CodeWarrior integrated development environment, see the *IDE User Guide*
- For information specific to the C/C++ front-end compiler, see the *C Compilers Reference*.

For reference information on Dreamcast programming:

Please contact the provider of your Dreamcast development hardware for programming manuals specific to Dreamcast and its SH processor.

Introduction

Where To Go from Here



Getting Started

This chapter gives you the information you need to install CodeWarrior and begin programming the Dreamcast game console.

This chapter includes the following topics:

- [System Requirements](#) — hardware and software requirements
- [Installing CodeWarrior for Dreamcast](#) — how to install the various tools

System Requirements

- A Pentium-class or higher computer. For best performance, we recommend a Pentium II-class processor.
- Windows 95/98, or Windows NT 4.0 operating system
- 500MB of hard disk space.
- A minimum of 32MB RAM. 64MB RAM is preferred.
- A CD-ROM drive to install CodeWarrior software, documentation, and examples.

In addition to the requirements above, you also need:

- HKT-01 development hardware, revision 5-24. The serial number on the bottom of your HKT-01 contains the revision code. If the serial number does not begin "S524. . .", contact Sega for new hardware.
- Version 1.30j of the SDK Shinobi libraries.

Installing CodeWarrior for Dreamcast

Programming for the Dreamcast game console requires installing and configuring both the CodeWarrior development tools and the Dreamcast development hardware.

Installing and configuring the software is not immediately obvious, so this chapter is essential reading. At this point, you should have the Dreamcast development hardware connected to your PC.

Before you can begin using the CodeWarrior tools, you must

1. Install CodeWarrior

For complete details, see [“Installing the CodeWarrior for Dreamcast Software” on page 14.](#)

2. Install the Dreamcast libraries

For complete details, see [“Installing the Dreamcast Runtime Library” on page 15.](#)

3. Test your system.

Before you begin programming, see [“Making Sure Your Dreamcast Development System Works” on page 15.](#)

Installing the CodeWarrior for Dreamcast Software

Your first step towards developing software for your target is to install the CodeWarrior tools.

Double-click the setup.exe file from the CD, and follow the instructions that the installation wizard provides. If you have any questions regarding the installer, read the instructions built into the CodeWarrior Installer for further information.

NOTE: If you are using a dual-boot system with Windows 95/98 and Windows NT installed, install the tools on Windows 95/98 first. After the installation has finished, shutdown, reboot into Windows NT, and install the CodeWarrior tools in the same directory selected in the Windows 95/98 installation.

This completes the CodeWarrior for Dreamcast tools installation.

Installing the Dreamcast Runtime Library

The Shinobi libraries are used in almost every Dreamcast project you develop.

In this beta release, we have included CodeWarrior-compatible Shinobi libraries in the folder named "Dreamcast Support". They are copied over as part of the installation procedure.

Making Sure Your Dreamcast Development System Works

After installing the software, you should make sure it works. To do this, compile and execute the teapot demo that is included in the CodeWarrior example files.

- 1. Launch the CodeWarrior IDE**

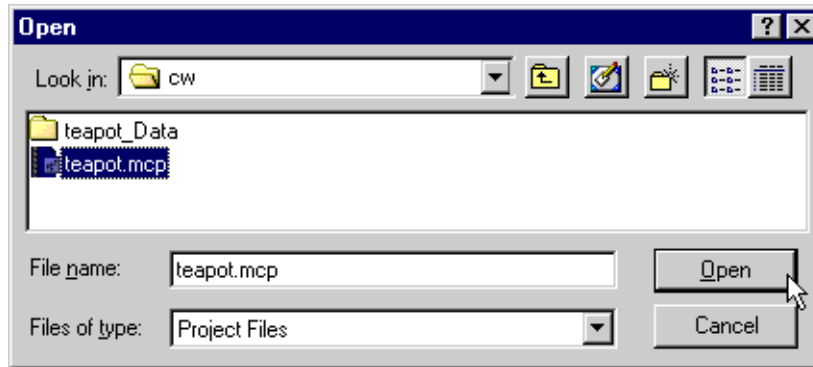
Locate the icon for the CodeWarrior IDE, and launch the application.

- 2. Open the project.**

From the **File** menu, choose the **Open** item. The dialog box in [Figure 2.1](#) appears.

Locate the project `CodeWarrior Examples/Dreamcast/Sdk/Teapot/cw/teapot.mcp`.

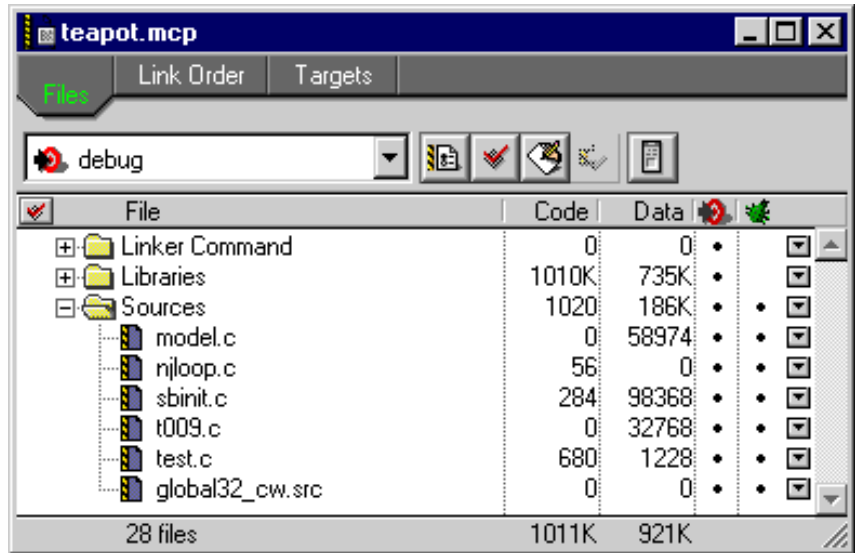
Figure 2.1 The 'open' dialog box



Select the project file and open it. The CodeWarrior project window will appear, as shown in [Figure 2.2](#).

The project window is the central location from which you control development. This is where you can add or remove source files, add libraries of code, compile your code, generate debugging information, and much more. For full information on the CodeWarrior IDE and project manager, you should see the *CodeWarrior IDE User Guide*.

Figure 2.2 The 'project' window



3. Build the project.

Choose the **Make** command from the **Project** menu to build the project. CodeWarrior will compile and link your project into a program file called `teapot_debug.elf`.

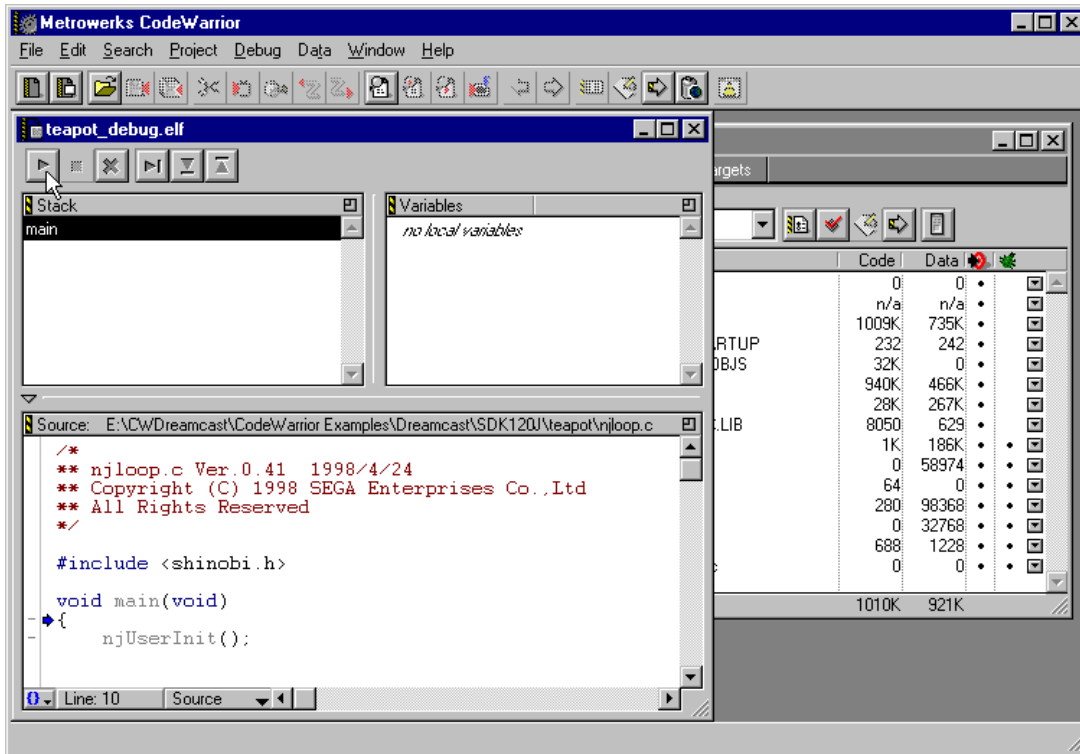
4. Debug the project.

Click the **Debug** command from the **Project** menu. After CodeWarrior uploads the compiled teapot program to your HKT-01 hardware, the program window will appear as shown in [Figure 2.3](#).

Getting Started

Installing CodeWarrior for Dreamcast

Figure 2.3 The 'program' window



5. Run the project.

Click the **Run** command from the **Project** menu. If your software and hardware are set up correctly, the teapot demo will run, as shown in [Figure 2.4](#).

Figure 2.4 The teapot demo





The Dreamcast Tools

This chapter briefly explains the CodeWarrior for Dreamcast development environment.

For new CodeWarrior users, this chapter provides a brief overview of the CodeWarrior development environment, as well as a description of the development process in CodeWarrior as compared to a command-line environment.

The topics in the chapter are:

- [Introduction to the Dreamcast Tools](#)
- [The Development Process with CodeWarrior](#)

Introduction to the Dreamcast Tools

Programming with CodeWarrior for Dreamcast is much like programming for any other CodeWarrior target. If you have never used CodeWarrior before, the tools you will need to become familiar with are:

- [CodeWarrior IDE](#)
- [CodeWarrior Compiler for Dreamcast](#)
- [CodeWarrior Assembler for Dreamcast](#)
- [CodeWarrior Linker for Dreamcast](#)
- [Codescape Debugger for Dreamcast](#)

If you are an experienced CodeWarrior user, this is the same IDE and debugger you've been using all along.

CodeWarrior IDE

The CodeWarrior IDE is the application that allows you to write your executable. It controls the project manager, the source code, editor, the class browser, and the compilers and linkers.

The CodeWarrior project manager may be new to those more familiar with command-line development tools. All files related to your project are organized in the project manager. This allows you to see your project at a glance, and eases the organization of and navigation between your source code files.

For more information about how the CodeWarrior IDE compares to a command-line environment, see [“The Development Process with CodeWarrior” on page 22](#). That short section discusses how various parts of the IDE implement the classic features of a makefile-based command-line development system.

The CodeWarrior IDE has an extensible architecture that uses plug-in compilers and linkers to target various operating systems and microprocessors. The CodeWarrior for Dreamcast package includes a C/C++ compiler for the Hitachi SH4 processor. Other CodeWarrior packages include C and C++ compilers for x86 and 68000 processors, among other platforms.

For more information about the CodeWarrior IDE, you should read the *CodeWarrior IDE User Guide*.

CodeWarrior Compiler for Dreamcast

The CodeWarrior compiler for Dreamcast is an ANSI compliant C/C++ compiler. This compiler is based on the same compiler architecture that is used in all of the CodeWarrior C/C++ compilers. When used with the CodeWarrior linker for Dreamcast, you can generate Dreamcast applications and libraries.

For more information on the Compiler Settings, see [“Target Settings for Dreamcast” on page 49](#). For more information about the CodeWarrior C/C++ language implementation, you should read the *C Compiler Guide*.

CodeWarrior Assembler for Dreamcast

The CodeWarrior assembler for Dreamcast allows you to include assembly source code as part of your project.

For more information about Dreamcast assembly programming, you should read Hitachi's *SH4 Assembler Guide*.

CodeWarrior Linker for Dreamcast

The CodeWarrior linker for Dreamcast links object code into an ELF format executable. It also generates DWARF format debugging information. This linker creates code using absolute addressing.

For more information about the linker settings, see [“Target Settings for Dreamcast” on page 49](#).

CodeWarrior Debugger for Dreamcast

CodeWarrior's debugger allows you to see what is happening inside your application as it runs.

You use the debugger to find problems in your program's execution. The debugger can execute your program one statement at a time and suspend execution when you reach a specified point. When the debugger stops a program, you can view the chain of function calls, examine and change the values of variables, and inspect the content of the processor's registers.

For general information about debugging, including all of its features and its visual interface, you should read the *Debugger User Guide*. Specific information pertaining to debugging the Dreamcast can be found in [“Debugging For Dreamcast” on page 43](#).

Codescape Debugger for Dreamcast

The Codescape debugger from Cross Products is a stand-alone application separate from the CodeWarrior IDE.

Specific information about interfacing the Codescape. For general information about the Codescape debugger, including all of its features and its visual interface, you should read the *Codescape for Set 5 User Guide*.

The Development Process with CodeWarrior

While working with CodeWarrior, you will still proceed through the development stages familiar to all programmers: write code, compile, link, and debug. For complete information on performing software development tasks like editing, compiling, and linking, refer to the *CodeWarrior IDE User Guide*. For debugging using Codescape, see the *Codescape for Set 5 User Guide*.

The difference between CodeWarrior and traditional command line environments is in how the software (in this case the IDE) helps you manage your work more effectively. If you are unfamiliar with an integrated environment in general, or with CodeWarrior in particular, you may find the topics in this section helpful. Each topic discusses how one component of the CodeWarrior tools relates to a traditional command line environment.

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

- [Makefiles](#)—the IDE uses a project to control source file dependencies and settings for compilers and linkers
- [Editing](#)—an overview of source code editing from the IDE
- [Compiling](#)—how the IDE performs compile operations
- [Linking](#)—how the linker performs linking operations
- [Debugging](#)—how to debug a program

Makefiles

The CodeWarrior IDE *project* is analogous to a makefile. Because you can have multiple builds in the same project, in fact the project is analogous to a collection of makefiles. For example, you can have one project that has both a debug version and a release version of your code. You can build one or the other, or both as you wish. In

CodeWarrior, these different builds within a single project are called “targets”.

The IDE uses the project manager window to list all the files in the project. Among the kinds of files in a project are source code files and libraries.

You can add or remove files easily. You can assign files to one or more different targets within the project, so files common to multiple targets can be managed simply.

The IDE manages all the interdependencies between files automatically, and tracks which files have been changed since the last build. When you rebuild, only those files that have changed are recompiled.

The IDE also stores the settings for compiler and linker options in the project. You can modify these settings using the IDE, or use `#pragma` statements in your code.

Editing

The CodeWarrior IDE has an integral text editor to edit source code. It handles text files in MS-DOS/Windows, UNIX, and Mac OS formats.

To edit a source code file, or any other editable file that is in a project, just double-click the file's name in the project window to open the file.

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

Compiling

To compile a source code file, it must be among the files that are part of the current target. If it is, you simply select it in the project window and choose **Compile** from the **Project** menu.

The Dreamcast Tools

The Development Process with CodeWarrior

To compile all the files in the current target that have been modified since they were last compiled, choose **Bring Up To Date** in the **Project** menu.

In UNIX and other command-line environments, object code compiled from a source code file is stored in a binary file (a “.o” or “.obj” file). The CodeWarrior IDE stores and manages object files transparently.

Linking

Linking object code into a final binary is easy: use the **Make** command in the **Project** menu. The **Make** command brings the active project up to date, then links the resulting object code into a final output file.

You control the linker through the IDE. There is no need to specify a list of object files. The project manager tracks all the object files automatically.

You can use the project manager to specify link order as well.

Debugging

To debug a project, select **Debug** from the **Project** menu.



Creating Applications

A Dreamcast application is a stand-alone, executable program. You compiled and ran one such Dreamcast application when you verified your CodeWarrior installation.

In this chapter, we will take this one step further, and show you how to create your own application.

This chapter includes the following topic:

- [Creating an Application](#)

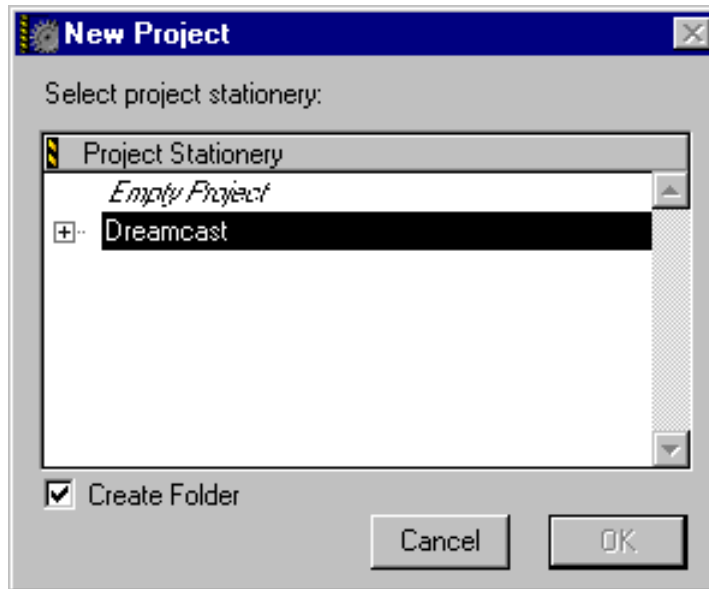
Creating an Application

To create a Dreamcast application, perform the following steps:

1. **Display the New Project dialog box.**

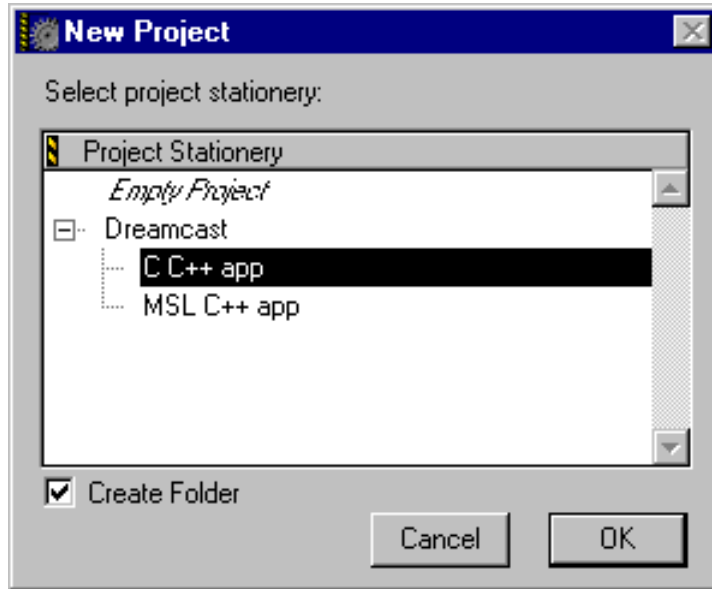
Choose the **New Project** command from the File menu. CodeWarrior will display the **New Project** dialog box as seen in [Figure 4.1](#), with instructions to select your project stationery.

Figure 4.1 New Project window



2. **Display the available Dreamcast project stationery.**
Click the hierarchical control to the left of the **Dreamcast** listing to see the project stationery available to you. [Figure 4.2](#) shows the expanded stationery list.

Figure 4.2 Selecting project stationery

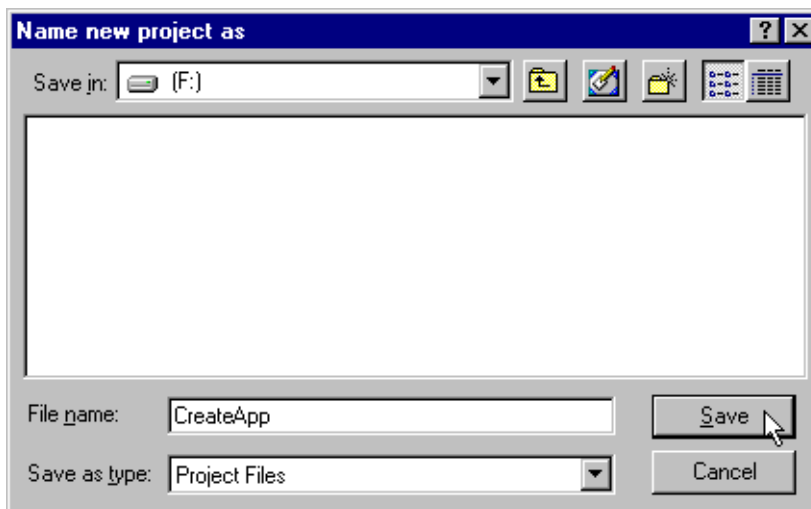


3. **Select your project stationery.**

Click on the line containing the Dreamcast stationery you want, then click OK. You will see the Name new project dialog box as shown in [Figure 4.3](#).

NOTE: To create a new project without using project stationery, select **Empty Project** in the **New Project** window. This lets you create a project from scratch, but it is not recommended because of the complexities of including the correct libraries and specifying the correct settings.

Figure 4.3 Name new project dialog window



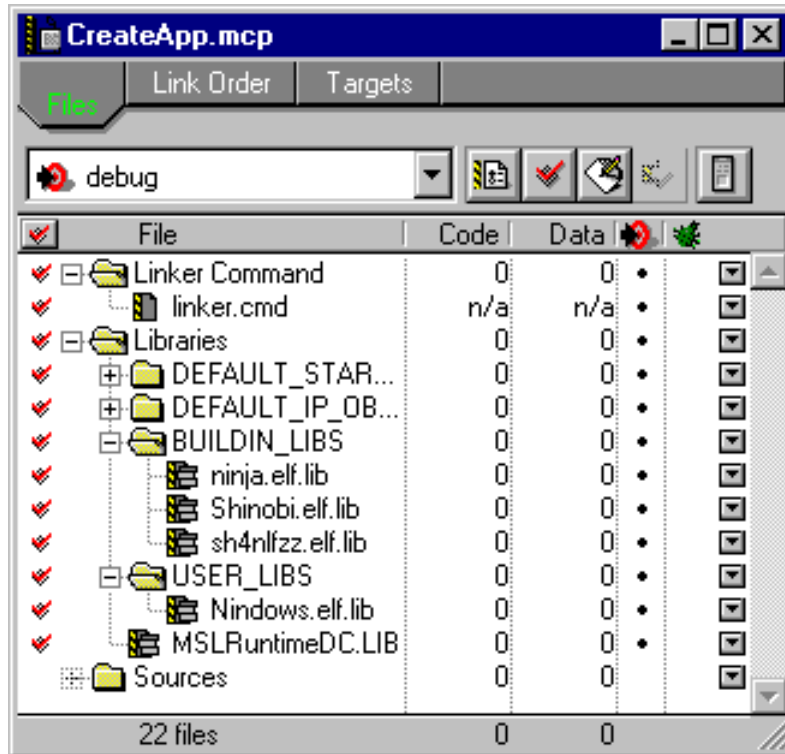
TIP: If you name your project without the `.mcp` extension, CodeWarrior adds `.mcp` to the project name for you. Don't add the `.mcp` extension to your project name if the Create Folder option in the New Project dialog box is checked. If you do, `.mcp` is added to the end of the folder name.

4. Complete the Name new project dialog.

Navigate to the directory in which you want to place your new project and type the project's name in the box labeled **File name**. When you click the **Save** button, CodeWarrior will create a new project file in the designated directory, with the conventional extension `.mcp`.

The project window you see on your screen contains the Shinobi libraries and an empty place for your program's source files. It should resemble the window shown in [Figure 4.4](#)

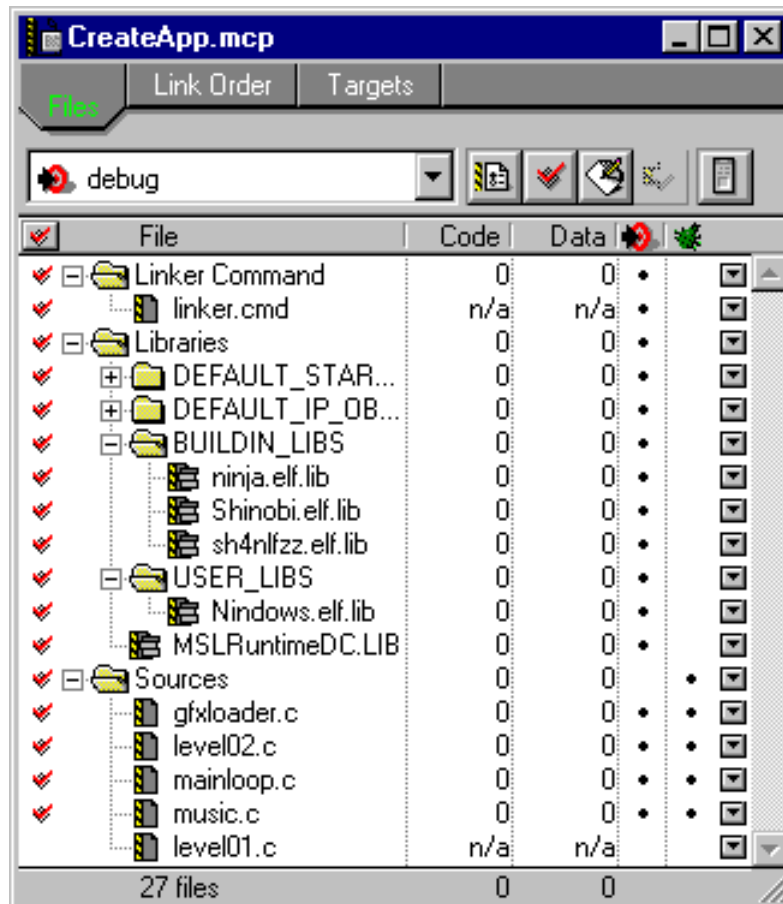
Figure 4.4 Project window



5. **Modify the contents of the new project.**

You will want to add your own source files to your new project. [Figure 4.5](#) shows the project window with some source files added.

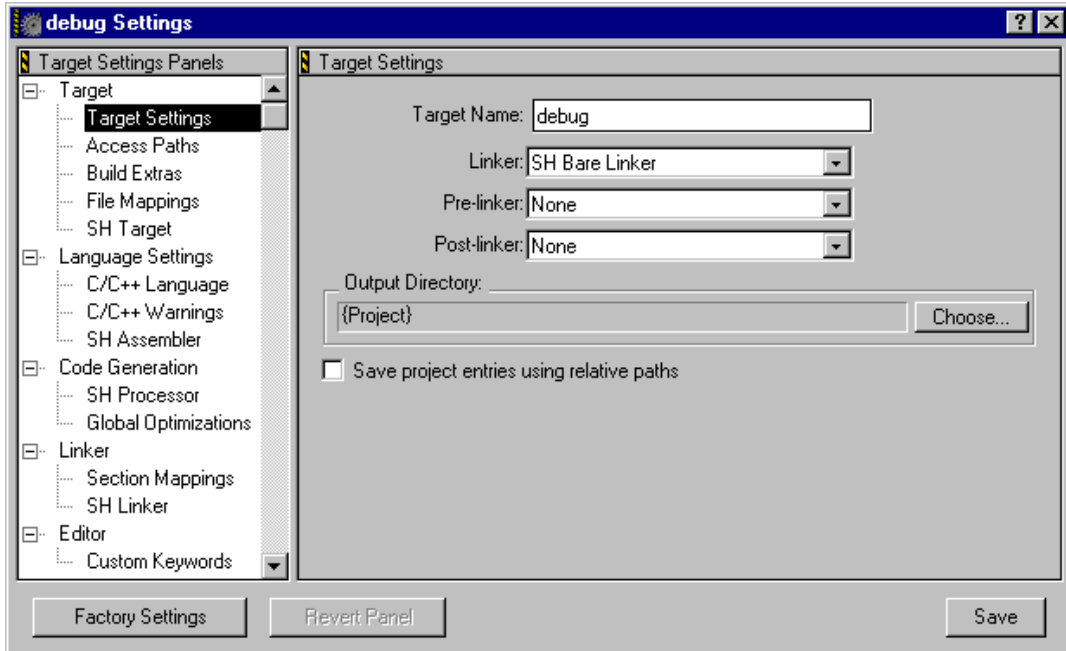
Figure 4.5 Project window with modifications



6. **Open the Target Settings window.**

Make sure your project window is active (front-most) on the screen, then choose the **Settings** command from the **Edit** menu. (The command actually appears on the menu as **Target Settings**, where **Target** is the name of the project's currently selected target. In the project shown in [Figure 4.5](#), for example, the name of the command would be **debug Settings**).

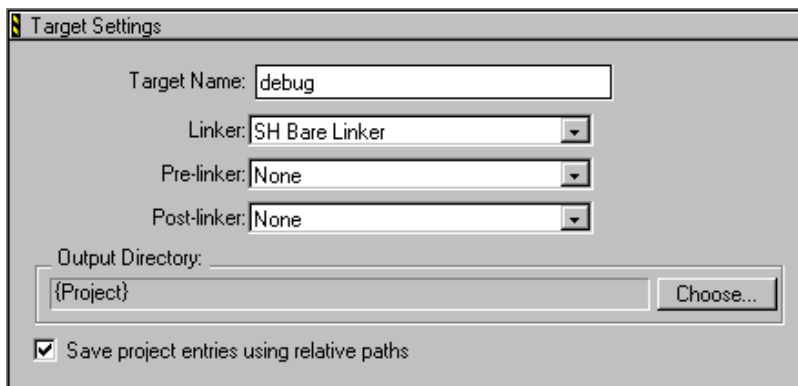
Figure 4.6 Target settings dialog box



CodeWarrior displays the Target Settings dialog box in which you can specify various optional settings for your project. This dialog box is shown in [Figure 4.6](#).

For Dreamcast projects, you must specify settings for the target platform, the project type, the compiler, and the linker. There are other, optional settings that you can specify as well.

Figure 4.7 Target Settings panel



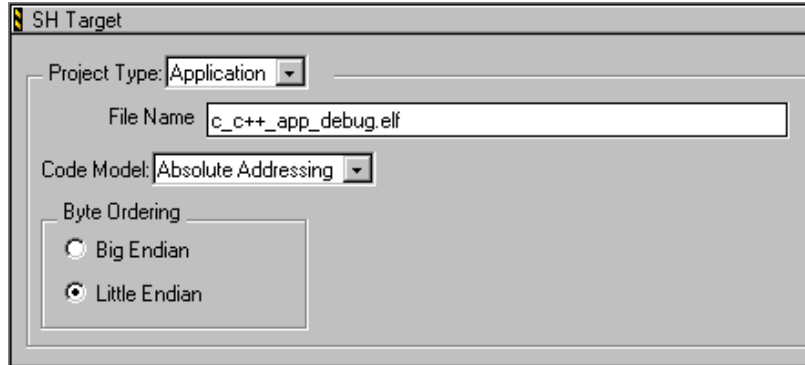
7. Specify target settings.

A list of settings panels are displayed to the left of the Target Settings dialog box. Select **Target Settings**; the window will display the **Target Settings** panel for the project's currently selected target, as shown in [Figure 4.7](#). The **Linker** setting is preset to **SH Linker** by the project stationery you selected, but you can edit the target's name or change other settings if you wish.

8. Set the project type.

Click **SH Target** in the panel list to display the settings panel shown in [Figure 4.8](#). Again, the project type and other default settings are preset for you by the project stationery. For an application project, you should leave the project type set to **Application**, but you can modify the output file name and other settings if you wish.

Figure 4.8 SH Target settings for application projects



4. Specify additional settings.

You can continue to display other project settings panels and specify any settings you wish. For more information on the various panels and settings available, see [“Target Settings for Dreamcast” on page 49](#) as well as the relevant sections of the *IDE User Guide*, and the *C Compilers Reference*.

When you’re finished specifying project settings, close the project settings window

5. Build your project.

After your project is created and its contents and all necessary settings are specified, you’re ready to compile and debug your code. The **Make** command on the Project menu compiles and links your project. If successful the resulting output file is stored in your project folder under the name you specified in the **SH Target** settings panel.

For more information on compiling and linking, see the *IDE User Guide*.

6. Debug your application.

Once you have successfully built your project, you can launch the debugger to debug and run your code.

Building a .bin file from Metrowerks.

- 1) From the CodeWarrior environment, generate a .elf file (example: debug.elf).
- 2) Launch the standard Dreamcast development DOS shell.
- 3) Cd (change directory) into the directory that contains the debug.elf...
- 4) and type:

```
elf2bin -s 8C010000 debug.elf
```

- 5) Resulting file will be a debug.bin



Creating Static Libraries

This chapter describes the role of static libraries in Dreamcast projects and how to create them.

Topics in this chapter are:

- [About Static Libraries](#)
- [Creating a Static Library](#)

See also [“Creating Applications” on page 25](#) for information on creating executable applications. For more information on projects in general, see the *IDE User Guide*.

About Static Libraries

A *static library* is a collection of functions and data that can be incorporated into an application program (or another library). You can use predefined libraries supplied with CodeWarrior, and you can create your own custom-designed libraries for use in your own projects.

Creating a Static Library

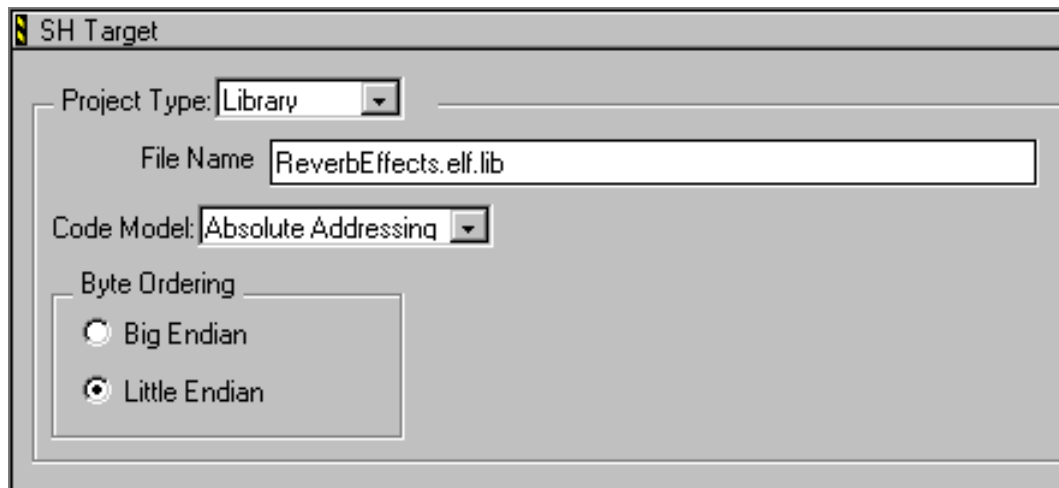
The steps for creating a static library are essentially the same as those for creating a stand-alone application, but with the following exceptions:

The **Project Type** in the **SH Target** settings panel shown in [Figure 5.1](#) must be set to **Library** instead of **Application**.

Creating Static Libraries

Creating a Static Library

Figure 5.1 SH Target panel



- You may invent your own naming convention, or you may use ours. Our naming convention is to use the file name extension `.elf.lib` for libraries and `.elf` for executables.
- After successfully building your static library, you incorporate it into another application by adding it to the project window before building the application.
- You cannot debug a static library by itself, but you can debug it as part of the application in which it is included.

See [“Creating an Application” on page 25](#) for step-by-step instructions on creating an application project. For details on the various project settings and panels available, see [“Target Settings for Dreamcast” on page 49](#) as well as the relevant sections of the *IDE User Guide* and the *C Compilers Reference*.



Converting SH Projects

This chapter shows you how to make CodeWarrior projects out of existing, makefile-based SH projects.

The topic covered in this chapter is:

- [Steps for Converting SH Projects](#)

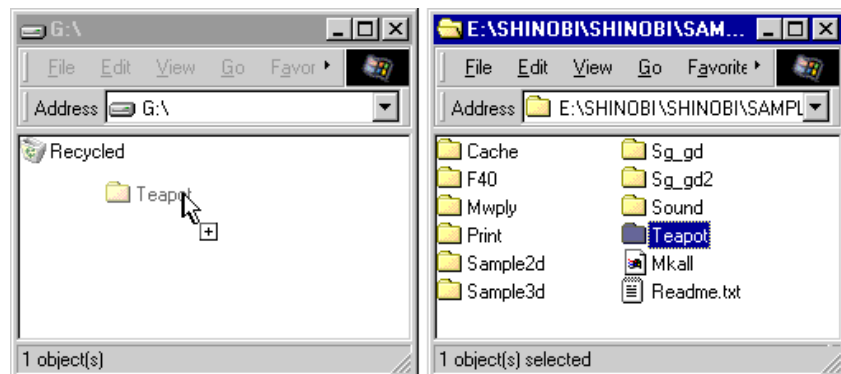
Steps for Converting SH Projects

In the steps that follow, we will convert the SDK Teapot demo into a CodeWarrior project we can compile, link, and debug.

1. **Copy the teapot sample to its own folder.**

Copy all the teapot files to a new folder. In our example shown in [Figure 6.1](#), our new teapot folder is on G:\.

Figure 6.1 Copying teapot files to a new folder



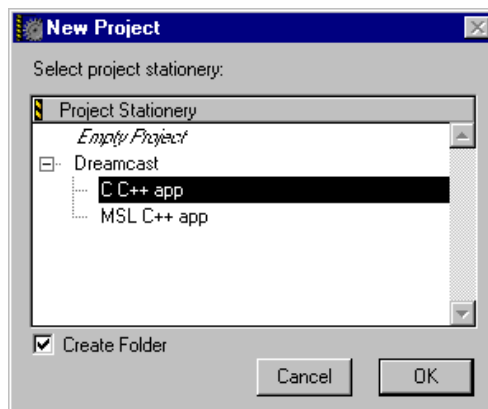
Converting SH Projects

Steps for Converting SH Projects

2. Create a new project.

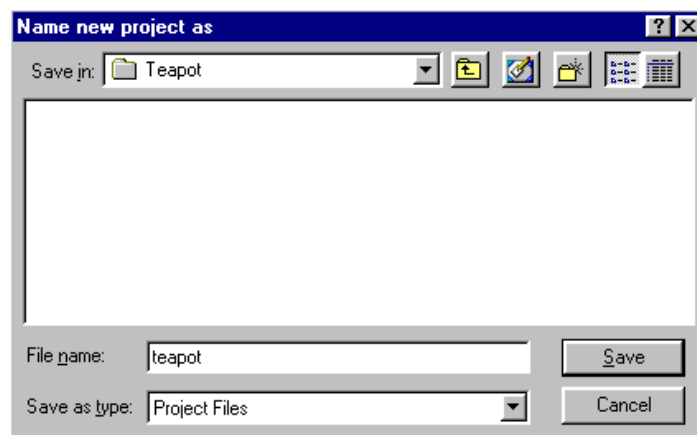
In CodeWarrior, choose **New Project** from the **File** menu. From the New Project window, select the Dreamcast **C app (no source)** stationery as shown in [Figure 6.2](#), and click **OK**.

Figure 6.2 Select the Dreamcast C app (no source) stationery



Please note that we do not check the **Create Folder** checkbox. We already have a folder for our new CodeWarrior project—the copied teapot folder. As in [Figure 6.3](#), save your new project in the teapot folder, with the file name teapot.

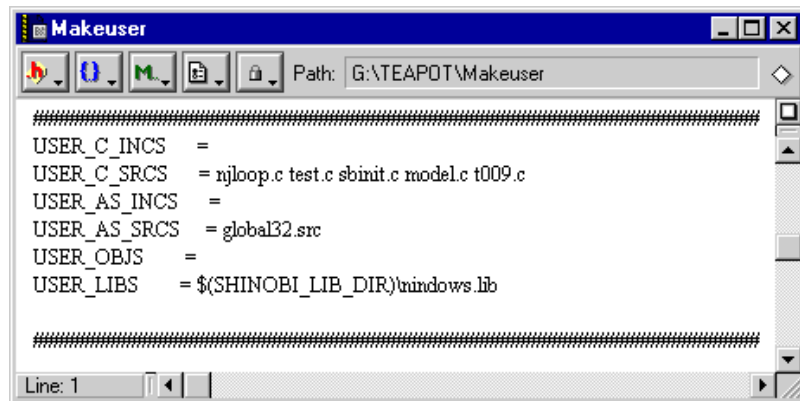
Figure 6.3 Start the new project in the teapot folder



3. **Add the source files from the makefile.**

The Makeuser file contains the names of the source files we want to add to our project. Open the Makeuser file that is in the teapot folder.

Figure 6.4 Finding source files in Makeuser



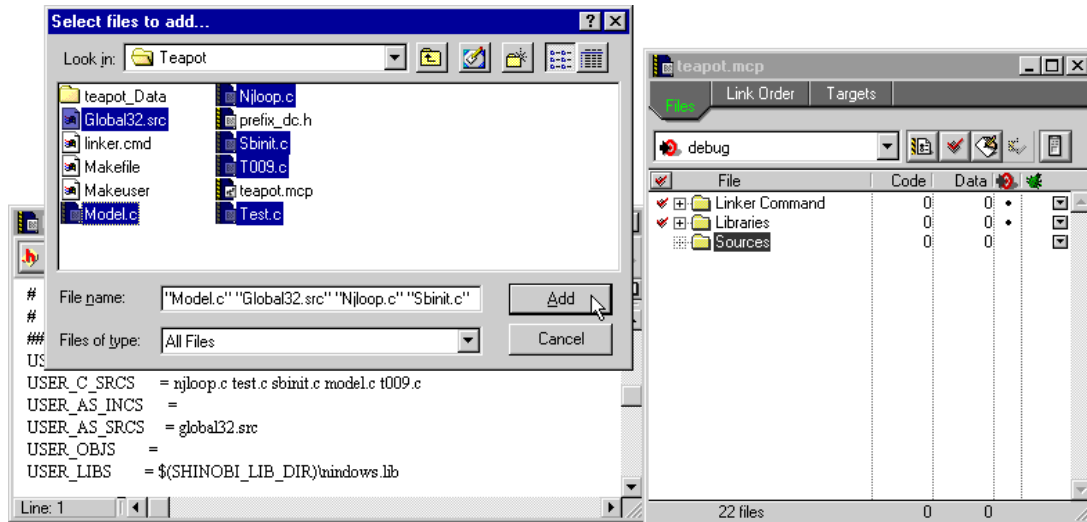
The files listed in [Figure 6.4](#) need to be added to our project. Placing them into our sources group will help keep our project organized.

Highlight the Sources group folder in the project window. From the **Project** menu, select **Add Files...** This takes you to the file selection dialog shown in [Figure 6.5](#). From here, you can select the source files from the teapot folder and add them to the project. The files you add are automatically placed at the bottom of the link order.

Converting SH Projects

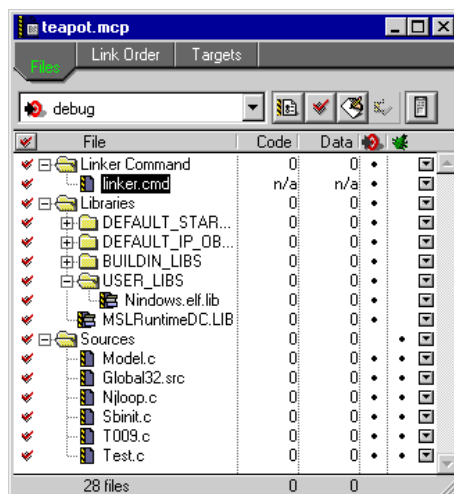
Steps for Converting SH Projects

Figure 6.5 Adding source files to the project window



Please note that you do not have to add `nindows.lib`. The CodeWarrior version, `nindows.elf.lib`, was included as part of the stationery. It is located inside the Libraries\USER_LIBS group. After adding the sources, your project window will resemble [Figure 6.6](#).

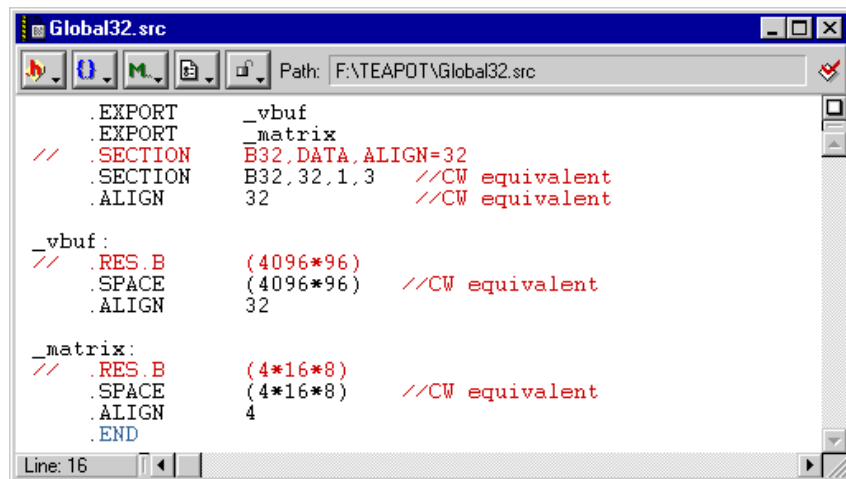
Figure 6.6 All files have been added



4. Convert assembler files.

Before teapot will compile on CodeWarrior, we must make a few changes to the assembler source file, `global32.src`, shown in [Figure 6.7](#). We must replace the Hitachi assembler directives with CodeWarrior equivalents.

Figure 6.7 Convert Hitachi assembler to CodeWarrior assembler



Hitachi's `.SECTION` directive specifies the B32 section as a bss section aligned on 32 bytes. The CodeWarrior equivalent of this is:

```
SECTION B32, 32, 1, 3  
.ALIGN 32
```

Replace the Hitachi `.SECTION` directive with the CodeWarrior directive.

NOTE: For a complete list of ELF section flags, see the "Using Directives" chapter of the *SH Assembler Reference*.

In CodeWarrior, we use `.SPACE` instead of `.RES.B`. Replace all instances of `.RES.B` with `.SPACE`.

Converting SH Projects

Steps for Converting SH Projects

5. The project has been converted.

You have successfully converted the teapot sample into a CodeWarrior project. You may compile and debug this project as if it were any other CodeWarrior project.



Debugging For Dreamcast

This chapter discusses how to use CodeWarrior to debug Dreamcast code. It covers those aspects of debugging that are specific to the Dreamcast platform or are different from the processes described in the *IDE User Guide* and the *Debugger User Guide*.

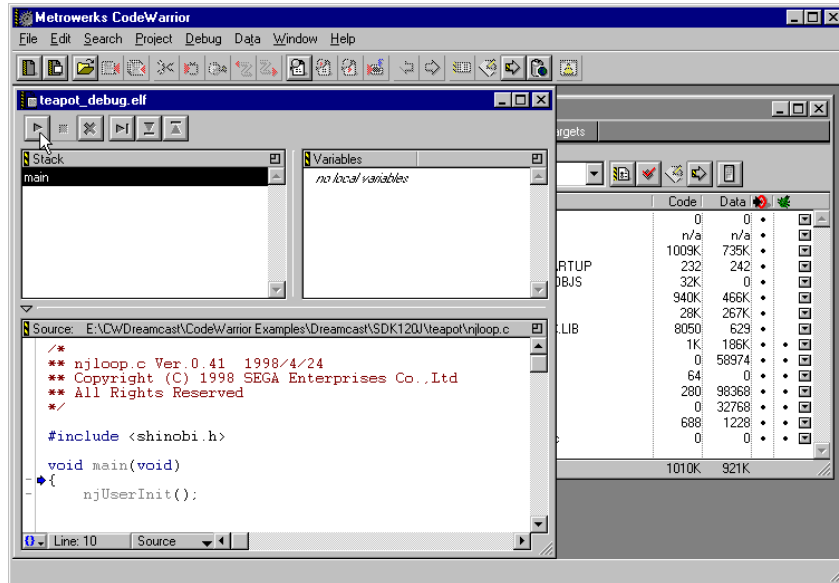
This chapter contains the following topics:

- [Debugging with CodeWarrior](#)
- [Using printf\(\)](#)
- [Debugging Static Libraries](#)

Debugging with CodeWarrior

Choose **Projects > Debug** to bring up the debugger program window as shown in [Figure 7.1](#).

Figure 7.1 The Program window



In the program window contains the stack crawl pane, the variables window, and the code window. The debugger control bar is at the top of the window. From here, you can run, stop, and single-step through your program.

For detailed explanations and guidance, please see our *Debugger User Guide*.

Using `printf()`

The `printf()` function will only work if you include the 'mw output.lib' library in your project. The output from your `printf()` functions will appear in the debugger log window.

Debugging Static Libraries

You can debug static libraries as part of a larger application, but you cannot debug them on their own.



Debugging With Codescape

This chapter discusses how to use CodeWarrior in conjunction with Codescape to debug Dreamcast code.

This chapter includes the following topics:

- [Debugging with the Codescape debugger](#)
- [Using `printf\(\)`](#)

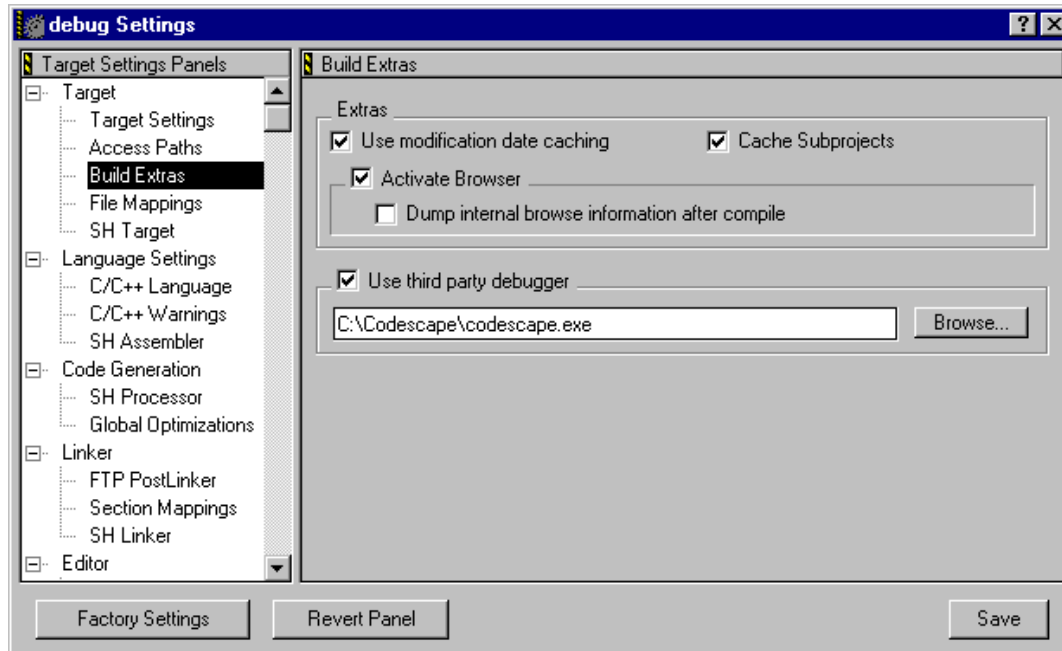
NOTE: Please see the Debugger release notes for the latest news about our Codescape interoperability.

Debugging with the Codescape debugger

To have CodeWarrior launch the Codescape debugger when you select **Debug** from the **Project** menu, you must specify Codescape as your third-party debugger.

Set Codescape to be your third-party debugger in the **Build Extras** target settings panel shown in [Figure 8.1](#). Click the **Use third party debugger** box, and enter the path to your Codescape executable.

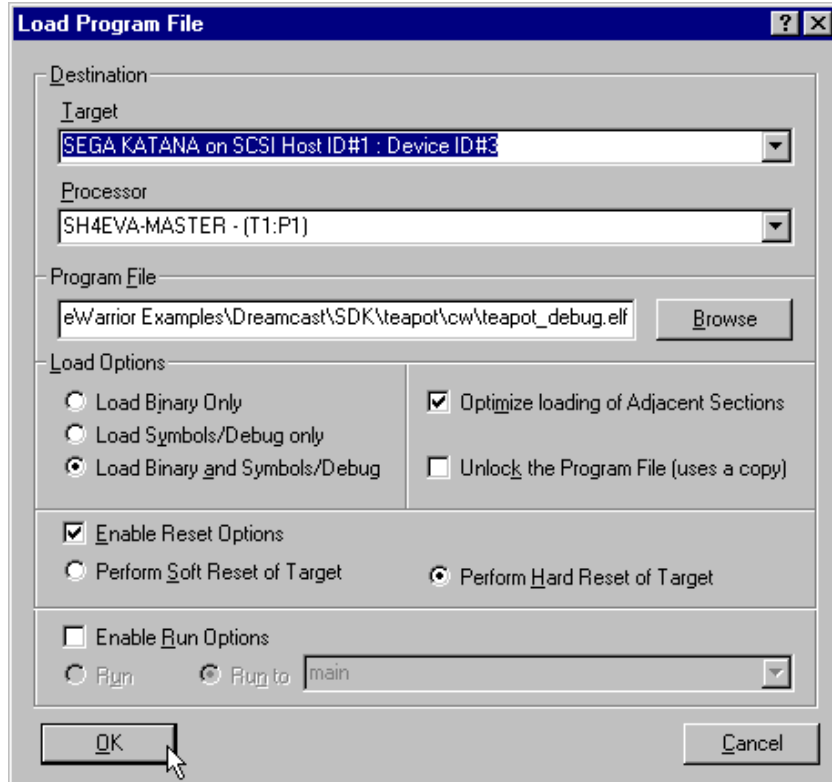
Figure 8.1 Set CodeScape to be your third-party debugger



Now when you select **Debug** from your project, CodeWarrior will automatically launch the Codescape debugger.

Once you are in Codescape, you will need to click **File > Load Program File** to load your CodeWarrior-built executable into the debugger. In [Figure 8.2](#), we illustrate how you would do this for the SDK teapot executable, `teapot_debug.elf`

Figure 8.2 Codescape's 'load program file' menu



Please see the *Codescape User Guide* for detailed instructions on how to use the Codescape debugger.

Using printf()

The `printf()` function will only work if you include the 'mw_output.lib' library in your project. The output from your `printf()` functions will appear in the debugger log window.

Debugging With Codescape

Using *printf()*



Target Settings for Dreamcast

This chapter discusses each of the settings panels that affect code generation for Dreamcast development. By modifying the settings for the individual items within a panel you control the compiler, linker, and other aspects of code generation.

Specific details about how the compiler and linker work for Dreamcast development, such as compiler pragmas, linker symbols and so forth, is found in *C and C++ for Dreamcast*.

The sections in this chapter are:

- [Target Settings Overview](#)
- [Settings Panels for Dreamcast](#)

Target Settings Overview

Each target in a CodeWarrior project has its own individual settings. These settings control a variety of features such as compiler options, linker output, error and warning messages, and so forth. You modify these settings through the Target Settings dialog box. This interface is fully explained in the *IDE User Guide*.

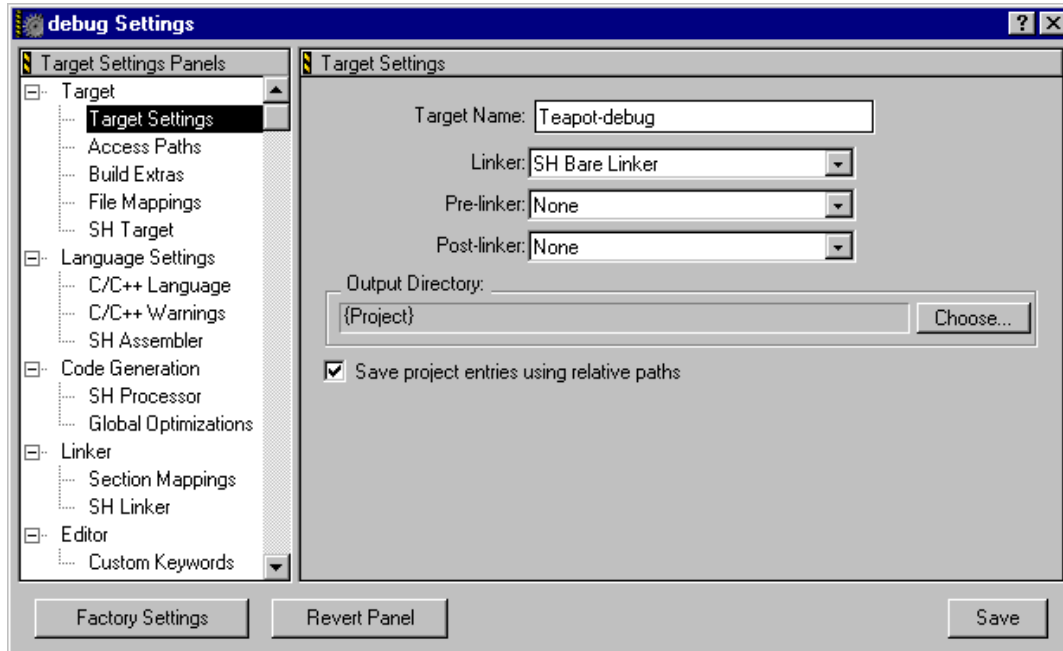
In brief, you control compiler and linker behavior for a particular target by modifying settings in the appropriate settings panels in the Target Settings dialog box. To open any settings panel, choose **Target Settings** from the **Edit** menu, where **Target** is the current target in the CodeWarrior project. Or, go to the Target view of the Project window and double-click the target of interest.

When you do, the Target Settings dialog box appears, as shown in [Figure 9.1](#).

Target Settings for Dreamcast

Target Settings Overview

Figure 9.1 Target Settings dialog box



Select the panel you wish to see from the hierarchical list of panels on the left side of the dialog box. When you do, that panel appears. You can then modify the settings to suit your needs.

When you modify the settings on a panel, you can restore the previous values by using the **Revert Panel** button at the bottom of the dialog box. To restore the settings to the factory defaults, use the **Factory Settings** button at the bottom of the panel.

TIP: Use project stationery when you create a new project. The stationery has all settings in all panels set to reasonable or default values. You can create your own stationery file with your preferred settings. Modify a new project to suit your needs, then save it in the stationery folder. See the *IDE User Guide* for details.

Settings Panels for Dreamcast

This section discusses those panels that are specific to Dreamcast development, and the purpose and effect of each setting. The panels are:

- [Target Settings](#)
- [SH Target](#)
- [SH Assembler](#)
- [SH Processor](#)
- [Global Optimizations](#)
- [Section Mappings](#)
- [SH Linker](#)
- [Debugger Settings](#)

Settings panels of more general interest are discussed in other CodeWarrior manuals. [Table 9.1](#) lists several panels and where you can find information about them.

Table 9.1 **Where to find information on other settings panels**

Panel	Manual
Access Paths	<i>IDE User Guide</i>
Build Extras	<i>IDE User Guide</i>
File Mappings	<i>IDE User Guide</i>
Custom Keywords	<i>IDE User Guide</i>
C/C++ Language	<i>C Compilers Reference</i>
C/C++ Warnings	<i>C Compilers Reference</i>

Target Settings

The Target Settings *dialog box* contains a Target Settings *panel*. The dialog box and the panel are not the same. The dialog box displays

Target Settings for Dreamcast

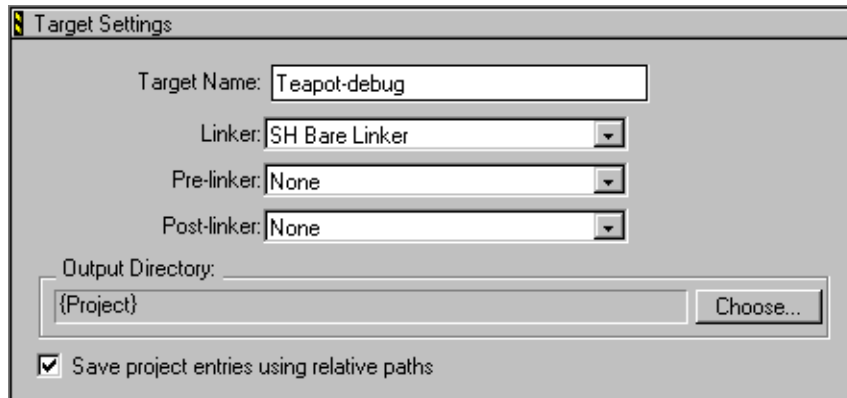
Settings Panels for Dreamcast

all panels, one at a time. The Target Settings *panel* is one of those panels.

The Target Settings panel, shown in [Figure 9.2](#), is perhaps the most important panel in CodeWarrior. This is the panel where you pick your target. When you select a linker in the Target Settings panel, you specify the target operating system and/or chip. The other panels listed in the Settings dialog box will change to reflect your choice.

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

Figure 9.2 The Target Settings panel



NOTE: The Target Settings panel is not the same as the [SH Target](#) panel. You specify the target in the Target Settings panel. You set other project options in the [SH Target](#) panel.

The items in this panel are:

[Target Name](#)

[Post-Linker](#)

[Linker](#)

[Output Directory](#)

[Pre-Linker](#)

[Save Project Entries Using Relative Paths](#)

Target Name

Use the Target Name text field to set or change the name of a target. When you use the Targets view in the Project window, you will see the name that you have set.

The name you set here is *not* the name of your final output file. It is the name you assign to the target for your personal use. The name of the final output file is set in the [SH Target](#) panel.

Linker

Choose a linker from the items listed in the Linker pop-up menu. For Dreamcast, use **SH Bare Linker**

Pre-Linker

Some targets have pre-linkers that perform work on object code before it is linked. There is no pre-linker for Dreamcast development.

Post-Linker

Some targets have post-linkers that perform additional work (such as object code format conversion) on the final executable. There is no post linker for Dreamcast development.

Output Directory

This is the directory where your final linked output file will be placed. The default location is the directory that contains your project file. Click the **Choose** button to specify another directory.

Save Project Entries Using Relative Paths

To add two or more files with the same name to a project, select this option. When this option is off, each project entry must have a unique name.

When this option is selected, the IDE includes information about the path used to access the file as well as the file name when it stores information about the file. When searching for a file, the IDE com-

bines **Access Path** settings with the path settings it includes for each project entry.

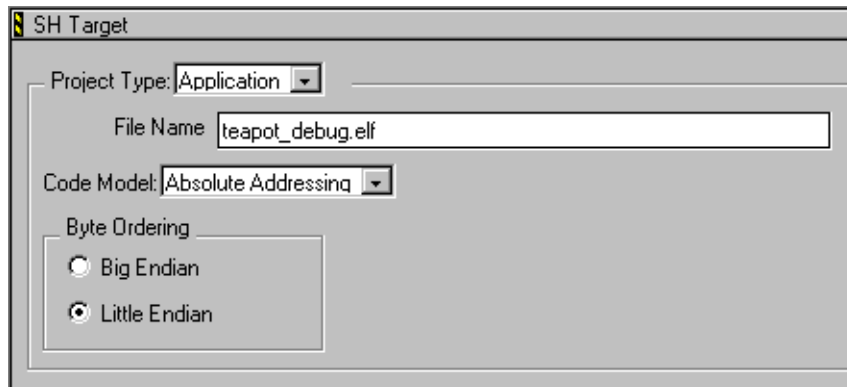
When this option is off, the IDE only records information about each project entry's file name. When searching for a file, the IDE only uses Access Paths.

SH Target

The SH Target panel, shown in [Figure 9.3](#), is where you set the name of your final output file.

The settings you can specify in this panel depend on the type of project you are creating.

Figure 9.3 The SH Target panel.



The items in this panel are:

[Project Type](#)

[File Name](#)

[Code Model](#)

[Byte Ordering](#)

Project Type

The **Project Type** pull-down menu determines the kind of project you are creating. The available project types are shown in [Figure 9.4](#)

Figure 9.4 SH Target type options



Set this menu so that the selected menu item reflects the kind of project you are building. You typically want to build an Application.

File Name

The **File Name** edit field specifies the name of the executable or library you create. Our convention is to end this name with the extension `.elf` for executables and `.elf.lib` for libraries.

Byte Ordering

The **Byte Ordering** radio button controls whether the code generated is stored in little endian or big endian format. In big endian format, the most significant byte comes first (B3 B2 B1 B0). In little endian format, the bytes are organized with the least significant byte first (B0 B1 B2 B3).

For Dreamcast applications, this option must be set to **Little Endian**.

Code Model

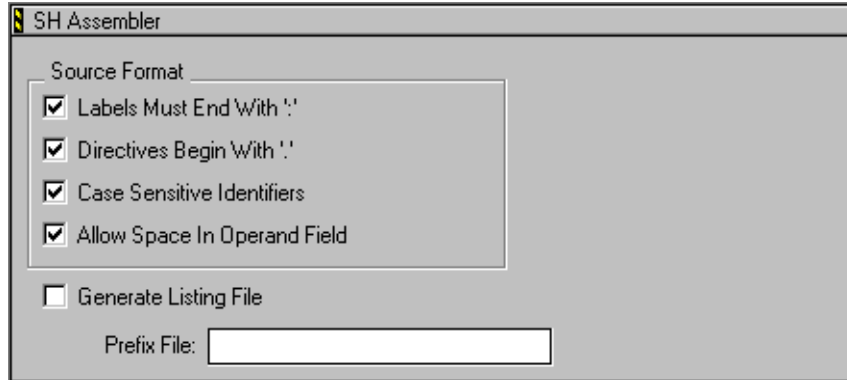
The **Code Model** pull-down menu determines the addressing mode for the generated executable.

For Dreamcast applications, this option must be set to **Absolute Addressing**.

SH Assembler

The SH Assembler panel, shown in [Figure 9.5](#), controls how the SH assembler processes assembly language instructions.

Figure 9.5 The SH Assembler panel



The items in this panel are:

[Labels Must End With ':'](#)

[Directives Begin With '.'](#)

[Case Sensitive Identifiers](#)

[Allow Space In Operand Field](#)

[Generate Listing File](#)

[Prefix File](#)

Labels Must End With ':'

Specifies that labels must end with a colon character (:).

Directives Begin With '.'

Specifies that assembler directives begin with a period character (.).

Case Sensitive Identifiers

Displays identifiers using the same letter case used in source code. When deselected, identifiers appear in uppercase only.

Allow Space In Operand Field

Allows you to use space characters to separate operands

Generate Listing File

Determines whether or not a listing file will be generated when the source files in the project are assembled.

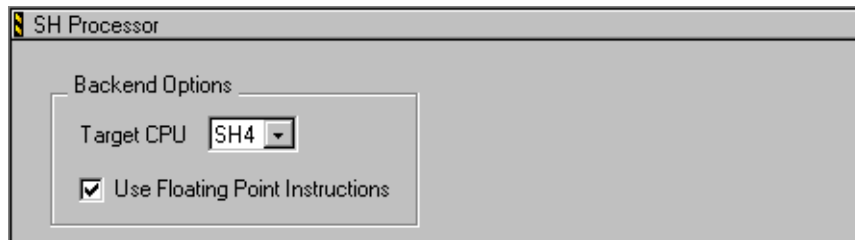
Prefix File

Defines a file that is automatically included in all assembly files in the project. This field allows you to include common definitions without including the file in every source file.

SH Processor

The SH Processor panel, shown in [Figure 9.6](#), is where you control settings related to code generation for the Dreamcast platform.

Figure 9.6 The SH Processor panel.



The items in this panel are:

[Target CPU](#)

[Use Floating Point Instructions](#)

Target CPU

Defines the CPU for which the compiler generates code. For Dreamcast, this should be set to **SH4**.

Use Floating Point Instructions

If this option is active, the compiler makes use of the processor's floating point instructions.

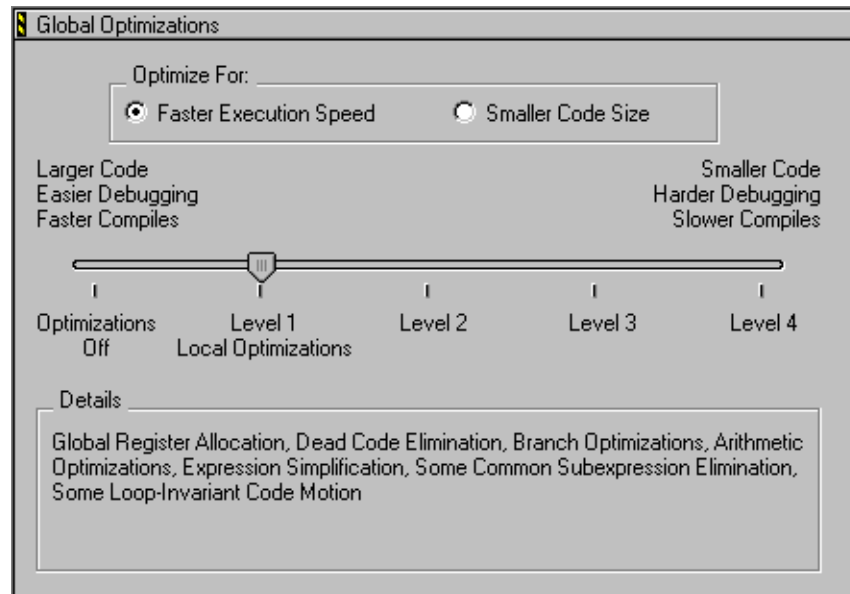
If this option is not active, the compiler calls runtime routines for floating-point operations. The processor's floating point registers will not be used.

NOTE: In this release, this option is ignored, and the floating point registers are always used.

Global Optimizations

The Global Optimization panel, shown in [Figure 9.7](#), controls the method and depth by which the compiler optimizes your code.

Figure 9.7 Global Optimization panel



The items in this panel are:

[Optimize For](#)

[Optimization Level Slider](#)

Optimize For

Use these options to configure how the CodeWarrior IDE optimizes your code.

- **Faster Execution Speed**

This option improves the execution speed of object code. Object code is faster, but may be larger.

- **Smaller Code Size**

This option reduces the size of object code that the compiler produces. Object code is smaller, but may be slower.

Optimization Level Slider

Use the slider to determine the level of optimization applied to your code. You can choose to turn off code optimizations, or you can choose to apply one of four levels of optimization. The higher the level that you select, the more optimizations are applied to your code.

The Details text field, below the slider, lists the optimizations that are applied. [Table 9.2](#) repeats the information found in the Details text field. For more information about these optimizations, see [“Optimizing Code for Dreamcast” on page 68](#).

Table 9.2 **SH optimizer levels**

Level	Effect	Debugging
0	Global Register Allocation for temporary values	safe
1	Global Register Allocation Dead Code Elimination Loop Invariant Code Motion Branch Optimization Arithmetic Optimizations Expression Simplification	not safe
2	Common Sub-Expression Elimination Instruction Scheduling Delay-slot Filling Copy and Expression Propagation Peephole Optimization	not safe

Target Settings for Dreamcast

Settings Panels for Dreamcast

Level	Effect	Debugging
3	Dead Store Elimination Strength Reduction Lifetime Based Register Allocation Loop Unrolling Loop Transformations Life Range Splitting Vectorization	not safe
4	Optimizations are repeated	n/a

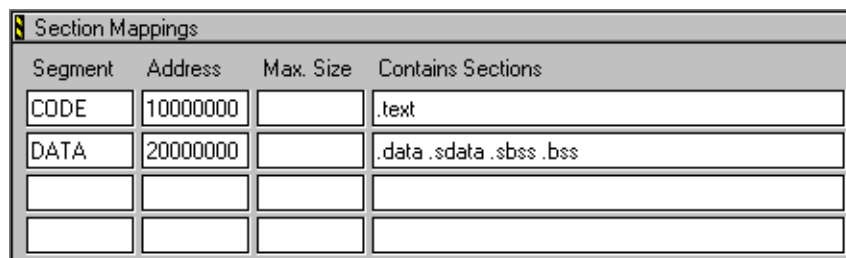
NOTE: If you use Smart inlining, do not use Level 0 optimization.

Section Mappings

The Section Mappings panel, shown in [Figure 9.8](#), maps your sections layout for the linker. In the absence of a linker command file, the linker uses the information in the Section Mappings panel to link your code.

You cannot use Section Mappings with the Dreamcast SDK libraries because of the complexity of defining the mapping and special link considerations. You must use a linker command file instead. However, the Section Mapping is useful when building programs that do not use the SDK libraries.

Figure 9.8 The Section Mappings panel.



The items in this panel are:

[Segment](#)

[Address](#)

[Max Size](#)

[Contains Sections](#)

Segment

The user defined name for a segment that contains one or more sections.

Address

The starting address for the segment

Max Size

The maximum size of the segment. The linker will report an error if this size is exceeded.

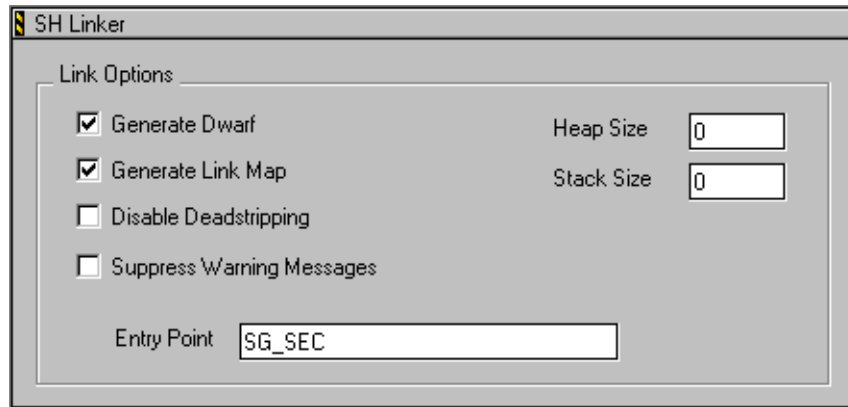
Contains Sections

The names of the sections that are contained in the segment.

SH Linker

The SH Linker panel, shown in [Figure 9.9](#), is where you control settings related to linking your object code into final form, be it executable, library, or other type of code.

Figure 9.9 The SH Linker panel.



These items in this panel are:

[Generate Dwarf Info](#)

[Heap Size](#)

[Generate Link Map](#)

[Stack Size](#)

[Disable Deadstripping](#)

[Suppress Warning Messages](#)

[Entry Point](#)

Generate Dwarf Info

The linker includes debug information generated by the compiler. You can not debug your program unless this information is present.

Generate Link Map

When this setting is on, the linker generates a link map. When this setting is off, the linker does not generate a link map.

The link map shows which file provided the definition for every object and function in the output file. It also displays the address given to each object and function, a memory map of where each section will reside in memory, and the value of each linker generated symbol.

Disable Deadstripping

Enabling the **Disable Deadstripping** option will prevent the linker from removing dead code.

Suppress Warning Messages

The **Suppress Warning Messages** checkbox controls whether the linker displays warnings. This checkbox is not supported in this release.

Heap Size

This is not used. The heap size is specified by the Dreamcast SDK libraries.

Stack Size

This is not used. The stack size is specified by the Dreamcast SDK libraries.

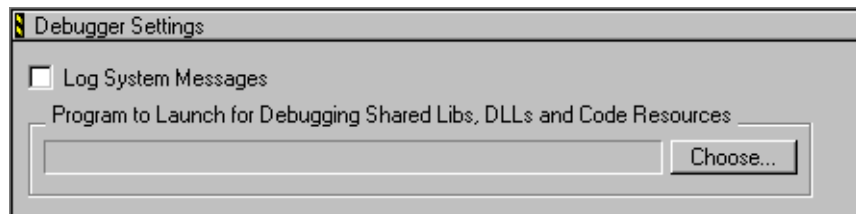
Entry Point

This is the name of the first function that will be called. Its value is the starting address of your program.

Debugger Settings

The Debugger Settings panel shown in [Figure 9.9](#) is not used in this release.

Figure 9.10 The Debugger Settings panel.



Target Settings for Dreamcast

Settings Panels for Dreamcast



C and C++ for Dreamcast

This chapter describes both the Metrowerks back-end compiler and linker for Dreamcast.

The sections in this chapter are:

- [Number Formats for Dreamcast](#)
- [Calling Conventions for Dreamcast](#)
- [Variable Allocation for Dreamcast](#)
- [Optimizing Code for Dreamcast](#)
- [Linker Issues for Dreamcast](#)
- [C++ issues for Dreamcast](#)

However, this chapter does *not* discuss front-end compiler issues, support for inline assembly, compiler and linker errors, controlling the size of C++ code, and so forth. These topics are covered in other CodeWarrior documentation as outlined in [Table 10.1](#).

Table 10.1 Other compiler/linker documentation

For this topic...	See...
how CodeWarrior implements the C/C++ language	<i>C Compilers Reference</i> generally
using C/C++ Language and C/C++ Warnings settings panels	<i>C Compilers Reference</i> , “Setting C/C++ Compiler Options” chapter
controlling the size of C++ code	<i>C Compilers Reference</i> , “C++ and Embedded Systems” chapter

For this topic...	See...
using compiler pragmas	<i>C Compilers Reference</i> , “Pragmas and Symbols” chapter
initiating a build, controlling which files are compiled, handling error reports	<i>IDE User Guide</i> , “Compiling and Linking” chapter
information about a particular error	<i>Error Reference</i>
inline assembly	Inline Assembler and Intrinsics for Dreamcast
Dreamcast assembler	SH processor manual

NOTE: Some of the items discussed in this chapter may actually be implemented in the front-end compiler. However, it really doesn't matter whether the actual implementation of a feature occurs in the front-end or back-end compiler. From the programmer's point of view, it is all one compiler.

Number Formats for Dreamcast

This section describes how the CodeWarrior C/C++ compiler implement integer and floating-point types for the Dreamcast processor. You can also read `limits.h` for more information on integer types, and `float.h` for more information on floating-point types.

The topics in this section are:

- [Dreamcast Integer Formats](#)
- [Dreamcast Floating-Point Formats](#)

Dreamcast Integer Formats

The Dreamcast back-end compiler does not allow you to change the sizes of integers. Thus, the size of a `short int` is always 2 bytes, and the size of `int` or `long int` is always 4 bytes.

[Table 10.2](#) shows the size and range of the integer types for the Dreamcast compiler.

Table 10.2 **Dreamcast integer Types**

Type	Size	Range
<code>bool</code>	8 bits	true or false
<code>char</code>	8 bits	-128 to 127
<code>unsigned char</code>	8 bits	0 to 255
<code>short</code>	16 bits	-32,768 to 32,767
<code>unsigned short</code>	16 bits	0 to 65,535
<code>int</code>	32 bits	-2,147,483,648 to 2,147,483,647
<code>unsigned int</code>	32 bits	0 to 4,294,967,295
<code>long</code>	32 bits	-2,147,483,648 to 2,147,483,647
<code>unsigned long</code>	16 bits	0 to 4,294,967,295
<code>long long</code>	not supported	not supported

Dreamcast Floating-Point Formats

[Table 10.3](#) shows the sizes and ranges of the floating point types for the Dreamcast compiler.

NOTE: `double` is currently implemented as `float`

Table 10.3 Dreamcast floating point types

Type	Size	Range
float	32 bits	1.17549e-38 to 3.40282e+38

Calling Conventions for Dreamcast

This section describes the C/C++ calling conventions for Dreamcast development.

CodeWarrior is fully compliant with Hitachi's ABi specifications. Hitachi's compiler conventions are documented in the *SH Series C Compiler User's Manual*, available from Hitachi.

Variable Allocation for Dreamcast

(K&R, §A4.3, §A8.3, §A8.6.2) This section describes how the C/C++ compiler allocates space for variables.

The compiler places no limits on how large your variables may be, or how you allocate them.

Optimizing Code for Dreamcast

This section discusses optimizations that are specific to Dreamcast development with CodeWarrior. They are activated and deactivated through the Global Optimization panel described in [“Global Optimizations” on page 58](#).

The optimizations are:

- [Global Register Allocation](#)
- [Loop Invariant Code Motion](#)
- [Dead Code Elimination](#)
- [Dead Store Elimination](#)
- [Common Sub-Expression Elimination](#)

- [Instruction Scheduling](#)
- [Delay-slot Filling](#)
- [Copy and Expression Propagation](#)
- [Peephole Optimization](#)
- [Strength Reduction](#)
- [Lifetime Based Register Allocation](#)
- [Loop Unrolling](#)

Global Register Allocation

In this optimization, the compiler assigns two or more variables to the same register. It does this if the code does not use the variables at the same time. In this example, the compiler could place `i` and `j` in the same register:

```
short i;
int j;

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

However, if a line of code like the one below appears anywhere in the function, the compiler would realize that you are using `i` and `j` at the same time, and place them in different registers.

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

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

If register allocation is on while you debug your code, it may appear as though there's something wrong with the variables that share a single register. In the example above, `i` and `j` would always have the same value. When `i` changes, `j` changes in the same way, and vice versa.

Register allocation is activated from the [SH Processor](#) panel by selecting optimization level 1. Because it can affect debugging, we rec-

ommend you use optimization level 0 when compiling your debug targets.

Loop Invariant Code Motion

This optimization moves computations that don't change on the inside of the loop. They are moved to the outside of the loop to improve the loop's speed. With this option, your object code is faster.

Dead Code Elimination

The compiler removes statements that logically can never be executed, or statements that are never referred to by other statements. The result is that your object code is smaller.

Dead Store Elimination

Removes assignments to a variable if the variable is not used before being reassigned again. With this option on, object code is smaller and faster.

Common Sub-Expression Elimination

The compiler replaces similar redundant expressions with a single expression. For example, if two consecutive statements both use the expression $a * b * c + 10$, the compiler generates object code that computes the expression only once, and applies the resulting value to both statements.

With this optimization, your object code is smaller and faster.

Instruction Scheduling

The compiler uses the *instruction scheduling* optimization to increase the speed of execution. When possible, this optimization rearranges processor instructions so that the execution of one instruction doesn't delay the execution of others.

Delay-slot Filling

Delay-slot filling is an optimization used by the compiler to fill in the delay-slot of delay-slot instructions. As an example, take the following sequence:

```
JSR  
NOP
```

JSR is a delay-slot instruction, but in this case its delay-slot is inactive. You could take advantage of its delay-slot feature by adding an instruction after JSR.

```
JSR  
instruction
```

When delay-slot filling is active, *instruction* will be placed in the delay-slot of the JSR instruction. The instruction in the delay-slot will be executed before the JSR.

Copy and Expression Propagation

Replaces multiple occurrences of one variable with a single occurrence. With this option on, object code is smaller and faster.

Peephole Optimization

Applies local optimizations to small sections of your code. With this option, the optimized sections of code are faster.

Strength Reduction

Replaces multiplication instructions that are inside loops with addition instructions to speed up the loop. With this option, object code is larger, but executes faster.

Lifetime Based Register Allocation

Uses the same processor register for different variables in the same routine if the variables aren't used in the same statement. With this option on, object code executes faster.

Loop Unrolling

The compiler performs loop unrolling when the optimization level is set to Level 3 or Level 4. The unrolling factor is set to 2. As long as the loop does not have more than 20 instructions, the loop will be unrolled.

To disable loop unrolling, add the following pragma to your source code:

```
#pragma opt_unroll_loops off
```

Pragmas for Dreamcast

The pragmas supported by the CodeWarrior for Dreamcast compiler are defined in the *C Compilers Reference*. A PDF version of this manual is located in your CodeWarrior Documentation folder.

[Table 10.4](#) lists some of the pragmas that are not supported for Dreamcast development.

Table 10.4 Pragmas not supported for Dreamcast

code_seg	define_section	disable_registers
interrupt	longlong	longlong_enums
no_register_coloring	peephole	register_coloring
scheduling	section	stack_cleanup
use_fp_instructions		

Linker Issues for Dreamcast

This section discusses the background information on the Dreamcast linker and how it works. The topics in this section are:

- [Linker Command File](#)
- [Deadstripping Unused Code and Data](#)

- [Link Order](#)

Linker Command File

When the **Section Mappings** panel does not give you enough control over the link process, you can use a linker command file (LCF). This file must be included as part of your project. The linker command file defines the arrangement of your program and data sections.

The CodeWarrior linker supports the features listed here. Optional parameters are indicated by square brackets, '[' and ']'. If there are variables to be defined, these are indicated by italic text.

- [Comments](#)
- [Assignments](#)
- [Location Counter](#)
- [Symbols](#)
- [Arithmetic operations](#)
- [Alignment](#)
- [\\$segment directive](#)
- [\\$include directive](#)
- [\\$output_name directive](#)

You can use these features in your linker command file. For an example linker command file that uses all these constructs, please see [Listing 10.3 on page 77](#)

Comments

You may add comments by using the pound sign, '#'. Characters appearing to the right of the pound sign will be ignored by the parser. The following are valid comments:

```
# This is a one-line comments
.data    # This is a partial-line comment
```

Assignments

You can create global symbols and assign addresses to them using the standard assignment operator as shown:

```
_symbolicname = expression;
```

A semicolon is required at the end of an assignment statement. An assignment may only be used at the start of an expression. The following assignment is illegal:

```
_sym1 + _sym2 = _sym3;           #illegal
```

Location Counter

The period character, '.', always maintains the current position of the output location. Since the period always refers to a location in a section, it must always appear *within* a \$section statement.

The period character may appear anywhere that a symbol is allowed. [Listing 10.1](#) is an example of using the location counter.

Listing 10.1 Usage of location counter

```
$segment DATA 0x30000000 LENGTH 0 R
{
    _start_data = .;
    #_start_data now contains the starting address of .data

    .data
    _end_data = .;
    #_end_data now contains the address just past the end of .data
}
```

Symbols

All symbols defined in the linker command file must begin with an underscore and be defined inside a segment.

```
_start_data = .;
_address = 0x2544000;
```

Arithmetic operations

You may use standard C arithmetic operators. All operators are left-associative. For more information on C arithmetic operators, refer to the *C Compilers Reference*.

```
_sizeof_data = _end_data - (_start_data);
```

Alignment

You can force alignment in the linker command file with the `ALIGN` command. The section group directly after the `ALIGN` command is affected.

```
ALIGN (0X8)  
*(.data)
```

\$segment directive

Description	Defines segment boundaries and contents.
Prototype	<pre>\$segment name [<i>baseaddress</i>] [LENGTH <i>length</i>] [R] { symbols and sections go here }</pre>
Remarks	<p>An unspecified base address implies that you want the segment to be placed immediately following the last segment. It will be properly aligned. For unlimited segment length, use a length of 0. The R option will process the section for ROM.</p> <p>Examples of usage is given in Listing 10.2.</p>

Listing 10.2 Examples of \$segment usage

```
# The following segment has a base address of 0x2F001050 and a
# maximum length of 0xC000. The linker will give an error if it
# is larger.
$segment CODE 0x2F001050 LENGTH 0xC000
{
# .text from all files will be mapped to this segment
    *(.text)
}
#####
# The following segment has no maximum length, and will be
# processed for ROM.

$segment DATA 0x30000000 LENGTH 0 R
{
# .data from all files will be mapped to this segment
    *(.data)
}
#####
# The following segment will have the address immediately
# following the last
# segment (properly aligned). It will also have unlimited length
# size, since the length is not indicated.

$segment BSS
{
# .data from the foo.c will be remapped to this segment
    foo.c(.data)
}
```

\$include directive

Description	Force a symbol or section into closure.
Prototype	<pre>\$include { _my_unreferenced_function_name .myunreferencedsection }</pre>
Remarks	Using <code>\$include</code> prevents the linker from deadstripping a symbol or section that is not directly referenced.

\$output_name directive

Description	Change the name of the output file without changing the file name extension.
Prototype	<pre>\$output_name { my_output_name }</pre>

Listing 10.3 A model linker command file

```
# This is a comment.      18/2/99

# The following segment will have a base address
# of 0x2f001050 and a maximum length of 0xC000.
# The linker will emit an error if it is larger.

$segment CODE 0x2f001050 LENGTH 0xC000
{

# .text from all files will be mapped to this segment.

    *(.text)

# The following symbol, _my_stack_symbol, is being assigned
# to the value of a symbol, __stack_begin, generated internally by
```

C and C++ for Dreamcast

Linker Issues for Dreamcast

```
# the linker. This can be useful for porting code when they
# use a symbol that is different from the one generated by our
# linkers.
# Note: not all of the linkers use "__SP_INIT" for the stack
# pointer.
```

```
    _my_stack_symbol = __stack_begin;
}
```

```
# The following segment will have a base address of
# 0x30000000 and an unlimited length. A length of
# zero or no length indicates this. It will also be
# processed for ROM.
```

```
$segment DATA 0x30000000 LENGTH 0 R
{
```

```
# All symbols defined in the command file must begin with
# an underscore and must be defined inside a segment.
```

```
# The following symbol will be assigned to the value of the
# base address of the .data section group if it exists, assignment
# to '.' means position in the output:
```

```
    _begin_data = .;
```

```
# This will force the next section group encountered to an 8 byte
# alignment.
```

```
    ALIGN(0x8)
```

```
    *(.data)
```

```
# The following symbol will be assigned to the address immediately
# following the last byte of the .data section group since there
# are no section groups following it:
```

```
    _end_data = .;
```

```
# Arithmetic operations are allowed:
```

```
_sizeof_data = _end_data - (_begin_data);
}

# The following segment will have the address immediately
# following the last segment (properly aligned). It will
# also have unlimited size since the length is not indicated.

$segment BSS
{
    *(.bss)
}

$segment FOO_DATA
{

# .data from foo.c will be mapped to this segment. This overrides
# the previous mapping *(.data) in segment DATA for .data from
foo.c
# only.

    foo.c(.data)
}

# The following symbols will be forced into closure. If
# it is a section group name, all sections of that name will
# forced into closure. Useful for symbols and sections that
# are not directly referenced.

$include
{
    _my_unreferenced_function_name
    .myunreferencedsection

# The following must be done if the sections were mapped from
specific file
# to a segment (e.g. .data in foo.c).

    .data_foo.c
}
```

```
# The output file names will be changed to the following
# entry, extensions will remain the same.
$output_name
{
    my_output_name
}
```

Deadstripping Unused Code and Data

The Shinobi libraries and libraries built with the CodeWarrior C/C++ compiler only contribute the used objects to the linked program. If a library has assembly or other C/C++ compiler built files, only those files that have at least one referenced object contribute to the linked program. Completely unreferenced object files are always ignored.

The Dreamcast linker deadstrips unused code and data from files compiled by the CodeWarrior C/C++ compiler. Other assembler relocatable files and C/C++ object files built by other compilers are not deadstripped.

If you have unreferenced sections of code or data that must be kept in the final application, you may use the [\\$include directive](#) of the linker command file to prevent the linker from deadstripping those unreferenced sections. For more information about the `$include` directive and others, see [“Linker Command File” on page 73](#). You can also set the **Do No Deadstrip** option in the **SH Linker** preferences panel. For a description of this panel, see [“SH Linker” on page 61](#).

Link Order

Link order is generally specified in the Link Order view of the Project window. For general information on setting link order, see the *IDE User Guide*.

The link order of the libraries is very important. The default stationery is set up with the correct link order for the libraries. If you are

not using the stationery, please make sure that the libraries are linked in this exact order:

```
strt1.obj.elf  
strt2.obj.elf  
systemid.obj.elf  
toc.obj.elf  
sg_sec.obj.elf  
sg_arejp.obj.elf  
sg_areus.obj.elf  
sg_areec.obj.elf  
sg_are00.obj.elf  
sg_are01.obj.elf  
sg_are02.obj.elf  
sg_are03.obj.elf  
sg_are04.obj.elf  
sg_ini.obj.elf  
aip.obj.elf  
zero.obj.elf
```

Place other libraries and your source files after the listed libraries.

The Dreamcast linker ignores executable files that are in the project. You may find it convenient to keep the executable there so that you can disassemble it. If a build is successful, the file will show up in the project as out of date (there will be a check mark in the touch column on the left side of the project window) because it is a new file. If a build is unsuccessful, the IDE won't be able to find the executable file and will stop the build with an appropriate message.

C++ issues for Dreamcast

To access the standard C++ libraries, you can add the `MSLCppDC.lib` library to your project. This is our standard C++ library.

We support C++ fully in this release, with the following exceptions:

- defining member templates / nested class template members outside of the template definition

- member template conversion functions
- member template friends
- template template arguments
- 'exported' templates
- there is no support for exceptions in this release
- there is no support for stream classes.
- there is no support for IO.



Inline Assembler and Intrinsic Functions for Dreamcast

This chapter describes support for inline assembly language programming built into the CodeWarrior compiler. For more information on Dreamcast assembly instructions, refer to the hardware manual of the SH processor.

The sections in this chapter are:

- [Working with Inline Assembly](#)
- [Assembler Directives](#)
- [Intrinsic Functions](#)
- [Mnemonics for Inline Assembly](#)

Working with Inline Assembly

This section describes how to use the compiler's built-in support for assembly language programming.

The topics in this section include:

- [Inline Assembler Syntax](#)
- [Using Labels](#)
- [Using Comments](#)
- [Using Registers](#)

Inline Assembler Syntax

There are two ways to add assembly language statements to a C or C++ source code file.

The first method is shown in [Listing 11.1](#). This method uses the `asm` qualifier to specify that all statements in a function are in assembly language. You may define local variables in functions defined with the `asm` qualifier.

Listing 11.1 Defining a function with `asm`

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

The second method is shown in [Listing 11.2](#). This method uses the `asm` qualifier as a statement to provide “inline” assembly language instructions.

In other words, assembly language statements and regular C/C++ statements can be combined within the same function definition. However, the inline `asm` statements are not allowed to reference that function’s local variables.

Listing 11.2 Inline assembly with `asm`

```
int MyInlineAsmFunction(void)
{
    /* Local variable definitions and C/C++ statements */
    asm { /* Assembly language instructions */ }
    /* Local variable definitions and C/C++ and asm {} statements */
}
```

To ensure that the C/C++ compiler recognizes the `asm` keyword, you must turn off the **ANSI Keywords Only** option in the C/C++

language settings panel. This panel and its options are fully described in the *C Compilers Reference*.

The built-in assembler supports all the standard SH assembler instructions.

TIP: To enter a few lines of assembly language code within a single function, you can use the compiler's support for intrinsic functions instead of inline assembler. See [“Intrinsic Functions” on page 90](#).

Keep these points in mind as you write assembly functions:

- Some optimizations may be performed on assembly language functions and functions that contain `asm` blocks. This depends on your compiler optimization setting. For information on setting the optimization level, see [“Global Optimizations” on page 58](#).

You may suppress assembly optimizations by using the `..set noreorder` directive. For information on the `..set` directive, see [“`..set`” on page 89](#).

- All statements must either be a label, like this:

[*LocalLabel* :]

or be an instruction, like this:

((*instruction* \ *directive*) [*operands*])

- Each statement must end with a newline.
- The compiler will not recognize variables that are initialized inside blocks of inline assembly.
- Assembler directives, instructions, and registers are case-sensitive and must be in uppercase. For example, these two statements are different:

```
ADD      R2, R4          // OK
add      r2, r4          // ERROR
```

- Hex constants must be in C-Style.

```
0x123ABC // OK
$123ABC  // ERROR
H'123ABC // ERROR
```

Using Labels

A label can be any identifier that you have not already declared as a local variable. A label must end with a colon. An instruction cannot follow a label on the same line. Take the following as an illustration:

```
x1:    ADD    R2,R3    // ERROR
x2:                                // OK
      ADD    R2,R3    // OK
```

Listing 11.3 Example of Using Labels

```
extern void foo(void);

int foo() {
    asm
    {
        MOVA    foo_addr, R0;
foo_addr:
        .data.w 0;
        .data.l foo;
    }
}
```

Using Comments

You can use C and C++ comments, but you cannot use a semicolon ';' to denote a comment. For example:

```
ADD    R2,R4    // OK
ADD    R2,R4    /* OK */
ADD    R2,R4    ; ERROR
```

Using Registers

In [Listing 11.4](#), we see three assembly statements embedded within a function. To reference 'i' directly from the inline assembly statement, we type the variable as a register.

Listing 11.4 Example of using registers

```
int foo3(int register i){
    asm{
        MOV i,R1;
        ADD 1, R1;
        MOV R1, R4;
    }
    return i;
}
```

Status Register

The status register can be read and set through inline assembly. See [Listing 11.5](#) for an example.

Listing 11.5 Example of using the status register

```
/* Get status register */
static inline unsigned int get_sr(void)
{
    register unsigned int sr = 0;

    asm
    {
        STC SR, sr
    };
    return sr;
}
/* Set status register */
static inline void set_sr(unsigned int sr)
{
    register int value = sr;

    asm
    {
        LDC value, SR
    };
}
```

Assembler Directives

At the time of this writing, there are two directives specific to Dreamcast assembler.

.set

Prototype `.set [reorder | noreorder]`

If you use the `reorder` option, the assembler uses *instruction scheduling* to improve performance. This optimization reorders processor instructions so that the execution of one instruction doesn't delay the execution of others.

The optimization level determines the default setting of `.set`. At optimization levels of 0 and 1, the default is `.set noreorder`. At other optimization levels, the default is `.set reorder`. For more information on setting your optimization level, see [“Global Optimizations” on page 58](#).

The example shown in [Listing 11.6](#) computes $x + y$ in the delay-slot for the call to `foo()`. Because we are purposefully putting the `ADD` instruction after the `JSR` instruction, we use `.set noreorder` to tell the compiler not to change our instruction sequence.

Listing 11.6 .set example

```
asm int ADD (int x, int y)
{
    .set    noreorder
    // y = x + y
    // call foo
    MOV.L   foo, R0;
    JSR     @R0;
    // return x + y;
    ADD     R4, R5;
    MOV     R5, R0;
}
```

.frame

Prototype `.frame`

The `.frame` directive generates the epilogue and prologue for the creation of a stack frame. You could create the stack frame yourself using inline assembly instructions, but using `.frame` is easier. You must create a stack frame if the function:

- calls other functions
- declares local variables

Listing shows the syntax of `.frame`. Note that we have commented out the RTS instruction. If you use `.frame`, the compiler generates the RTS automatically.

Listing 11.7 .frame example

```
asm int foo()  
{  
    .frame  
    MOV 12, R0;  
    // RTS;  
    ADD 1, R0;  
}
```

Intrinsic Functions

The compiler provides intrinsic functions that can generate inline assembly instructions. These intrinsic functions execute faster than other functions, because the compiler translates them into inline assembly instructions. Rather than using inline assembly syntax and specifying opcodes in an `asm` block, you may find it more convenient to call an intrinsic functions that matches what you want to do.

NOTE: Support for intrinsic functions is not part of the ANSI C or C++ standards. They are an extension provided by the CodeWarrior compiler.

When the compiler encounters the intrinsic function in your source code, it immediately substitutes the assembly instruction or instructions that match your function call. As a result, no actual function call occurs in the final object code. The final code contains the assembly language instructions that correspond to the intrinsic functions.

The topics in this section are:

- [List of Intrinsic Functions](#)

List of Intrinsic Functions

The compiler has support for the following intrinsic functions:

- [__abs](#)
- [__labs](#)
- [__fabs](#)
- [__fsqrt](#)
- [__alloca](#)
- [__memcpy](#)

Function	<code>__abs</code>
Description	Intrinsic for absolute value
Example	<pre>int Intrinsic_abs (int i) { int j; j = __abs(i); return j; }</pre>

Inline Assembler and Intrinsics for Dreamcast

Intrinsic Functions

Function `__labs`

Description Intrinsic for long absolute value

Example

```
long Intrinsic_labs (long i)
{
    long j;
    j = __labs(i);
    return j;
}
```

Function `__fabs`

Description Intrinsic for floating point absolute value

Example

```
double Intrinsic_fabs (double i)
{
    double j;
    j = __fabs(i);
    return j;
}
```

Function `__fsqrt`

Description Intrinsic for square root

Example

```
float Intrinsic_fsqrt (float i)
{
    float j;
    j = __fsqrt(i);
    return j;
}
```

Function `__alloca`

Description Intrinsic for dynamic stack allocation

Example

```
void Intrinsic_alloca(void)
{
    int i;
    short *x = (short
*)__alloca(1024*sizeof(short));
    for (i = 0; i < 1024; i++) x[i] = i;
}
```

Function `__memcpy`

Description Intrinsic for memory copy

Example

```
typedef struct s
{
    int i1;
    int i2;
    int i3;
}
s;

s s1;
s s2;

void Intrinsic_memcpy(s si)
{
    s2 = si;
    __memcpy(&s1, &si, sizeof(s));
}
```

Mnemonics for Inline Assembly

The instructions for inline assembly are a little bit different than those for regular assembly.

- [Special Instructions for Inline Assembly](#)

- [Complete List of Inline Assembly Mnemonics](#)

Special Instructions for Inline Assembly

These are special instructions for inline assembly. The following instructions are expanded by the compiler into a sequence of machine instructions. They are presented in the form:

"mnemonic", "format"

Move a constant into Rn.

"MOV.L", "w,Rn"

Load effective address of label

"MOVA", "l,=R0"

Load from constant pool

"MOV.L", "l,Rn"

Inline assembly directive

"_set", ""

"_unset", ""

Embedding Data Within Code Streams

Use the following inline instructions to embed data within code streams.

".data.b" "u"

".data.w" "v"

".data.l" "w"

Special Instructions Example

If you are unsure of how these instructions might be used, look at [Listing 11.8](#) for an example. Here, we use the special MOV.L instruction to load the constant 12345678 into R1.

Listing 11.8 Example of using special instructions

```
asm int fool() {  
    MOV.L    12345678,R1;  
    RTS;
```

```
    NOP;  
}
```

The compiler actually expands the special instruction into the machine instructions shown in [Listing 11.9](#).

Listing 11.9 Compiler expansion of the special instruction

```
    _foo4:  
0xD101      mov.l    @(4,pc),r1  
0x000B      rts  
0x0009      nop  
0x0000      .data.w  0x0000  
0x614E      .data.w  0x614E  
0x00BC      .data.w  0x00BC
```

If you do not use this special instruction, you become responsible for computing the displacement and alignment to access the constant that is embedded in the code. Without the special instruction, you would have to write code that resembles [Listing 11.10](#). Note that in the `MOV.L` instruction below, the displacement is multiplied by the compiler by a factor that is the same as the size of the data being accessed (in our case, this is 4 for a long).

Listing 11.10 Alternative to using the special instruction

```
asm int foo2() {  
    MOV.L @(1,PC), R0;  
    RTS;  
    NOP;  
    .data.w 0;  
    .data.l 12345678;  
}
```

Complete List of Inline Assembly Mnemonics

[Table 11.1](#) lists the inline assembly instructions supported by our compiler. They are similar to the regular assembler instructions, but

Inline Assembler and Intrinsics for Dreamcast

Mnemonics for Inline Assembly

'/' characters have changed to '_'. The instructions that we do not support in inline assembly are greyed out and marked as unsupported.

Table 11.1 **List of Inline Assembler Mnemonics**

Mnemonic	Format	Support
"ADD "	" i , Rn "	
"ADD "	" Rm , Rn "	
"ADDC "	" Rm , Rn "	
"ADDV "	" Rm , Rn "	
"AND "	" i , R0 "	
"AND "	" Rm , Rn "	
"AND .B "	" i , @(R0 , GBR) "	unsupported
"BF "	" l "	
"BF_S "	" l "	
"BRA "	" m "	
"BRAf "	" Rn "	
"BSR "	" m "	unsupported
"BSRF "	" Rn "	
"BT "	" l "	
"BT_S "	" l "	
"CLRMAC "	" "	
"CLRS "	" "	
"CLRT "	" "	
"CMP_EQ "	" i , R0 "	
"CMP_EQ "	" Rm , Rn "	
"CMP_GE "	" Rm , Rn "	

Inline Assembler and Intrinsics for Dreamcast

Mnemonics for Inline Assembly

Mnemonic	Format	Support
"CMP_GT "	"Rm , Rn "	
"CMP_HI "	"Rm , Rn "	
"CMP_HS "	"Rm , Rn "	
"CMP_PL "	"Rn "	
"CMP_PZ "	"Rn "	
"CMP_STR "	"Rm , Rn "	
"DIV0S "	"Rm , Rn "	
"DIV0U "	" "	
"DIV1 "	"Rm , Rn "	
"DMULS . L "	"Rm , Rn "	
"DMULU . L "	"Rm , Rn "	
"DT "	"Rn "	
"EXTS . B "	"Rm , Rn "	
"EXTS . W "	"Rm , Rn "	
"EXTU . B "	"Rm , Rn "	
"EXTU . W "	"Rm , Rn "	
"FABS "	"Fn "	
"FADD "	"Fm , Fn "	
"FCMP_EQ "	"Fm , Fn "	
"FCMP_GT "	"Fm , Fn "	
"FCNVDS "	"Fn "	
"FCNVSD "	"Fn "	
"FDIV "	"Fm , Fn "	
"FIPR "	"FVm , FVn "	unsupported

Inline Assembler and Intrinsics for Dreamcast

Mnemonics for Inline Assembly

Mnemonic	Format	Support
"FLDI0 "	"Fn "	
"FLDI1 "	"Fn "	
"FLDS "	"Fn "	
"FLOAT "	"Fn "	
"FMAC "	"F0 , Fm , Fn "	
"FMOV "	"Fm , Fn "	
"FMOV . S "	"Fm , @Rn "	
"FMOV . S "	"@Rm , Fn "	
"FMOV . S "	"@Rm+ , Fn "	
"FMOV . S "	"Fm , @-Rn "	
"FMOV . S "	"@ (R0 , Rm) , Fn "	
"FMOV . S "	"Fm , @ (R0 , Rn) "	
"FMOV "	"Xm , @Rn "	unsupported
"FMOV "	"@Rm , Xn "	unsupported
"FMOV "	"@Rm+ , Xn "	unsupported
"FMOV "	"Xm , @-Rn "	unsupported
"FMOV "	"@ (R0 , Rm) , Xn "	unsupported
"FMOV "	"Xm , @ (R0 , Rn) "	unsupported
"FMOV "	"Xm , Xn "	unsupported
"FMOV "	"Xm , Dn "	unsupported
"FMOV "	"Dm , Xn "	unsupported
"FMUL "	"Fm , Fn "	
"FNEG "	"Fn "	
"FRCHG "	" "	

Inline Assembler and Intrinsics for Dreamcast

Mnemonics for Inline Assembly

Mnemonic	Format	Support
"FSCHG "	" "	
"FSQRT "	"Fn "	
"FSTS "	"Fn "	
"FSUB "	"Fm , Fn "	
"FTRC "	"Fn "	
"FTRV "	"XM , FVn "	unsupported
"JMP "	"@Rn "	
"JSR "	"@Rn "	unsupported
"LDC "	"Rn , GBR "	unsupported
"LDC "	"Rn , SR "	
"LDC "	"Rn , VBR "	
"LDC "	"Rn , SSR "	
"LDC "	"Rn , SPC "	
"LDC "	"Rn , DBR "	
"LDC "	"Rn , Rb "	unsupported
"LDC . L "	"@Rn+ , GBR "	unsupported
"LDC . L "	"@Rn+ , SR "	
"LDC . L "	"@Rn+ , VBR "	
"LDC . L "	"@Rn+ , SSR "	
"LDC . L "	"@Rn+ , SPC "	
"LDC . L "	"@Rn+ , DBR "	
"LDC . L "	"@Rn+ , Rb "	unsupported
"LDS "	"Rn , FPSCR "	
"LDS "	"Rn , MACH "	

Inline Assembler and Intrinsics for Dreamcast

Mnemonics for Inline Assembly

Mnemonic	Format	Support
"LDS "	"Rn, MACL "	
"LDS "	"Rn, PR "	
"LDS "	"Rn, FPUL "	
"LDS .L "	"@Rn+, FPSCR "	
"LDS .L "	"@Rn+, MACH "	
"LDS .L "	"@Rn+, MACL "	
"LDS .L "	"@Rn+, PR "	
"LDS .L "	"@Rn+, FPUL "	
"LDTLB "	" "	
"MAC .L "	"@Rm+, @Rn+ "	
"MAC .W "	"@Rm+, @Rn+ "	
"MOV "	"i, Rn "	
"MOV "	"Rm, Rn "	
"MOV .B "	"@(d8, GBR), R0 "	unsupported
"MOV .B "	"@(d4, Rm), R0 "	
"MOV .B "	"@(R0, Rm), Rn "	
"MOV .B "	"@Rm+, Rn "	
"MOV .B "	"@Rm, Rn "	
"MOV .B "	"R0, @(d8, GBR) "	unsupported
"MOV .B "	"R0, @(d4, Rm) "	
"MOV .B "	"Rm, @(R0, Rn) "	
"MOV .B "	"Rm, @-Rn "	
"MOV .B "	"Rm, @Rn "	
"MOV .W "	"@(d8, GBR), R0 "	unsupported

Inline Assembler and Intrinsics for Dreamcast

Mnemonics for Inline Assembly

Mnemonic	Format	Support
"MOV.W"	"@(d8,PC),Rn"	unsupported
"MOV.W"	"@(d4,Rm),R0"	
"MOV.W"	"@(R0,Rm),Rn"	
"MOV.W"	"@Rm+,Rn"	
"MOV.W"	"@Rm,Rn"	
"MOV.W"	"R0,@(d8,GBR)"	unsupported
"MOV.W"	"R0,@(d4,Rm)"	
"MOV.W"	"Rm,@(R0,Rn)"	
"MOV.W"	"Rm,@-Rn"	
"MOV.W"	"Rm,@Rn"	
"MOV.L"	"@(d8,GBR),R0"	unsupported
"MOV.L"	"@(d8,PC),Rn"	
"MOV.L"	"@(d4,Rm),Rn"	
"MOV.L"	"@(R0,Rm),Rn"	
"MOV.L"	"@Rm+,Rn"	
"MOV.L"	"@Rm,Rn"	
"MOV.L"	"R0,@(d8,GBR)"	unsupported
"MOV.L"	"Rm,@(d4,Rn)"	
"MOV.L"	"Rm,@(R0,Rn)"	
"MOV.L"	"Rm,@-Rn"	
"MOV.L"	"Rm,@Rn"	
"MOVA"	"@(d8,PC),R0"	
"MOVA"	<label>,R0	
"MOVCA.L"	"@R0,@Rn"	

Inline Assembler and Intrinsics for Dreamcast

Mnemonics for Inline Assembly

Mnemonic	Format	Support
"MOV _T "	"R _n "	
"MUL _{.L} "	"R _m , R _n "	
"MUL _{S.W} "	"R _m , R _n "	
"MUL _{U.W} "	"R _m , R _n "	
"NEG"	"R _m , R _n "	
"NEGC"	"R _m , R _n "	
"NOP"	" "	
"NOT"	"R _m , R _n "	
"OCBI"	"@R _n "	
"OCBP"	"@R _n "	
"OCBWB"	"@R _n "	
"OR"	"i, R ₀ "	
"OR"	"R _m , R _n "	
"OR _{.B} "	"i, @(R ₀ , GBR)"	unsupported
"PREF"	"@R _n "	
"ROTCL"	"R _n "	
"ROTCR"	"R _n "	
"ROTL"	"R _n "	
"ROTR"	"R _n "	
"RTE"	" "	
"RTS"	" "	
"SETS"	" "	
"SETT"	" "	
"SHAD"	"R _m , R _n "	

Inline Assembler and Intrinsics for Dreamcast

Mnemonics for Inline Assembly

Mnemonic	Format	Support
"SHAL "	"Rn "	
"SHAR "	"Rn "	
"SHLD "	"Rm , Rn "	
"SHLL "	"Rn "	
"SHLL2 "	"Rn "	
"SHLL8 "	"Rn "	
"SHLL16 "	"Rn "	
"SHLR "	"Rn "	
"SHLR2 "	"Rn "	
"SHLR8 "	"Rn "	
"SHLR16 "	"Rn "	
"SLEEP "	" "	
"STC "	"GBR , =Rn "	unsupported
"STC "	"SR , =Rn "	
"STC "	"VBR , =Rn "	
"STC "	"SSR , =Rn "	
"STC "	"SPC , =Rn "	
"STC "	"DBR , =Rn "	
"STC "	"Rb , =Rn "	unsupported
"STC . L "	"G , @Rn+ "	unsupported
"STC . L "	"SR , @Rn+ "	
"STC . L "	"VBR , @Rn+ "	
"STC . L "	"SSR , @Rn+ "	
"STC . L "	"SPC , @Rn+ "	

Inline Assembler and Intrinsics for Dreamcast

Mnemonics for Inline Assembly

Mnemonic	Format	Support
"STC.L"	"DBR,@Rn+"	unsupported
"STC.L"	"Rb,@Rn+"	
"STS"	"FPSCR,Rn"	
"STS"	"MACH,Rn"	
"STS"	"MACL,Rn"	
"STS"	"PR,Rn"	
"STS"	"FPUL,Rn"	
"STS.L"	"FPSCR,@-Rn"	
"STS.L"	"MACH,@-Rn"	
"STS.L"	"MACL,@-Rn"	
"STS.L"	"PR,@-Rn"	
"STS.L"	"FPUL,@-Rn"	
"SUB"	"Rm,Rn"	
"SUBC"	"Rm,Rn"	
"SUBV"	"Rm,Rn"	
"SWAP.B"	"Rm,Rn"	
"SWAP.W"	"Rm,Rn"	
"TAS.B"	"@Rn"	
"TRAPA"	"i"	
"TST"	"i,R0"	
"TST"	"Rm,Rn"	
"TST.B"	"i,@(R0,GBR)"	
"XOR"	"i,R0"	
"XOR"	"Rm,Rn"	

Inline Assembler and Intrinsics for Dreamcast

Mnemonics for Inline Assembly

Mnemonic	Format	Support
"XOR.B"	"i,@(R0,GBR)"	unsupported
"XTRCT"	"Rm,Rn"	



Libraries and Runtime Code for Dreamcast

Metrowerks provides a variety of libraries for use with the CodeWarrior development environment. They include ANSI-standard libraries for C and C++, as well as runtime libraries and other code. This chapter discusses how to use these libraries for Dreamcast development.

The sections in this chapter are:

- [Runtime Libraries for Dreamcast](#)
- [Allocating Memory and Heaps for Dreamcast](#)

Runtime Libraries for Dreamcast

You may need to include the following runtime libraries in your project.

The following are the same runtime libraries that ship with the Dreamcast SDK, but they have been converted for use with CodeWarrior:

```
'nindows.elf.lib'  
'ninja.elf.lib'  
'shinobi.elf.lib'  
'sh4nlfzz.elf.lib'
```

The following runtime library is required by CodeWarrior:

```
'MSLRuntimeDC.lib'
```

The following library is required to use C++ standard libraries:

```
'MSLCppDC.lib'
```

The following library is required for using `printf()` functions:

`'mw output.lib'`

Allocating Memory and Heaps for Dreamcast

Please note that the heap and stack size are specified by the Dreamcast SDK libraries. You cannot specify heap or stack from the [SH Linker](#) settings panel.



Troubleshooting for Dreamcast

This chapter gives you a quick reference point for common problems (and their solutions) when using CodeWarrior for Dreamcast development. This should be the first place you look before contacting CodeWarrior support.

- [Hardware Communications](#)
- [Compiler Problems](#)
- [Debugger Problems](#)

Hardware Communications

This section describes possible solutions to communications problems between your host computer and your HKT-01.

CodeWarrior fails to recognize the HKT-01 hardware.

Problem: CodeWarrior can't communicate with the HKT-01.

Background: The HKT-01 is a SCSI device. If the HKT-01 is not turned on when the operating system starts, it will not be recognized.

Solution: Turn on the HKT-01 and reboot your computer.

Codescape asks you to update your SCSI driver.

Problem: Your SCSI driver is too old for Codescape to use.

Background: Codescape needs the latest version of the Adaptec SCSI driver.

Solution: Download the latest version of the SCSI driver from the Adaptec website: <http://www.adaptec.com>

Compiler Problems

This section provides possible solutions to problems you may encounter in using the compiler.

Error '@5' could not be assigned to a register

Problem: The compiler is rejecting your inline assembly statements when your global optimization setting is set to 0.

Background: The compiler does not use the virtual register allocator at optimization level 0. Therefore, it is possible that when the inline assembly routines are compiled, there are no more real registers available. .

Solution: You can set inlining to **Don't Inline** in the C/C++ language settings, or you can set the optimization level to Level 1 or higher.

Debugger Problems

This section provides possible solutions to problems you may encounter during debugging.

Programs with GDROM data files do not run

Problem: The debugger cannot find your data files.

Background: Data files that are meant to be spooled from the GDROM are loaded via GD Workshop, not the CodeWarrior debugger.

Solution: Use GD Workshop to emulate the GDROM device.