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 on the CodeWarrior CD for the latest up-to-date information.

Revised: 000301 rw

How to Contact Metrowerks:

U.S.A. and International	Metrowerks Corporation 9801 Metric Blvd., Suite #100 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite #300 Ville St-Laurent, QC Canada H4L 5G6
World Wide Web	`http://www.metrowerks.com`
Registration Information	`http://www.metrowerks.com/register` `mailto:register@metrowerks.com`
Desktop Technical Support	`http://www.metrowerks.com/support/desktop/` `mailto:cw_support@metrowerks.com`
Embedded Technical Support	`http://www.metrowerks.com/support/embedded/` `mailto:cw_emb_support@metrowerks.com`
Sales, Marketing, & Licensing	`mailto:sales@metrowerks.com`
Ordering	Voice: (800) 377–5416 Fax: (512) 873–4901
Intl. Sales, Mkt & Licensing	`mailto:intlsls@metrowerks.com`
International Ordering	Voice: +1 512 873 4724 Fax: +1 512 873 4901

Table of Contents

1

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
- What's New in This Release —new features since the previous release
- 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.

What's New in This Release

The CodeWarrior product for Dreamcast development has some new features. The following are most notable.

Flexible Linker Command File Language

The CodeWarrior linker uses a new command file language to arrange your code and data. The new linker command file format allows more flexibility in positioning than the previous version.

Complete SHC Intrinsics Support

It is now possible to use all of Hitachi's SHC compiler intrinsics in your CodeWarrior projects. Every intrinsic is supported.

More Inline Assembly Instructions

The new CodeWarrior compiler supports more inline assembly instructions and parameters than before. The additional instructions include `AND.B`, `STC`, `STC.L`, `LDC`, `LDC.L`, `MOV.B`, `MOV.W`, `MOV.L`, `OR.B`, `TST.B`, and `XOR.B`.

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	*Core Tutorials*
The Dreamcast Tools Target Settings for Dreamcast	*IDE User Guide*
"Debugging For Dreamcast"	*Debugger User Guide*
C and C++ for Dreamcast	*C Compilers Reference*

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 10 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
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 C/C++ compiler for Dreamcast development.
ELF Linker and Command Language	explores the linker and its command file syntax

Chapter	Description
Linker Issues for Dreamcast	examines Dreamcast specific linker issues
Inline Assembler and Intrinsics for Dreamcast	details support for inline assembly and instrinsic functions
Overlays	how to create and debug overlays
Libraries and Runtime Code for Dreamcast	libraries provided with CodeWarrior for Dreamcast
Command Line Tools	how to use command line tools
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.

2

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.
- Sega Dreamcast SDK 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 16.

2. Install the Dreamcast libraries

   For complete details, see "Installing the Dreamcast Runtime Library" on page 17.

3. Test your system.

   Before you begin programming, see "Making Sure Your Dreamcast Development System Works" on page 17.

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. When you finish the 95/98 installation, 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 Sega Dreamcast SDK libraries are used in almost every Dreamcast project you develop.

In this release, we have included CodeWarrior-compatible versions of the Sega Dreamcast SDK libraries in the folder named "`Dreamcast Support`". They are automatically 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 sample that is included in the CodeWarrior example files.

1. **Launch the CodeWarrior IDE**

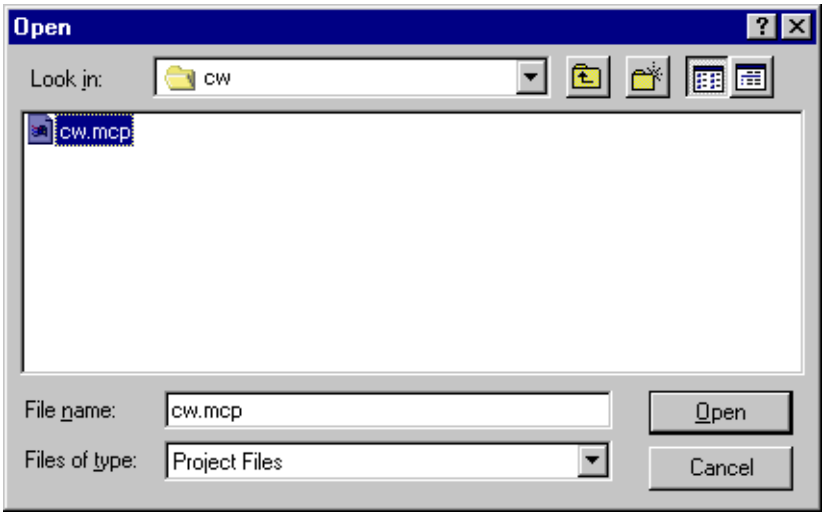
   Locate the icon for the CodeWarrior IDE, and launch the application.

2. **Open the project.**

   From the **File** menu, choose the **Open** item. The dialog box in Figure 2.1 appears.

   Locate the project `Dreamcast Examples/SDK 1.56j/sample3d/teapot/cw/cw.mcp`.

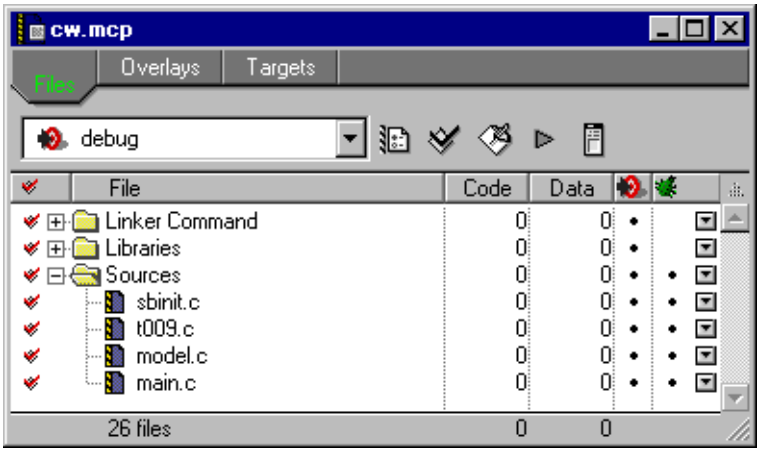
Figure 2.1 The 'open' dialog box



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



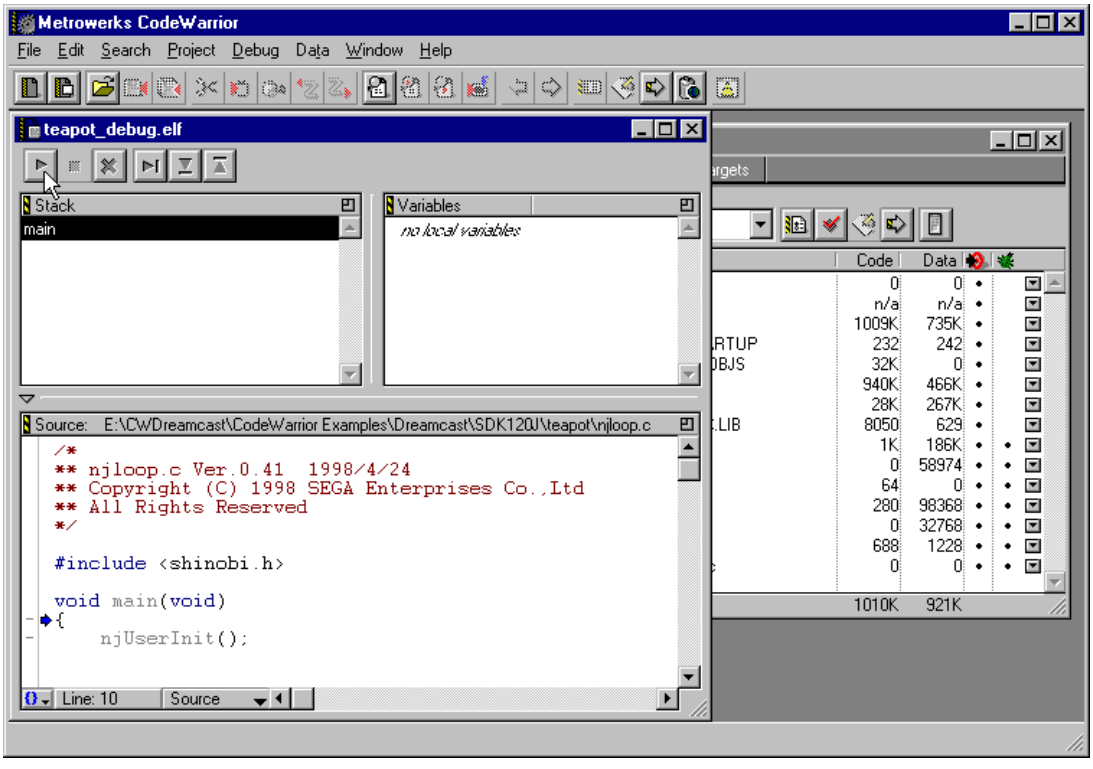
3. **Build the project.**

   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.

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

3

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

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 24. 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 generated Dreamcast applications and libraries.

For more information on the Compiler Settings, see "Target Settings for Dreamcast" on page 51. 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 51.

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 seperate from the CodeWarrior IDE.

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.

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.

4

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:

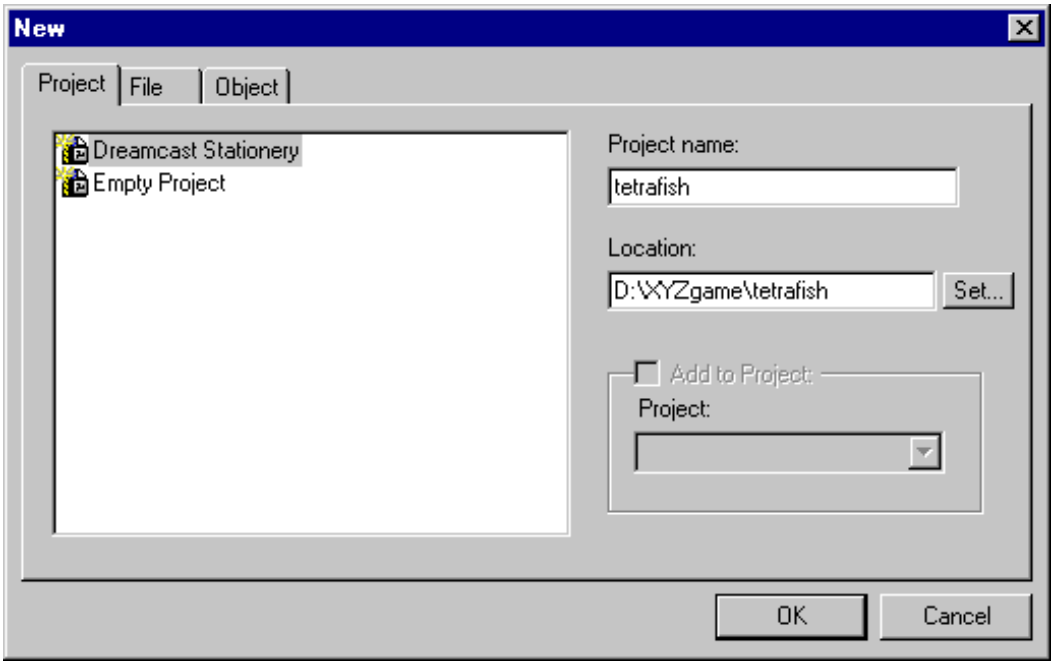
Creating an Application

To create a Dreamcast application, perform the following steps:

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

   Choose the **File > New Project**. The CodeWarrior IDE displays the **New** dialog box as seen in Figure 4.1 . Give your project a name and location. In this example, our project name is `tetrafish`, and its location is `D:\XYZgame\tetrafish`.

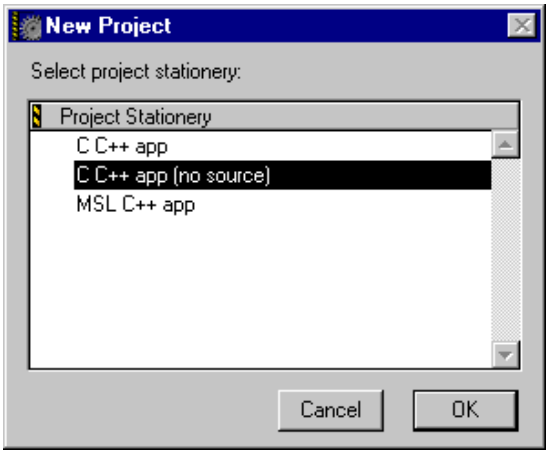
Figure 4.1 The New dialog box



2. Select your project stationery.

Click on the line containing the Dreamcast stationery you want, then click OK. For this example, select C/C++ App (no source) (Figure 4.2).

Figure 4.2 Stationery selection dialog box

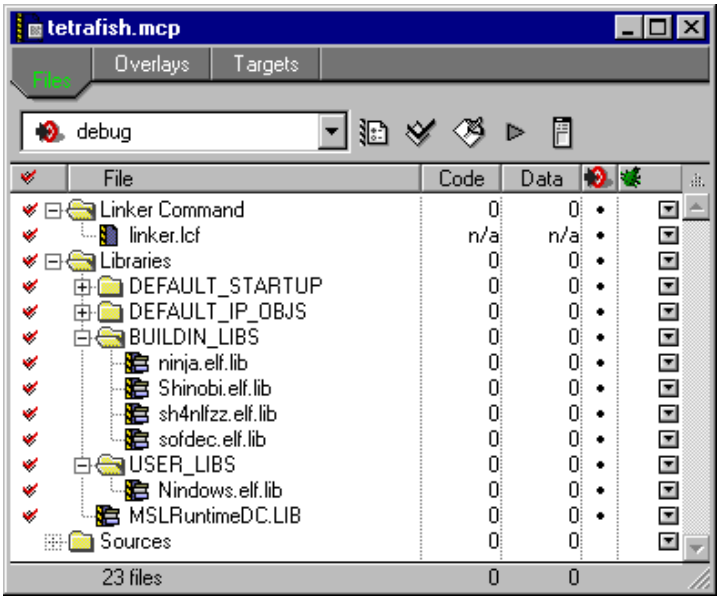


3. Examine the project window contents

When you clicked **OK**, the CodeWarrior IDE created a new project file in the designated directory, with the conventional extension .mcp.

The project window you see on your screen contains the Sega SDK libraries and an empty place for your program's source files. It should resemble the window shown in Figure 4.3

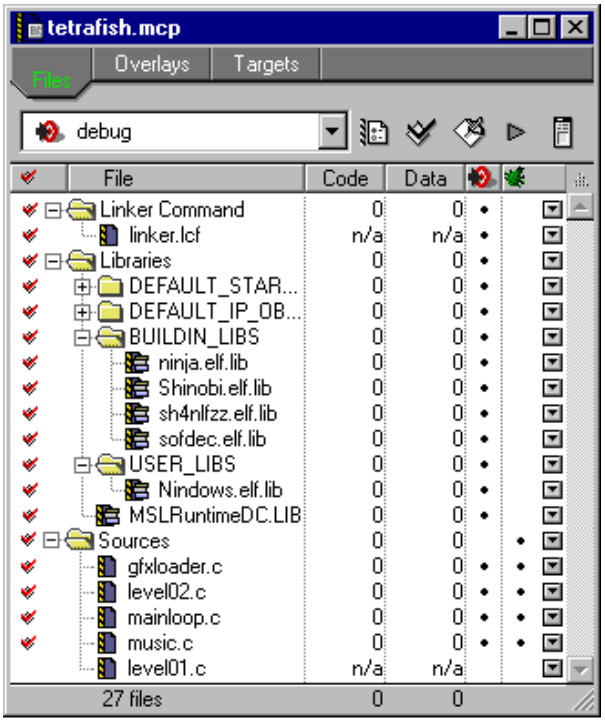
Figure 4.3 Project window



4. Modify the contents of the new project.

You will want to add your own source files to your new project. Figure 4.4 shows the project window with some source files added.

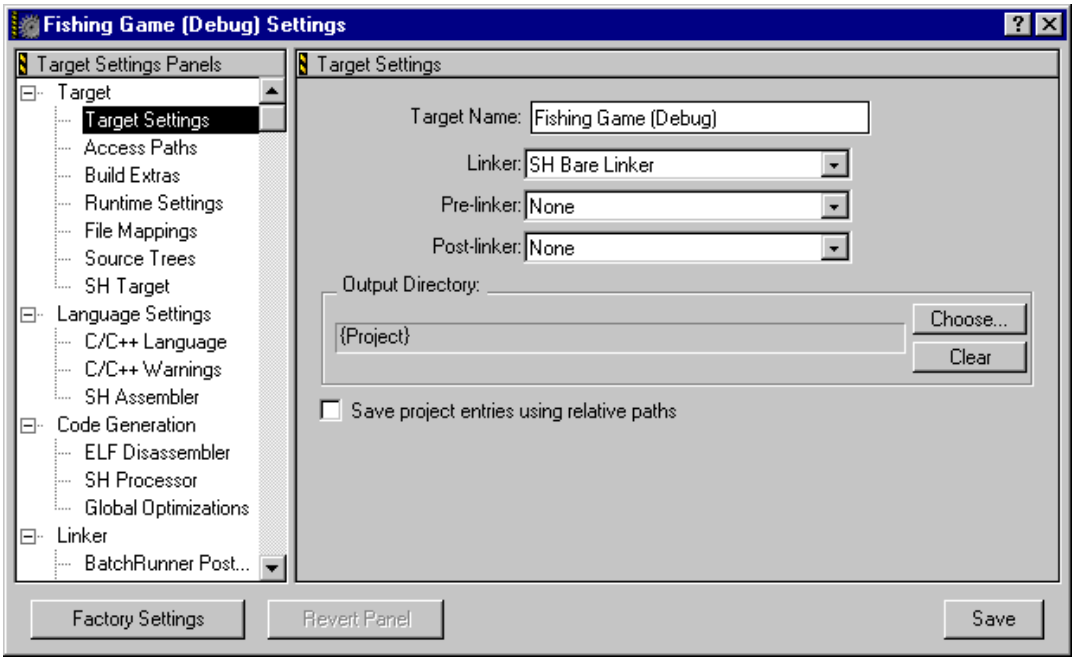
Figure 4.4 Project window with modifications



5. 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 4.4, for example, the name of the command would be **debug Settings**).

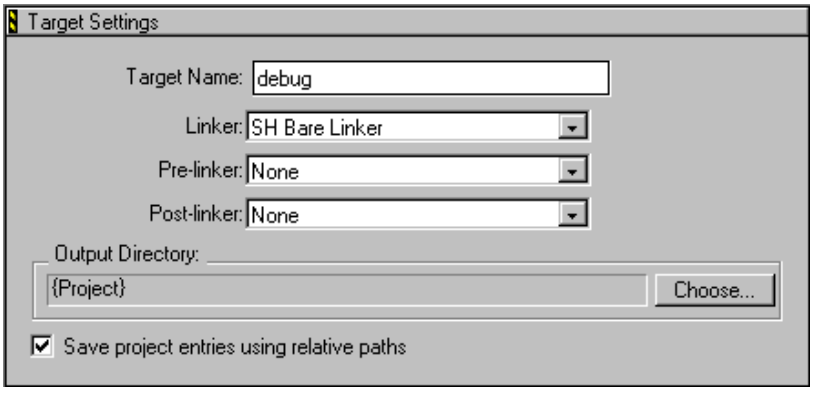
Figure 4.5 Target settings dialog box



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 4.5 .

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 4.6 Target Settings panel



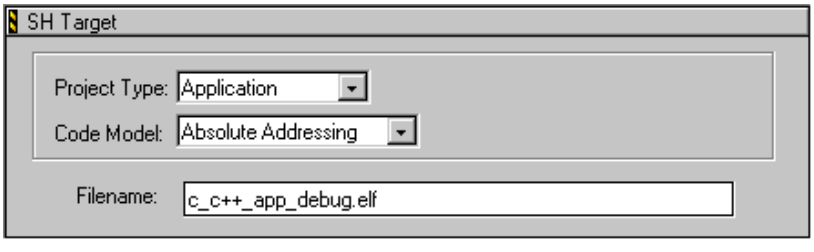
6. Specify target settings.

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 4.6 . 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.

7. Set the project type.

Click **SH Target** in the panel list to display the settings panel shown in Figure 4.7 . 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 4.7 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 51 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.

5

Creating Static Libraries

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

Topics in this chapter are:

See also "Creating Applications" on page 27 for information on creating executable applications. For more information on projects in general, see the *IDE User Guide*.

About Static Libraries

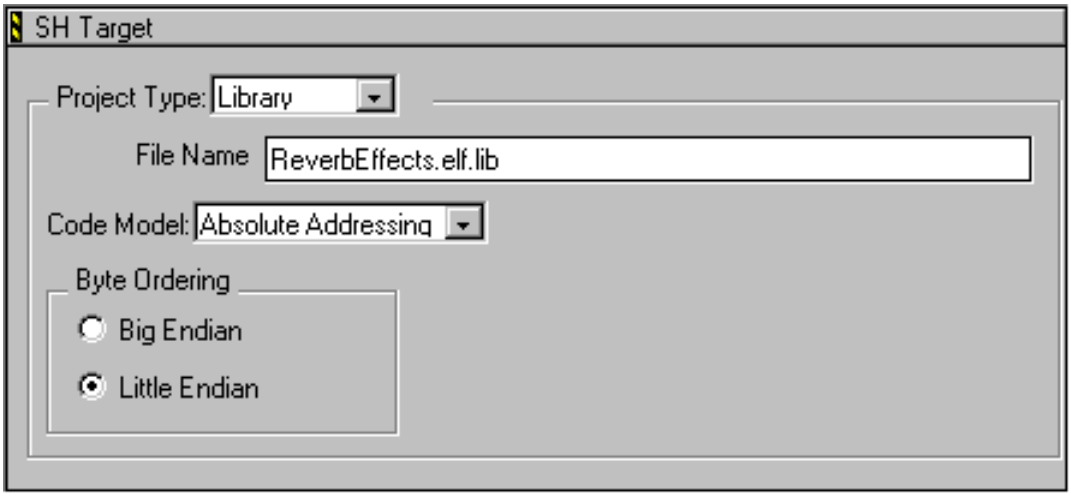
A *static library* is a collection of functions and data that can be incorporated into an application program (or another library). You can use predefined libraries supplied with CodeWarrior, and you can create your own custom-designed libraries for use in your own projects.

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**.

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 27 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 51 as well as the relevant sections of the *IDE User Guide* and the *C Compilers Reference.*

6

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

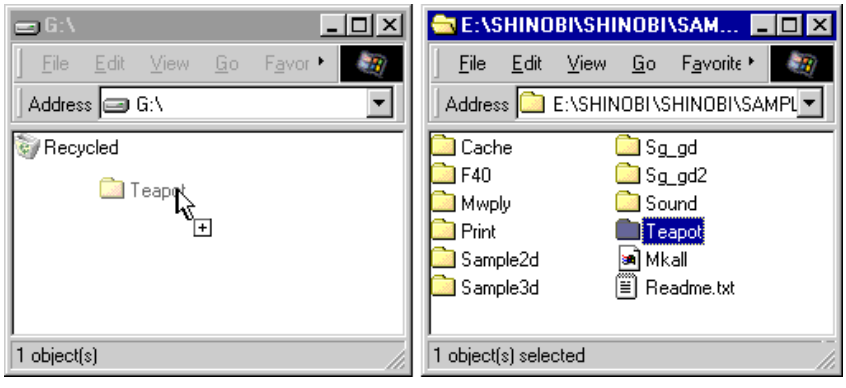
Steps for Converting SH Projects

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:\`.

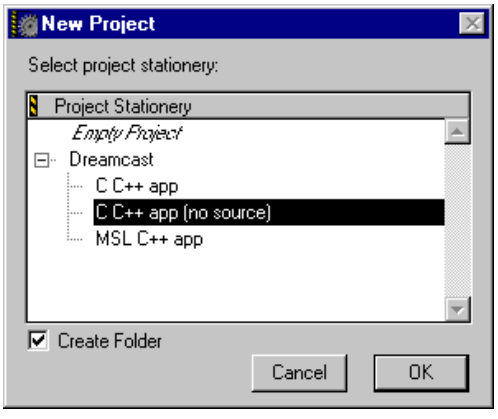
Figure 6.1 Copying teapot files to a new folder



2. Create a new project.

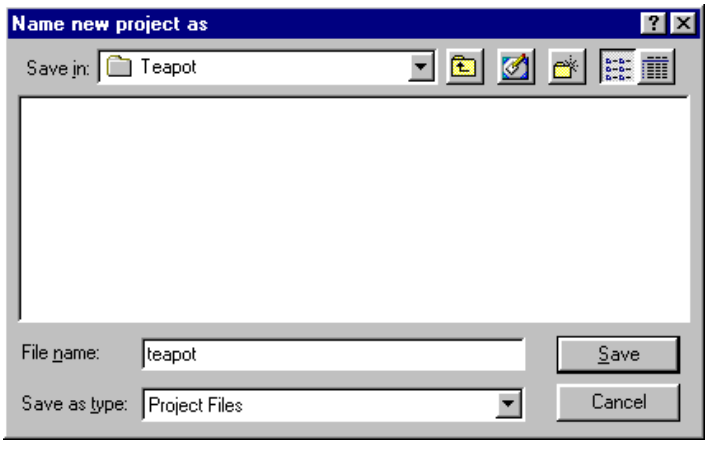
In CodeWarrior, choose **New Project** from the **File** menu. From the New Project window, select the Dreamcast **C app (no source)** stationery as shown in Figure 6.2, and click **OK**.

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`.

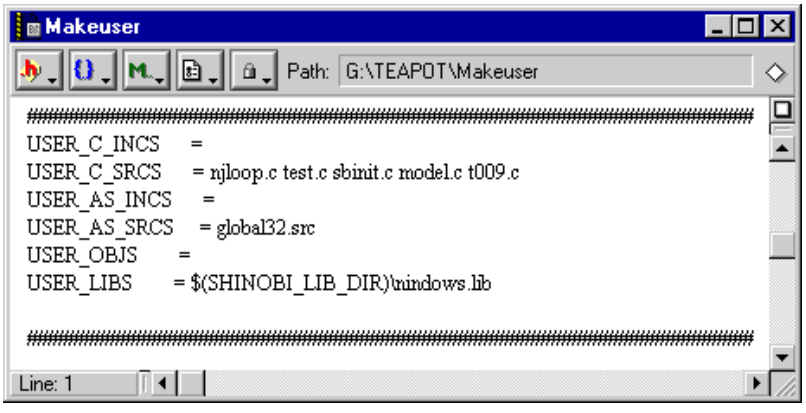
Figure 6.3 Start the new project in the teapot folder



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.

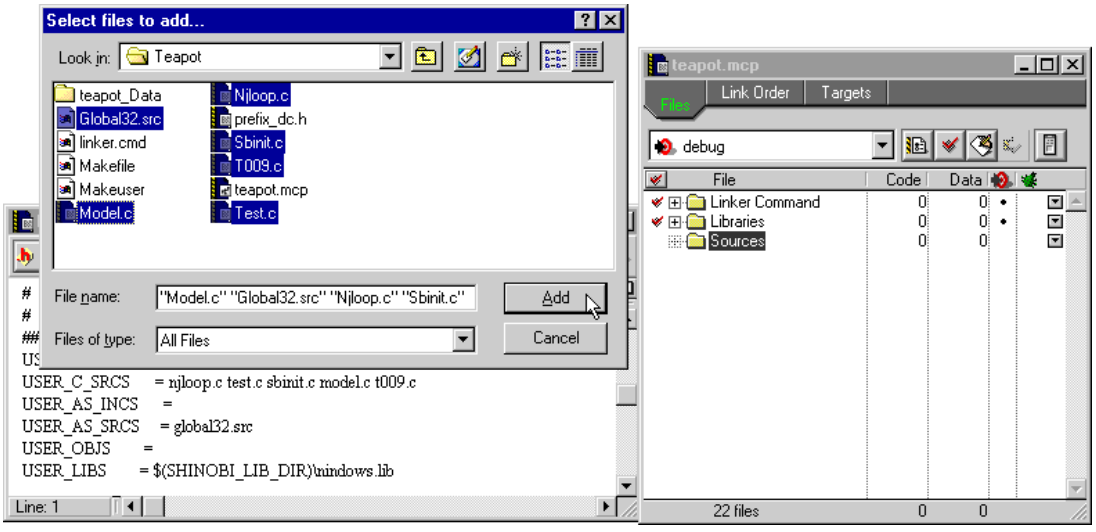
Figure 6.4 Finding source files in Makeuser



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.

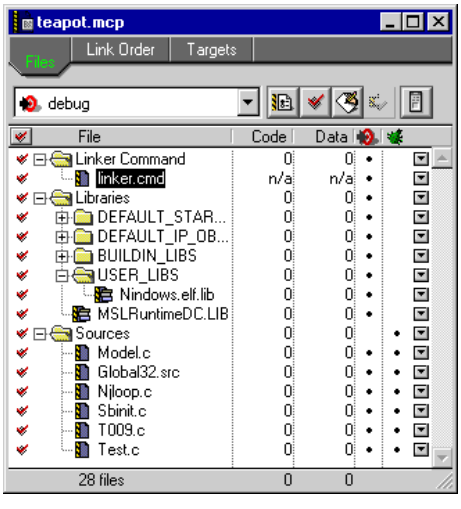
Figure 6.5 Adding source files to the project window



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.

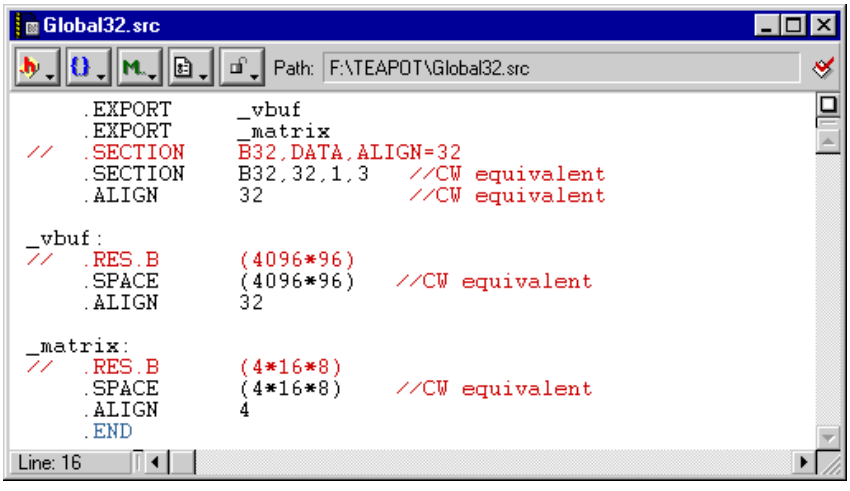
Figure 6.6 All files have been added



4. Convert assembler files.

Before teapot will compile on CodeWarrior, we must make a few changes to the assembly source file, `global32.src`, shown in Figure 6.7. The assembler directives in the `global32.src` file control the behaviour of the Hitachi assembler. The CodeWarrior assembler uses slightly different directives, so we have to replace the Hitachi assembler directives with the 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`.

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.

7

Debugging For Dreamcast

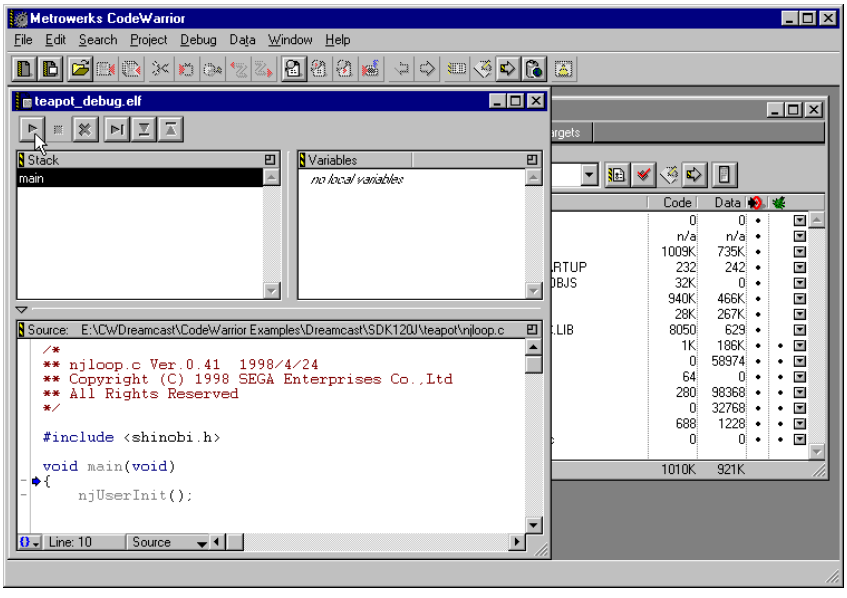
This chapter discusses how to use CodeWarrior to debug Dreamcast code. It covers those aspects of debugging that are specific to the Dreamcast platform or are different from the processes described in the *IDE User Guide* and the *Debugger User Guide.*

This chapter contains the following topics:

Debugging with CodeWarrior

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 `mw_pr()`

The `printf()` function does not work when you debug Dreamcast code. However, we provide an alternate function, `mw_pr(const char *)`, that will send a string to the console window.

To use `mw_pr()`, add the library `'mw output.lib'` to your project. This library is located in the `Dreamcast Support` folder.

`mw_pr()` takes a char pointer as its input. It recognizes `'\n'` as a newline character, and the largest string it will accept is 1024 bytes long. You might use it in the following way.

```
char *p = "Hello World!\n";
mw_pr(p);
```

Debugging Static Libraries

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

8

Debugging With Codescape

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

This chapter includes the following topics:

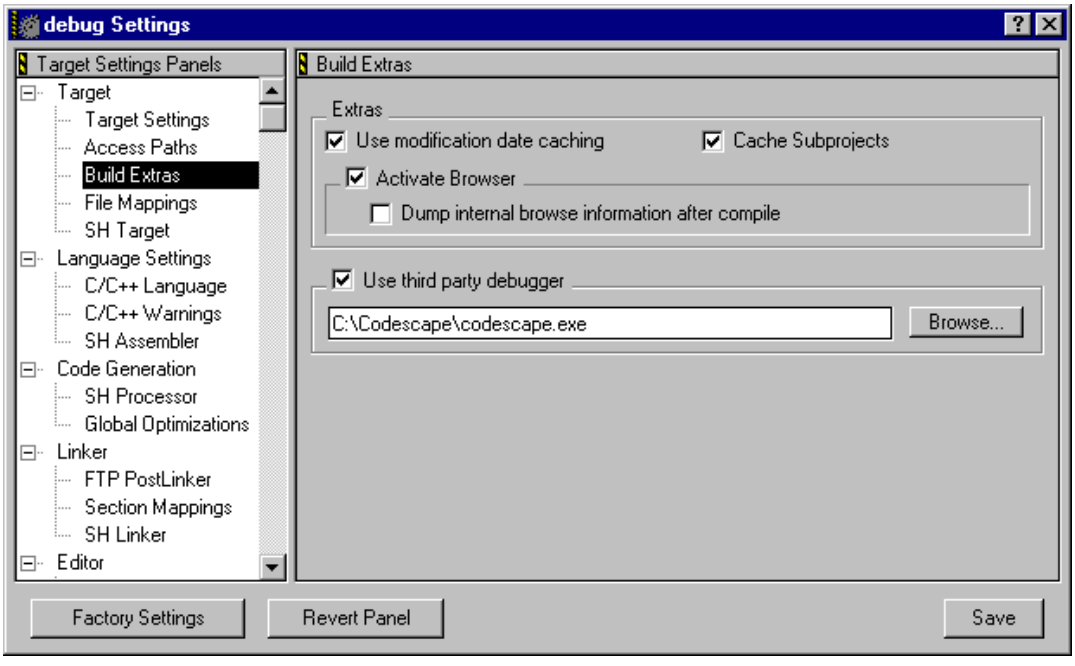
NOTE Please see the Debugger release notes for the latest news about our Codescape interoperability.

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.

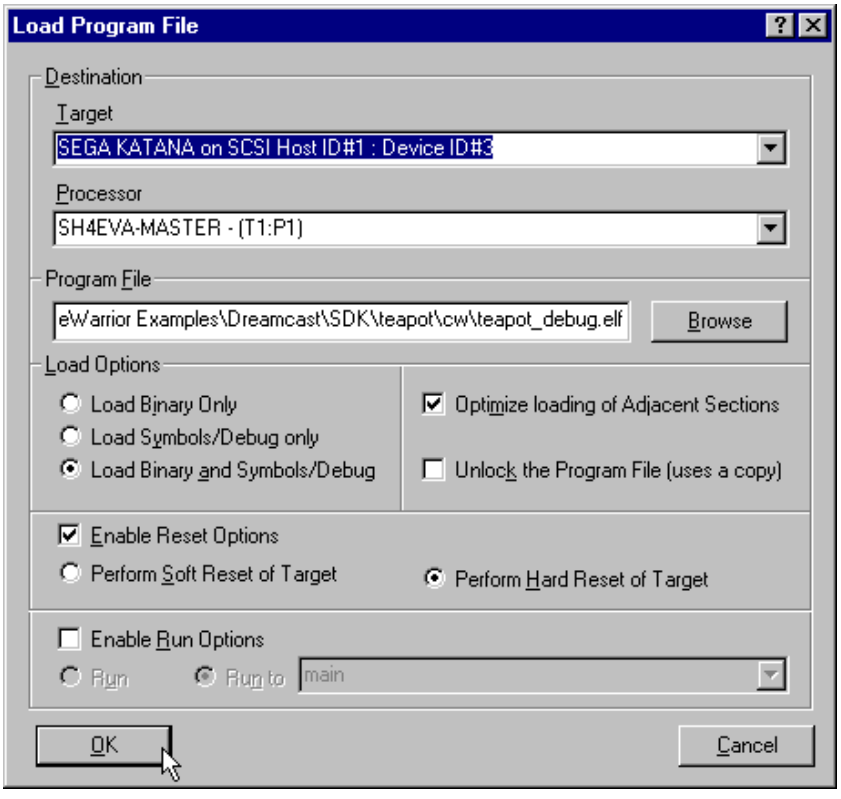
Figure 8.1 Set CodeScape to be your third-party debugger



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 does not work in the Codescape debugger. To print strings to the debugging console window, you must use the `LIBCRS` library provided by Cross Products. Please refer to their Codescape documentation for more information.

9

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:

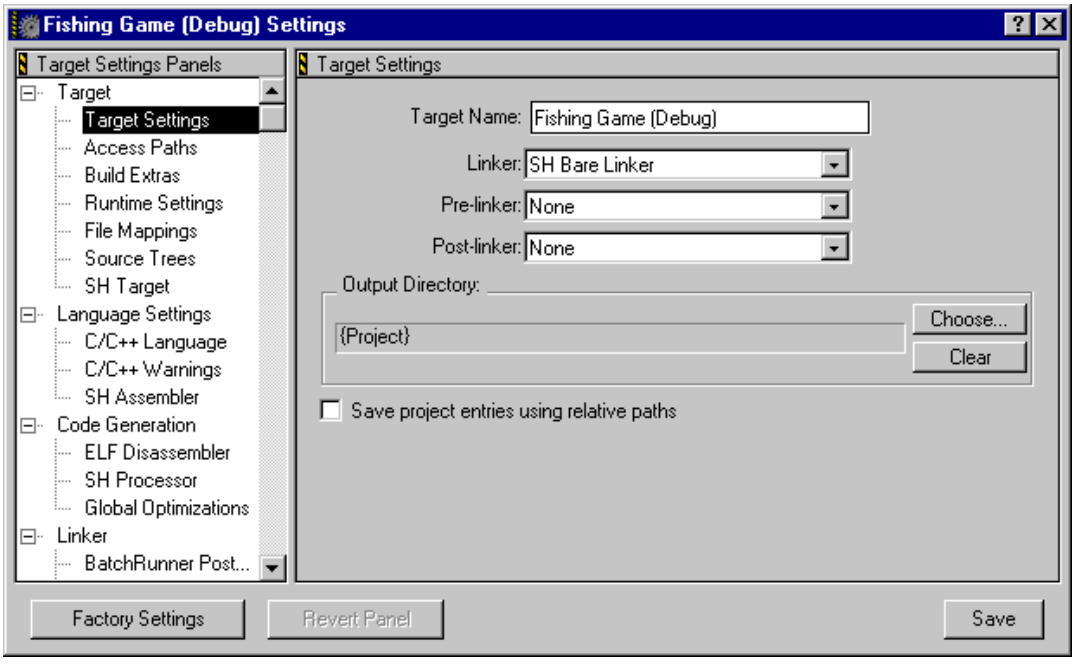
Target Settings Overview

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.

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
- ELF Disassembler
- SH Processor
- Global Optimizations
- BatchRunner PostLinker
- SH Linker

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	*IDE User Guide*
Build Extras	*IDE User Guide*
File Mappings	*IDE User Guide*
Custom Keywords	*IDE User Guide*
Debugger Settings	*IDE User Guide*
C/C++ Language	*C Compilers Reference*
C/C++ Warnings	*C Compilers Reference*

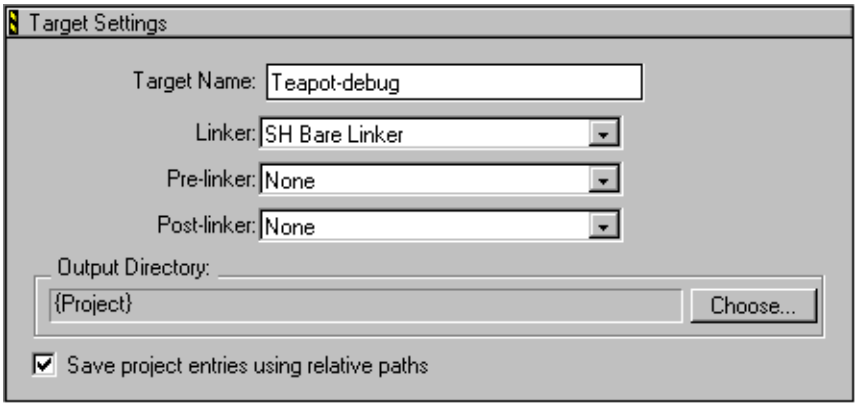
Target Settings

The Target Settings *dialog box* contains a Target Settings *panel*. The dialog box and the panel are not the same. The dialog box displays 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 combines **Access Path** settings with the path settings it includes for each project entry.

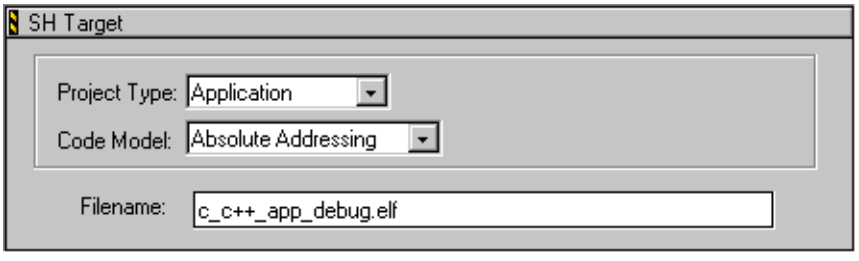
When this option is off, the IDE only records information about each project entry's file name. When searching for a file, the IDE only uses Access Paths.

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

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.

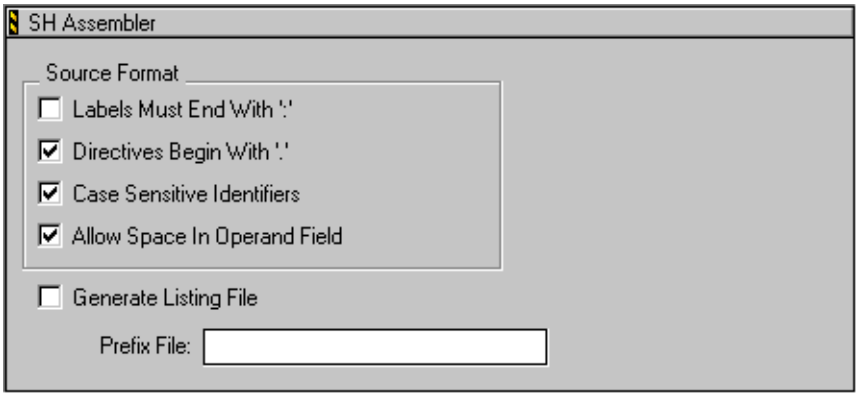
Code Model

For Dreamcast development, use Absolute addressing as the code model.

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.

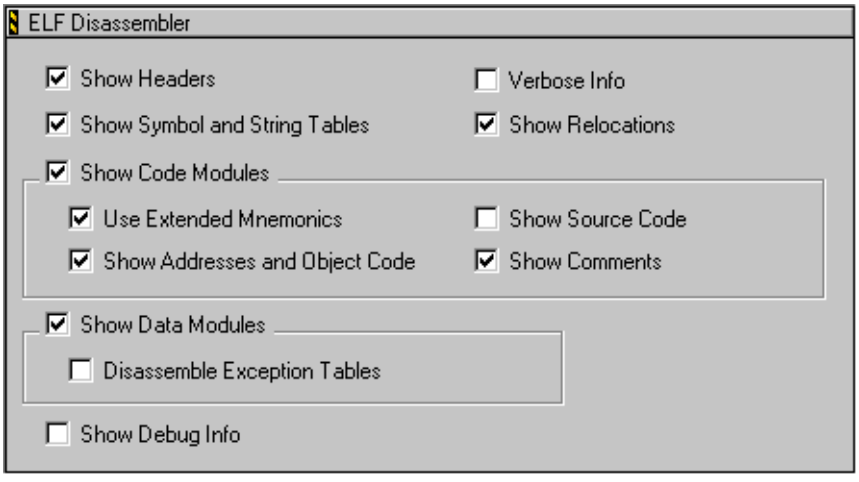
Prefix File

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.

ELF Disassembler

The **ELF Disassembler** panel, shown in Figure 9.6 , is where you control settings related to the disassembly view shown to you when you disassemble object files.

Figure 9.6 The ELF Disassembler panel



The items in this panel are:

Show Headers
Verbose Info
Show Symbol and String Tables
Show Relocations
Show Code Modules
Use Extended Mnemonics
Show Source Code
Show Address and Object Code
Show Comments
Show Data Modules
Disassemble Exception Tables
Show Debug Info

Show Headers

The **Show Headers** checkbox puts ELF header information into the disassembled output.

Verbose Info

The **Verbose Info** checkbox puts additional information into the disassembled output. For the `.symtab` section, some of the descriptive constants are shown with their numeric equivalents. .line, `.debug`, `extab` and `extabindex` sections are also shown with an unstructured hex dump.

Show Symbol and String Tables

The **Show Symbol and String Tables** checkbox lists the symbol table for the disassembled module.

Show Relocations

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

Show Code Modules

The **Show Code Modules** checkbox outputs the ELF code sections for the disassembled module.

Use Extended Mnemonics

The **Use Extended Mnemonics** checkbox lists the extended mnemonics for each instruction in the disassembled module.

Show Source Code

The **Show Source Code** checkbox lists the source code for the disassembled module. The source code is displayed in mixed mode, with line number information from the original C source.

This checkbox is only displayed if the **Show Code Modules** checkbox is enabled.

Show Address and Object Code

The **Show Address and Object Code** checkbox lists the address and object code for the disassembled module.

This checkbox is only displayed if the **Show Code Modules** checkbox is enabled.

Show Comments

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

This checkbox is only displayed if the **Show Code Modules** checkbox is enabled.

Show Data Modules

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

Disassemble Exception Tables

The **Disassemble Exception Tables** checkbox includes any C++ exception tables for the disassembled module.

This checkbox is only displayed if the **Show Data Modules** checkbox is enabled.

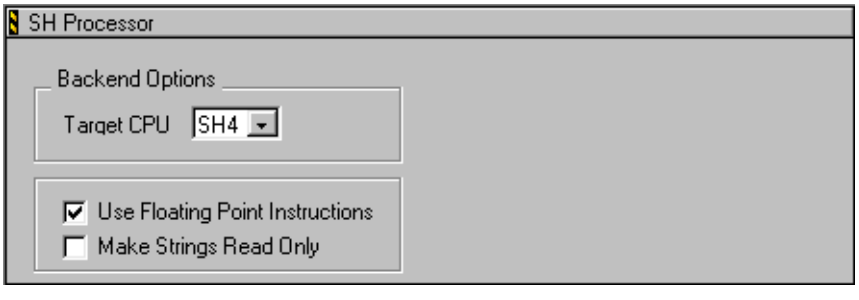
Show Debug Info

The **Show Debug Info** checkbox includes DWARF symbolics in the disassembled output.

SH Processor

The SH Processor panel, shown in Figure 9.7, is where you control settings related to code generation for the Dreamcast platform.

Figure 9.7 The SH Processor panel.



The items in this panel are:

Target CPU
Use Floating Point Instructions
Make Strings Read Only

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.

Make Strings Read Only

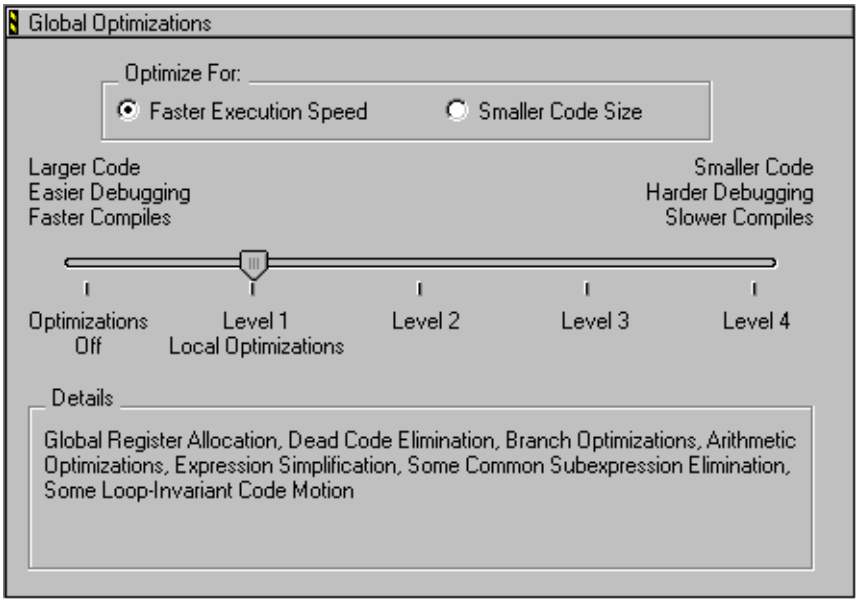
This option determines where character string constants are stored. If this option is off, the compiler stores string constants in the data section. If this option is on, the compiler stores string constants in the code section.

Variables that are not initialized to the address of another object at runtime are always placed in the code section, including C/C++ variables declared with the `const` storage-class modifier.

Global Optimizations

The Global Optimization panel, shown in Figure 9.8, controls the method and depth by which the compiler optimizes your code.

Figure 9.8 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 76.

Table 9.2 SH optimizer levels

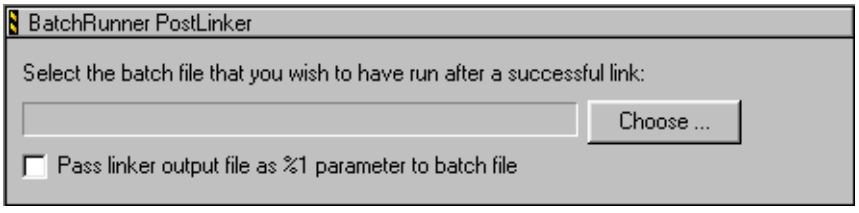
Level	Effect	Debugging
0	no optimizing	safe
1	Global Register Allocation Dead Code Elimination Loop Invariant Code Motion Branch Optimization Arithmetic Optimizations Expression Simplification	not safe
2	Common Sub-Expression Elimination Instruction Scheduling Delay-slot Filling Copy and Expression Propogation Peephole Optimization	not safe
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.

BatchRunner PostLinker

Figure 9.9 shows the BatchRunner Postlinker panel.

Figure 9.9 The BatchRunner Postlinker panel



his panel allows you to run a batch file after CodeWarrior successfully builds your project. To select a batch file to run, click **Choose** and locate the `.bat` file.

If you wish to have the name of the linker output file passed as a parameter to the batch file, click the checkbox.

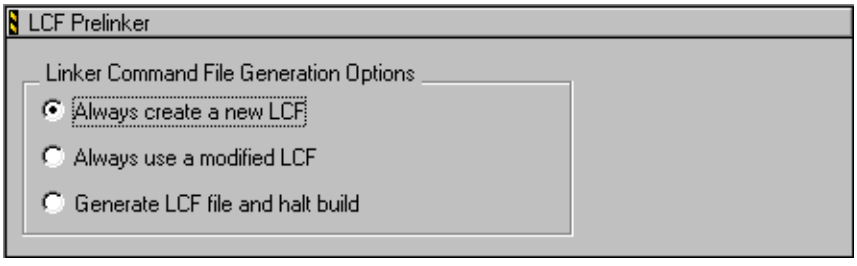
NOTE The batch file that you specify will not run unless the Batchrunner PostLinker has been selected as the post-linker for this target. The post-linker option can be modified in the Target Settings panel.

LCF Prelinker

The LCF Prelinker panel controls the behavior of the SH LCF (linker command file) Generator. This prelinker creates an LCF that contains the necessary linker instructions to correctly map your code and data in the final executable.

For more information about the LCF, see "ELF Linker and Command Language" on page 83. The LCF Prelinker panel itself is shown in Figure 9.10 .

Figure 9.10 LCF Prelinker panel



Linker Command File Generation Options

The options are set by a radio button, meaning that only one option may be selected at a time.

- **Always create a new LCF**

  This option forces the generator to create a new LCF every time you build the project. The new LCF will overwrite any LCF that already exists.

- **Always use a modified LCF**

  This option prevents the generator from creating a new LCF. Selecting this option has the same effect as not choosing any prelinker at all.

- **Generate LCF file and halt build**

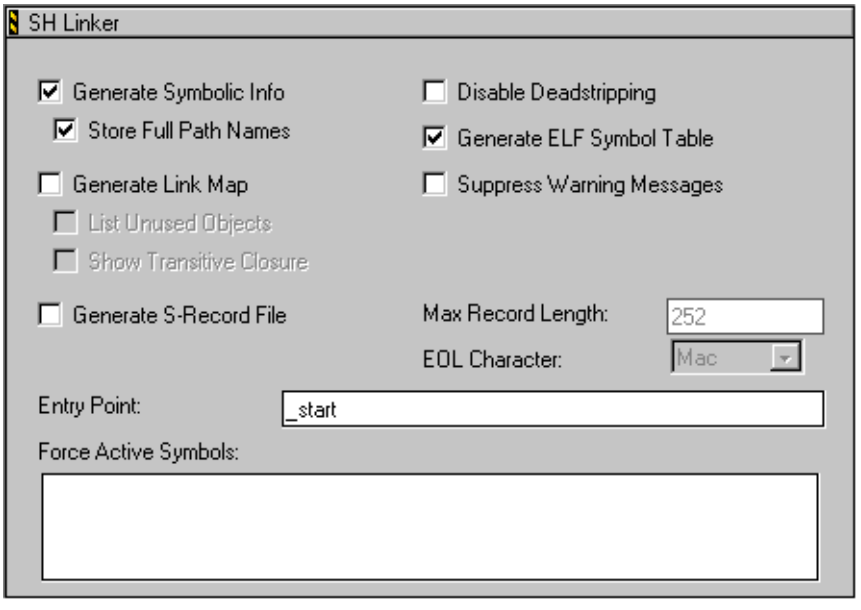
  This option forces the generator to create a new LCF, but it will halt the build process before the linker reads the file. Selecting this option allows you to make custom hand edit changes to a generated LCF.

WARNING! After you hand edit an LCF, you should either disable the LCF prelinker from the **Target Settings** panel, or select **Always use a modified LCF** from the choices listed above. Otherwise, your changes will be overwritten by the **SH LCF Generator** the next time you build your project.

SH Linker

The SH Linker panel, shown in Figure 9.11 , is where you control settings related to linking your object code into final form, be it executable, library, or other type of code.

Figure 9.11 The SH Linker panel.



The items in this panel are:

Generate Symbolic Info
Store Full Path Names
Generate Link Map
List Unused Objects
Generate S-Record File
EOL Character
Force Active Symbols
Disable Deadstripping
Generate ELF Symbol Table
Suppress Warning Messages
Show Transitive Closure
Max Record Length
Entry Point

Generate Symbolic Info

The **Generate Symbolic Info** checkbox controls whether the linker generates debugging information. The debugger information is included within the linked ELF file, and not generated as a separate file.

When this setting is on, the linker generates debugging information. Conversely, when this setting is off, the linker does *not* generate debugging information.

When you choose **Project > Enable Debugger**, CodeWarrior automatically turns on this item for you.

Store Full Path Names

The **Store Full Path Names** checkbox controls how the linker includes path information for source files, in the debugging information. When this setting is on, the linker generates debugging information and includes it within the linked ELF file. When this setting is off, the linker uses only the file names. In typical usage, this setting is on.

If memory is at a premium, you can uncheck this option to save memory on the target. The debugger can still find sources even if they do not have the paths, but the debugger may have to find the first file manually.

Disable Deadstripping

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

Generate ELF Symbol Table

The **Generate Elf Symbol Table** option generates an ELF symbol table, as well as a list of relocations, in the ELF output file.

Generate Link Map

The **Generate Link Map** option generates a link map, a test file that shows which files provide the definition for each object and function in your output file. The link map file also displays the address location given to each object and function, a memory

map of the sections residing in memory, and the value of each linker generated symbol.

The filename for the link map is the same as that of the output filename, but with the added extension of `.xMap`. The link map file is placed in the project folder.

List Unused Objects

The **List Unused Objects** option includes unused objects in the link map. This option only appears if the **Generate Link Map** option is active.

TIP The linker never deadstrips assembler relocatables or relocatables built with compilers other than CodeWarrior. If a relocatable wasn't built with the CodeWarrior C/C++ compiler, the link map can list all of the unused—but unstripped—symbols.

You can use this information to remove the symbols from the source and rebuild the relocatable to make your final process-image smaller.

Show Transitive Closure

The **Show Transitive Closure** option adds more detail to your link map file by recursively listing all of the objects referenced by `main()`.

Listing 9.1 shows some sample code.

Listing 9.1 Sample code to show transitive closure

```
void foo(){
    int a = 1001;
}
void foo1(){
    int b = 1002;
}
Sint32 njUserMain(void){
    foo();
    foo1();
```

```
    return NJD_USER_CONTINUE;
}
```

To show the effect of the **Show Transitive Closure** option, we compile the source and generate a link map file.

Listing 9.2 Effects of Show Transitive Closure in the link map file

```
# Link map of _start
  1] DSGLE found in strt1.obj.elf
      .
      .
      .
   4] _njUserMain (func,global) found in test.c
    5] _foo (func,global) found in test.c
    5] _foo1 (func,global) found in test.c
```

Suppress Warning Messages

Prevents the linker from reporting non-fatal warning messages. The **Suppress Warning Messages** option does not affect warning messages displayed by other parts of the IDE, including compilers. In typical usage, this setting is on.

Generate S-Record File

This option is not used for Dreamcast software development.

Max Record Length

This option is not used for Dreamcast software development.

EOL Character

This option is not used for Dreamcast software development.

Entry Point

The **Entry Point** edit field allows you to specify the first function that the linker uses when the program launches. This is the program's starting point.

The default `_start` function in the stationery is for CodeWarrior programs. When using the Sega Dreamcast SDK, the start function should be set to `SG_SEC`.

Force Active Symbols

The **Force Active Symbols** edit field allows the linker to include unreferenced symbols into the output file. It is a way to make symbols immune to deadstripping. This edit field is equivalent to `#pragma force_active`.

When listing multiple symbols, separate each item with a single space.

10

C and C++ for Dreamcast

This chapter describes the Metrowerks compiler for Dreamcast.

The sections in this chapter are:

- Number Formats for Dreamcast
- Calling Conventions for Dreamcast
- Variable Allocation for Dreamcast
- Optimizing Code 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	*C Compilers Reference* generally
using C/C++ Language and C/C++ Warnings settings panels	*C Compilers Reference*, "Setting C/C++ Compiler Options" chapter
controlling the size of C++ code	*C Compilers Reference*, "C++ and Embedded Systems" chapter
using compiler pragmas	*C Compilers Reference*, "Pragmas and Symbols" chapter

For this topic...	See...
initiating a build, controlling which files are compiled, handling error reports	*IDE User Guide*, "Compiling and Linking" chapter
information about a particular error	*Error Reference*
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
`bool`	8 bits	`true` or `false`
`char`	8 bits	`-128` to `127`
`unsigned char`	8 bits	`0` to `255`
`short`	16 bits	`-32,768` to `32,767`
`unsigned short`	16 bits	`0` to `65,535`
`int`	32 bits	`-2,147,483,648` to `2,147,483,647`
`unsigned int`	32 bits	`0` to `4,294,967,295`
`long`	32 bits	`-2,147,483,648` to `2,147,483,647`
`unsigned long`	16 bits	`0` to `4,294,967,295`
`long long`	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

CodeWarrior conforms fully with Hitachi's application binary interface (ABI) specification. Thus, the code generated by

CodeWarrior is compatible with code generated by Hitachi's SH compiler. The practical result is that you can link together any code built with these two different compilers.

Hitachi's ABI is 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 62.

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 Propogation
- 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 recommend 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 computers 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 Propogation

Replaces multiple occurences 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. Except for the pragmas in this list, you can use the pragmas defined in the *C Compilers Reference* in your code.

Table 10.4 Pragmas not supported for Dreamcast

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

The default Dreamcast values for pragmas `opt_unroll_count` and `opt_unroll_instr_count` are `2` and `40`, respectively. Documentation for these pragmas are in the *C Compilers Reference*.

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 restrictions:

- Exception Handling
- Streams and IO Classes
- Other Restrictions

Exception Handling

If you catch and throw exceptions in your program code, you must add the following lines in Listing 10.1 to your linker command file to define the exception table. In particular, you must add:

```
.exception
```

to the list of sections in the $INCLUDE, and you must create a new data segment for the exception table itself.

Listing 10.1 Creating an exception table in the LCF

```
$INCLUDE
{
   IP
   DSGLH
   DSGLE
   .exception   # Needed for C++
}

$SEGMENT DATA 0x8C040000 R
{
   # Include the exception table index.

   ALIGN(0x4);
   *(.exception)  # Needed for C++

   ALIGN(0x4);
   *(.exceptlist)  # Needed for C++
}
```

When you use exceptions and there is an exception handler, register R14 is used as the frame pointer. For example, when you use the intrinsic function `__alloca`, R14 is used as the frame pointer. You should reserve register R14 for this purpose and not use it for anything else. For more information on intrinsic functions, see "List of Intrinsic Functions" on page 116.

Streams and IO Classes

Neither streams nor IO classes are supported in this release, but there is a way to mimic the popular C++ stream function, `cout`, to send output to the debugger console window.

In your program, where you usually use something like

```
cout << "Hello"
```

you use the `mw_pr()` function as follows:

```
mw_pr("Hello");
```

To use the `mw_pr()` function, add the '`mw output.lib`' library to your project. This library is located in the `Dreamcast Support` folder.

Other Restrictions

The following are not supported in this release:

- defining member templates / nested class template members outside of the template definition
- member template conversion functions
- member template friends
- template template arguments
- 'exported' templates

11

ELF Linker and Command Language

The CodeWarrior ELF (Executable and Linking Format) Linker can do more than make a program file out of the object files of your project. The linker has several extended functions that allow you to manipulate your program's code in different ways. You can define variables during linking, control the link order to the granularity of a single function, and change the alignment.

All of these functions are accessed through commands in the linker command file (LCF). The linker command file has its own language complete with keywords, directives, and expressions, that are used to create the powerful specification for your output code.

NOTE The LCF syntax from the previous releases of the CodeWarrior for Dreamcast tools is not compatible with the new linker command file format.

The linker command file's syntax and structure is similar to that of a programming language. This language is described in the following sections:

- Structure of Linker Command Files —discusses command file organization.
- Linker Command File Syntax —how to program the linker to do specific tasks.
- Alphabetical Keyword Listing —an alphabetical listing of LCF functions and commands.

NOTE Understanding how ELF linkers work will help you understand our linker command file format. If you would like to become more familiar with the meaning of terms such as *.data* and the concepts of storage allocation and symbol management, we recommend that you read the following book:

John R. Levine,
Linkers and Loaders,
Ap Professional, 1999, ISBN 1-5586-0496-0.

Structure of Linker Command Files

Linker command files contain three main segments. These segments are listed below in the order they should appear in the command file:

- Closure Blocks —force functions into closure
- Memory Segment —map memory segments
- Sections Segment —define segment contents

A command file must contain a memory segment and a sections segment. Closure segments are optional.

Closure Blocks

The linker is very good at deadstripping unused code and data. We may sometimes find, however, that we have symbols that need to be kept in our output file even if they are never directly referenced. Interrupt handlers, for example, are usually linked at special addresses, without any explicit jumps to transfer control to these places.

Closure blocks provide a way for us to make symbols immune from deadstripping. The closure is transitive, meaning that symbols referenced by the symbol we are closing are also forced into closure, as are any symbols referenced by those symbols, and so on.

The two types of closure blocks available to us are as follows:

Symbol-level

Use FORCE_ACTIVE when you want to include a symbol into the link that would not be otherwise included. For example:

Listing 11.1 A sample symbol-level closure block

```
FORCE_ACTIVE {break_handler, interrupt_handler, my_function}
```

Section-level

Use KEEP_SECTION when you want to keep a section (usually a user-defined section) in the link. For example:

Listing 11.2 A sample section-level closure block

```
KEEP_SECTION {IP, DSGLH, DSGLE}
```

A variant is REF_INCLUDE. It keeps a section in the link, but only if the file where it is coming from is referenced. This is very useful to include version numbers. For example:

Listing 11.3 A sample section-level closure block with file dependency

```
REF_INCLUDE {.version}
```

Memory Segment

In the memory segment, we divide our available memory into segments. The memory segment format looks like Listing 11.4 .

Listing 11.4 A sample MEMORY segment

```
MEMORY {
    code_user (RWX): ORIGIN = 0x80001000, LENGTH = 0x19000
    bss_user  (RWX): ORIGIN = AFTER(segment_1), LENGTH = 0
    segment_x (RWX): ORIGIN = memory address, LENGTH = segment size
     and so on...
}
```

The (RWXO) portion consists of ELF access permission flags, **R**ead, **W**rite, and e**X**ecute. If CodeWarrior has overlay support for your target, the O flag is also available, and it represents a section of memory that is reserved for an **O**verlay.

ORIGIN represents the start address of the memory segment.

LENGTH represents the size of the memory segment.

If we can not predict how much space a segment will occupy, we can use the function AFTER and LENGTH = 0 (unlimited length) and CodeWarrior fills in the unknown values at link time.

For a detailed examination of the MEMORY segment, please read "MEMORY" on page 99.

Sections Segment

Inside the sections segment, we define the contents of our memory segments, and define any global symbols that we wish to use in our output file.

The format of a typical sections block looks like Listing 11.5 . In this sample segment,

Listing 11.5 A sample SECTIONS segment

```
SECTIONS {
   .section_name : #the section name is for your reference
   {               #the section name must begin with a '.'
      filename.c  (.text) #put the .text section from filename.c
      filename2.c (.text) #then the .text section from filename2.c
      filename.c  (.data)
      filename2.c (.data)
      filename.c  (.bss)
      filename2.c (.bss)
      . = ALIGN (0x10);   #align next section on 16-byte boundary.
   } > segment_1     #this means "map these contents to segment_1"

   .next_section_name:
   {
      more content descriptions
```

```
   } > segment_x      # end of .next_section_name definition
}                     # end of the sections block
```

For a detailed examination of the SECTIONS segment, please read "SECTIONS" on page 102.

Linker Command File Syntax

This section describes some practical ways in which you can use the commands of the linker command file to perform common tasks.

Alignment

To align data on a specific byte-boundary, you use the ALIGN and ALIGNALL commands to bump the location counter to the desired boundary. For example, the following fragment uses `ALIGN` to bump the location counter to the next 16-byte boundary.

```
file.c (.text)
. = ALIGN (0x10);
file.c (.data)    # aligned on a 16-byte boundary.
```

The same thing can be accomplished with `ALIGNALL` as follows:

```
file.c (.text)
ALIGNALL (0x10);  #everything past this point aligned on 16 bytes
file.c (.data)
```

For more information, see "ALIGN" on page 97 and "ALIGNALL" on page 97.

Arithmetic Operations

You may use standard C arithmetic and logical operations when you define and use symbols in the linker command file. Table 11.1 shows the order of precedence for each operator. All operators are left-associative. To learn more about C operators, refer to the *C Compiler Reference*.

Table 11.1 Arithmetic Operators

Precedence	Operators
highest (1)	- ~ !
2	* / %
3	+ -
4	>> <<
5	== != > < <= >=
6	&
7	|
8	&&
9	||

Comments

You may add comments to your file by using the pound character (#), C-style slash and asterisks (/*, */), or C++ style double-slashes (//). Comments are ignored by the LCF parser. The following are valid comments:

```
#  This is a one-line comment
/* This is a
             multiline comment */
* (.text) // This is a partial-line comment
```

Deadstrip Prevention

CodeWarrior removes unused code and data from the output file in a process known as deadstripping. To prevent the linker from deadstripping unreferenced code and data, use the FORCE_ACTIVE, KEEP_SECTION, and REF_INCLUDE directives to preserve them in the output file. Information on these directives can be found in "FORCE_ACTIVE" on page 98, "KEEP_SECTION" on page 99, and "REF_INCLUDE" on page 102.

Exception Tables

Exception tables are only required for C++. To create an exception table, add the `EXCEPTION` command to the end of your code section block. The two symbols, `__exception_table_start__` and `__exception_table_end__` are known to the runtime system.

Listing 11.6 Creating an exception table

```
__exception_table_start__ = .;
EXCEPTION
__exception_table_end__ = .;
```

Expressions, Variables and Integral Types

This section discusses variables, expressions, and integral types.

Variables and Symbols

All symbol names in a Linker Command File start with the underscore character (_), followed by letters, digits, or underscore characters. These are all valid lines for a command file:

```
_dec_num = 99999999;
_hex_num_ = 0x9011276;
```

Expressions and Assignments

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

```
_symbolicname = some_expression;
```

An assignment may only be used at the start of an expression, you cannot use something like this:

```
_sym1 + _sym2 = _sym3;  # ILLEGAL!
```

A semicolon is required at the end of an assignment statement.

When an expression is evaluated and assigned to a variable, it is given either an absolute or a relocatable type. An absolute expression type is one in which the symbol contains the value that it will have in the output file. A relocatable expression is one in which the value is expressed as a fixed offset from the base of a section.

Integral Types

The syntax for Linker Command File expressions is very similar to the syntax of the C programming language. All integer types are `long` or `unsigned long`.

Octal integers (commonly know as base eight integers) are specified with a leading zero, followed by numeral in the range of zero through seven. For example, here are some valid octal patterns you could put in your linker command file:

```
_octal_number  = 01234567;
_octal_number2 = 03245;
```

Decimal integers are specified as a non-zero numeral, followed by numerals in the range of zero through nine. Here are some

examples of valid decimal integers you could put in your linker command file:

```
_dec_num = 99999999;
_decimal_number = 123245;
_decyone = 9011276;
```

Hexadecimal (base sixteen) integers are specified as 0x or 0X (a zero with an X), followed by numerals in the range of zero through nine, and / or characters a through f. Here are some examples of valid hexadecimal integers you could put in your linker command file:

```
_somenumber = 0x999999FF;
_fudgefactorspace = 0X123245EE;
_hexonyou = 0xFFEE;
```

To create a negative integer, use the minus sign (-) in front of the number, as in:

```
_decimal_number = -123456;
```

File Selection

When defining the contents of a `SECTION` block, you must specify the source files that are contributing their sections. The standard way of doing this is to simply list the files.

```
SECTIONS {
  .example_section :
    {
       main.c  (.text)
       file2.c (.text)
       file3.c (.text)
```

In a large project, the list can grow to become very long. For this reason, we have the '*' keyword. It represents the filenames of every file in your project. Note that since we have already added the `.text` sections from the files `main.c`, `file2.c`, and `file3.c`, the '*' keyword will not add the `.text` sections from those files again.

```
        *  (.text)
```

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

```
        GROUP(fileGroup1) (.text)
        GROUP(fileGroup1) (.data)
    } > MYSEGMENT
}
```

See also "SECTIONS" on page 102.

Function Selection

The OBJECT keyword gives you precise control over how functions are placed within your section. For example, if you want the functions `bar` and `foo` to be placed before anything else in a section, you might use something like the following:

```
SECTIONS {
  .program_section :
    {
      OBJECT (bar, main.c)
      OBJECT (foo, main.c)
      * (.text)
     } > ROOT
```

NOTE When using C++, you must specify functions by their mangled names.

It is important to note that if an object is written once using the '`OBJECT`' function selection keyword, the same object will not be written again using the '`*`' file selection keyword.

See also "SECTIONS" on page 102.

Stack and Heap

To reserve space for the stack and heap, we perform some arithmetic operations to set the values of the symbols used by the runtime. The following is a code fragment from a section definition that illustrates this arithmetic (Listing 11.7).

Listing 11.7 Setting up some heap

```
_heap_addr = .;
_heap_size = 0x2000;  /* this is the size of our heap */
_heap_end = _heap_addr + _heap_size;
. = _heap_end         /* reserve the space */
```

We do the same thing for the stack, using the ending address of the heap as the start of our stack.

Listing 11.8 Setting up the stack

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

Static Initializers

Static initializers must be invoked to initialize static data before main() starts. The CodeWarrior linker generates the static initializer section with the `STATICINIT` keyword.

In your linker command file, use something similar to the following to tell the linker where to put the table (relative to the '`.`' location counter):

```
..sinit :
{
   . = ALIGN (0x08);
   __sinit__ - .;
   STATICINIT
   . = ALIGN(0x04);
} > .sinit
```

The symbol `__sinit__` is known to the runtime. In the startup code, you can use something similar to the following to call accompany the use of static initializers in the linker command file:

```
#ifdef __cplusplus
/* call the c++ static initializers */
__call_static_initializers();
#endif
```

Writing Data Directly to Memory

You can write data directly to memory using the `WRITEx` command in the linker command file. `WRITEB` writes a byte, `WRITEH` writes a two-byte halfword, and `WRITEW` writes a four-byte word. The data is inserted at the section's current address.

Listing 11.9 Embedding data directly into the output.

```
.example_data_section :
{
   WRITEB 0x48;  /*  'H'  */
   WRITEB 0x69;  /*  'i'  */
   WRITEB 0x21;  /*  '!'  */
```

The example shown in Listing 11.9 is similar to the technique used to insert overlay headers on targets that have overlay support.

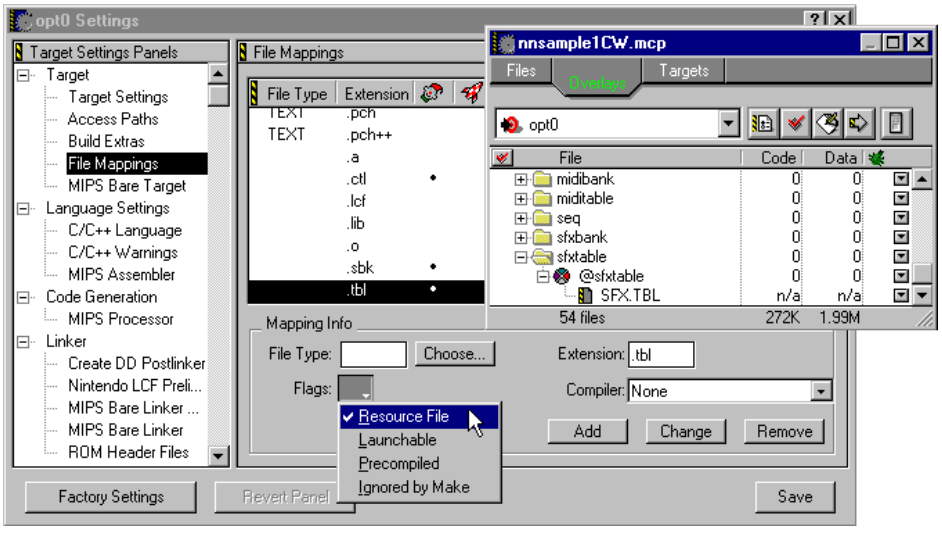
If you want to insert a complete binary file, you can use the INCLUDE command.

Listing 11.10 Embedding a binary file into the output.

```
    _musicStart = .;
    INCLUDE (music.mid)
    _musicEnd = .;
}  > DATA
```

The binary file must be included in your IDE project, and the file's extension must be typed as a resource file in the **File Mappings** target settings panel. For an illustration of how this is done, see Figure 11.1 . For more help with resource files, please refer to the *IDE User Guide*.

Figure 11.1 Marking a binary file type as a resource in File Mappings



Alphabetical Keyword Listing

The following is an alphabetical list of all the valid linker command file functions, keywords, directives, and commands:

. (location counter)	ADDR
ALIGN	ALIGNALL
EXCEPTION	FORCE_ACTIVE
GROUP	INCLUDE
KEEP_SECTION	MEMORY
OBJECT	OVERLAYID
REF_INCLUDE	SECTIONS
SIZEOF	STATICINIT

WRITEB WRITEH

WRITEW

. (location counter)

The period character '.' always maintains the current position of the output location. Since the period always refers to a location in a SECTIONS block, it can not be used outside a section definition.

'.' may appear anywhere a symbol is allowed. Assigning a value to '.' that is greater than its current value causes the location counter to move, but the location counter can never be decremented.

This effect can be used to create empty space in an output section. In the example that follows, the location counter is moved to a position that is 0x10000 bytes past the symbol __start.

Listing 11.11 Moving the location counter

```
..data :
{
     *.(data)
     *.(D)
     *.(D32)
     __start = .;
     . = __start + 0x10000;
     __end = .;
} > DATA
```

ADDR

The ADDR function returns the address of the named section or memory segment.

Prototype `ADDR (sectionName | segmentName)`

In the example below, we use ADDR to assign the address of ROOT to the symbol __rootbasecode (Listing 11.12).

Listing 11.12 ADDR() function

```
MEMORY{
    ROOT  (RWX) : ORIGIN = 0x80000400, LENGTH = 0
}

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

ALIGN

The ALIGN function returns the value of the location counter aligned on a boundary specified by the value of alignValue.

Prototype `ALIGN(alignValue)`

alignValue must be a power of two.

Please note that ALIGN does not update the location counter; it only performs arithmetic. To update the location counter, you have to use an assignment such as the following.

```
. = ALIGN(0x10);    #update location counter to 16 byte alignment
```

ALIGNALL

ALIGNALL is the command version of the ALIGN function. It forces the minimum alignment for all the objects in the current segment to the value of alignValue.

Prototype `ALIGNALL(alignValue);`

alignValue must be a power of two.

Unlike its counterpart ALIGN, ALIGNALL is an actual command. It updates the location counter as each object is written to the output (Listing 11.13).

Listing 11.13 ALIGNALL example

```
.code :
{
    ALIGNALL(16);  // Align code on 16 byte boundary
    *    (.init)
    *    (.text)

    ALIGNALL(64);  //align data on 64 byte boundary
    *    (.rodata)
} > .text
```

EXCEPTION

The EXCEPTION command creates the exception table index in the output file. Exception tables are only required for C++. To create an exception table, add the EXCEPTION command to the end of your code section block. The two symbols, __exception_table_start__ and __exception_table_end__ are known to the runtime system.

Listing 11.14 Creating an exception table

```
  __exception_table_start__ = .;
  EXCEPTION
  __exception_table_end__ = .;
```

FORCE_ACTIVE

The FORCE_ACTIVE directive allows you to specify symbols that you do not want the linker to deadstrip. When using C++, you must specify symbols using their mangled names.

Prototype `FORCE_ACTIVE{ symbol[, symbol] }`

GROUP

The GROUP keyword allows you to selectively include files and sections from certain file groups.

Prototype `GROUP (fileGroup) (sectionType)`

For example, if you specify this:

```
GROUP (BAR) (.bss)
```

you are specifying all the .bss sections of the files in the file group named BAR.

INCLUDE

The INCLUDE command allows you to include a binary file in the output file.

Prototype

```
INCLUDE filename
```

KEEP_SECTION

The KEEP_SECTION directive allows you to specify sections that you do not want the linker to deadstrip.

Prototype

```
KEEP_SECTION{ sectionType[, sectionType] }
```

MEMORY

The MEMORY directive allows you to describe the location and size of memory segment blocks in the target. Using this directive, you tell the linker the memory areas to avoid, and the memory areas into which it should link your code and date.

The linker command file may only contain one MEMORY directive. However, within the confines of the MEMORY directive, you may define as many memory segments as you wish.

Prototype

```
MEMORY { memory_spec }
```

The memory_spec is:

```
segmentName (accessFlags) : ORIGIN = address, LENGTH = length
[,COMPRESS] [> fileName]
```

segmentName can include alphanumeric characters and underscore '_' characters.

accessFlags are passed into the output ELF file (Phdr.p_flags). The accessFlags can be:

R–read	W–write
X–executable	O–overlay

address is one of the following:

- **a memory address**

  You can specify a hex address such as 0x80000400.

- **an AFTER command**

  If you do not want to compute the addresses using offsets, you can use the AFTER(*name [,name]*) command to tell the linker to place the memory segment after the specified segment. In the following example, overlay1 and overlay2 are placed after the code segment, and data is placed after the overlay segments.

```
MEMORY{
code     (RWX)  : ORIGIN = 0x80000400, LENGTH = 0
overlay1 (RWXO) : ORIGIN = AFTER(code), LENGTH = 0
overlay2 (RWXO) : ORIGIN = AFTER(code), LENGTH = 0
data      (RWX) : ORIGIN = AFTER (overlay1, overlay2), LENGTH = 0
}
```

When multiple memory segments are specified as parameters for AFTER, the highest memory address is used. This is useful for overlays when you do not know which overlay takes up the most memory space.

length is one of the following:

- **a value greater than zero**

  If you try to put more code and data into a memory segment than your specified length allows, the linker stops with an error.

- **autolength by specifying zero**

  When the length is 0, the linker lets you put as much code and data into a memory segment as you want.

NOTE There is no overflow checking with autolength. You can end up with an unexpected result if you use the autolength feature without leaving enough free memory space to contain the memory segment. For this reason, when you use autolength, we recommend that you use the AFTER keyword to specify origin addresses.

`> fileName` is an option to write the segment to a binary file on disk instead of an ELF program header. The binary file is put in the same folder as the ELF output file. This option has two variants:

- `> fileName`

  writes the segment to a new file.
- `>> fileName`

  appends the segment to an existing file.

OBJECT

The `OBJECT` keyword gives you control over the order in which functions are placed in the output file.

Prototype `OBJECT (`*`function, sourcefile.c`*`)`

It is important to note that if an object is written to the outfile using the `OBJECT` keyword, the same object will not be written again by either the `GROUP` keyword or the '*' wildcard selector.

OVERLAYID

The `OVERLAYID` function returns the overlay ID of a given section. This function is useful only if CodeWarrior supports overlays for your target.

Prototype `OVERLAYID (`*`sectionName | segmentName`*`)`

This function is commonly used to write part of the overlay header . For example:

```
WRITEW OVERLAYID (.myoverlay);
```

REF_INCLUDE

The REF_INCLUDE directive allows you to specify sections that you do not want the linker to deadstrip, but only if they satisfy a certain condition: the file that contains the section must be referenced. This is useful if you want to include version information from your sourcefile components.

Prototype

```
REF_INCLUDE{ sectionType [, sectionType]}
```

SECTIONS

A basic SECTIONS directive has the following form:

Prototype

```
SECTIONS { <section_spec> }
```

section_spec is one of the following:

```
sectionName : [AT (loadAddress)] {contents}  > segmentName
or,
sectionName : [AT (loadAddress]] {contents} >> segmentName
```

sectionName is the section name for the output section. It must start with a period character. For example, " .mysection".

AT (loadAddress) is an optional parameter that specifies the address of the section. The default (if not specified) is to make the load address the same as the relocation address.

contents are made up of statements. These statements can

- assign a value to a symbol. See "Alphabetical Keyword Listing" on page 95, "Arithmetic Operations" on page 88, and ". (location counter)" on page 96.
- describe the placement of an output section, including which input sections are placed into it. See "File Selection" on page 91, "Function Selection" on page 92, and "Alignment" on page 87.

segmentName is the predefined memory segment into which you want to put the contents of the section. The two variants are:

- **> segmentName**

This places the section contents at the beginning of the memory segment `segmentName`.

- **>> segmentName**

  This appends the section contents to the memory segment `segmentName`.

Here is an example section definition

Listing 11.15 An example section definition

```
SECTIONS {
  .text : {
            _textSegmentStart = .;
            foobar.c (.text)
            . = ALIGN (0x10);
            barfoo.c (.text)
            _textSegmentEnd = .;
  }
  .data : { *(.data) }
  .bss  : { *(.bss)
            *(COMMON)
  }
}
```

SIZEOF

The `SIZEOF` function returns the size of the given segment or section. The return value is the size in bytes.

Prototype `SIZEOF(`*`segmentName`* | *`sectionName`*`)`

STATICINIT

The STATICINIT directive creates the static initializer tables required for C++ programs.

Prototype `STATICINIT`

WRITEB

WRITEB inserts a byte of data at the current address of a section.

Prototype `WRITEB (expression);`

`expression` is any expression that returns a value `0x00` to `0xFF`.

WRITEH

WRITEH inserts a halfword of data at the current address of a section.

Prototype `WRITEH (expression);`

`expression` is any expression that returns a value `0x0000` to `0xFFFF`.

WRITEW

WRITEW inserts a word of data at the current address of a section.

Prototype `WRITEW (expression);`

`expression` is any expression that returns a value `0x00000000` to `0xFFFFFFFF`.

12

Linker Issues for Dreamcast

This section discusses issues surrounding the Dreamcast linker. The sections in this chapter are:

Deadstripping Unused Code and Data

The Dreamcast 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.

If you have unreferenced sections of code or data that must be kept in the final application, use FORCE_ACTIVE directive of the linker command file to prevent the linker from deadstripping those unreferenced sections. For more information about `FORCE_ACTIVE` and other closure directives, see "Closure Blocks" on page 84. You can also set the Disable Deadstripping option in the SH Linker preferences panel. For a description of this panel, see "SH Linker" on page 67.

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.

Link Order

The link order is generally specified in the Overlays 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 Dreamcast SDK libraries. If you are not using the stationery, please link the libraries 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
```

You must place your source files and other libraries after the files listed above.

TIP 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 will not be able to find the executable file and will stop the build with an appropriate message.

Function Reordering

Automatic function reordering is not supported in this release. To reorder functions manually, you must interpret the `.lor` file created by the CodeScape profiler to improve the hit rates for the instruction cache.

The .lor file contains the recommended arrangement for your functions. Use the OBJECT directive of the CodeWarrior linker to arrange the listed functions in the linker command file. For more information, see "Function Selection" on page 92.

13

Inline Assembler and Intrinsics 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

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

The topics in this section include:

Inline Assembly 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 13.1. 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 13.1 Defining a function with `asm`

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

The second method is shown in Listing 13.2. 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 13.2 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.

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 115.

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 62.

  You may suppress assembly optimizations by using the **..set noreorder** directive. For information on the .set directive, see ".set" on page 114.

- 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 not case sensitive. These two statements are exactly the same:

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

- 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 13.3 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 13.4, 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 13.4 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 13.5 for an example.

Listing 13.5 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 62.

The example shown in Listing 13.6 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 13.6 .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 13.7 .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 instrinsic 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

The intrinsic functions listed in Table 13.1 are available for you to use in your CodeWarrior project.

Table 13.1 Intrinsic functions

`__abs`	`__labs`
`__alloca`	`__memcpy`

__abs

Description Intrinsic for absolute value

Example
```
int Intrinsic_abs (int i)
{
    int j;
    j = __abs(i);
    return j;
}
```

__labs

Description Intrinsic for `long` absolute value

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

__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;
}
```

__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));
}
```

Hitachi SH C Compiler-compatible Intrinsic Functions

The intrinsic functions listed in Table 13.2 provide compatibility with the intrinsic functions of Hitachi's SH C compiler.

Table 13.2 Hitachi SH C compiler-compatible Intrinsic functions

set_cr	get_cr
set_imask	get_imask
set_vbr	get_vbr
set_gbr	get_gbr
gbr_read_byte	gbr_write_byte
gbr_read_word	gbr_write_word
gbr_read_long	gbr_write_long
gbr_and_bytes	gbr_or_bytes
gbr_xor_byte	gbr_tst_byte
sleep	tas
trapa	prefetch
macw	macwl
set_fpscr	get_fpscr
fipr	ftrv
ftrvadd	ftrvsub
add4	sub4
mtrx4mul	mtrx4muladd
mtrx4mulsub	
ld_ext	st_ext
fabs	fabsf
sqrt	sqrtf
fsrra	fsca
strcpy	strcmp

set_cr

Description Sets the 32-bit status register.

Prototype `void set_cr(int cr);`

get_cr

Description References the status register.

Prototype `int get_cr(void);`

set_imask

Description Sets the 4-bit interrupt mask.

Prototype `void set_imask(int mask);`

get_imask

Description References the interrupt mask.

Prototype `void get_imask(int mask);`

set_vbr

Description Sets the 32-bit vector base register.

Prototype `void set_vbr(void **base);`

get_vbr

Description References the vector base register.

Prototype `void **get_vbr(void);`

set_gbr

Description Sets the 32-bit global base register.

Prototype `void set_gbr(void *base);`

get_gbr

Description References the global base register.

Prototype `void *get_gbr(void);`

gbr_read_byte

Description References the8-bit data byte at the address indicated by the GBR and offset.

Prototype `unsigned char gbr_read_byte(int offset)`

gbr_write_byte

Description Writes a byte of data at the address indicated by the GBR and offset.

Prototype
```
void gbr_write_byte(int offset,
                    unsigned char data);
```

gbr_read_word

Description References the 16-bit data word at the address indicated by the GBR and offset.

Prototype
```
unsigned char gbr_read_word(int offset);
```

gbr_write_word

Description Writes a word of data at the address indicated by the GBR and offset.

Prototype
```
void gbr_write_wrod(int offset,
                    unsigned short data);
```

gbr_read_long

Description References the 32-bit data long at the address indicated by the GBR and offset.

Prototype
```
unsigned gbr_read_long(int offset);
```

gbr_write_long

Description writes a long of data at the address indicated by the GBR and offset.

Prototype
```
void gbr_write_long(int offset,
                    unsigned long data);
```

gbr_and_bytes

Description Takes the specified mask and ANDs it with the byte data at the address of the GBR and offset. The result is stored at the same address.

Prototype
```
void gbr_and_bytes(int offset,
                   unsigned char mask);
```

gbr_or_bytes

Description Takes the specified mask and ORs it with the byte data at the address of the GBR and offset. The result is stored at the same address.

Prototype
```
void gbr_or_bytes(int offset
                  unsigned char mask);
```

gbr_xor_byte

Description Takes the specified mask and XORs it with the byte data at the address of the GBR and offset. The result is stored at the same address.

Prototype
```
void gbr_xor_byte(int offset
                  unsigned char mask);
```

gbr_tst_byte

Description Takes the specified mask and ANDs it with the byte data at the address of the GBR and offset. If the result is 0, the T bit is set to 1 (true). Otherwise, the T bit is set to 0 (false).

Prototype
```
void gbr_tst_byte(int offset
                  unsigned char mask);
```

sleep

Description Invokes the SLEEP instruction

Prototype
```
void sleep(void);
```

tas

Description Invokes the TAS.B instruction with addr.

Prototype
```
int tas(char *addr);
```

trapa

Description Invokes the TRAPA instruction with trap_no.

Prototype
```
int trapa(int trap_no);
```

prefetch

Description — Invokes the PREF instruction. This writes the 16-bytes of memory indicated by the pointer to the cache memory.

Prototype
```
void prefetch (void *p);
```

macw

Description — Multiplies and accumulates the contents of two data tables.

Prototype
```
int macw(short *ptr1,
         short *ptr2,
         unsigned int count);
```

Remarks — The sizes of the tables at the addresses indicated by ptr1 and ptr2 must have 2-byte and 4-byte alignment, respectively.

macwl

Description — Multiplies and accumulates the contents of two data tables using a ring buffer mask.

Prototype
```
int macwl(short *ptr1,
          short *ptr2,
          unsigned int count,
          unsigned int mask);
```

Remarks — The tables at the addresses indicated by ptr1 and ptr2 must be aligned to twice the size of the ring buffer mask.

set_fpscr

Description — Writes a 32-bit value to the floating-point unit system / control register.

Prototype
```
void set_fpscr(int cr);
```

get_fpscr

Description — Reads the value of the floating point unit system / control register.

Prototype
```
int get_fpscr();
```

Return — `get_fpscr()` returns the FPSCR value.

fipr

Description Calculates the inner product of single-precision floating-point vectors `vect1` and `vect2`.

Prototype
```
float fipr(float vect1[4], float vect2[4]);
```

Remarks No remarks

Return `fipr()` returns the inner product of `vect1` and `vect2` as a `float`.

ftrv

Description Multiplies a single-precision floating-point vector `(vect1)` with a 4x4 matrix stored in the extension registers `(MTX)`. The result is stored in `vect2`.

```
vect2 = vect1 x MTX
```

Prototype
```
void ftrv(float vect1[4], float vect2[4]);
```

Remarks Before using this function, you must first use the `ld_ext` intrinsic function to load `MTX`'s data into the extension registers.

ftrvadd

Description Multiplies a single-precision floating-point vector `(vect1)` with a 4x4 matrix stored in the extension registers `(MTX)`, then adds `vect2` to the result. The result is stored in `vect3`.

```
vect3 = (vect1 x MTX) + vect2
```

Prototype
```
void ftrvad(float vect1[4], float vect2[4],
            float vect3[4]);
```

Remarks Before using this function, you must first use the `ld_ext` intrinsic function to load `MTX`'s data into the extension registers.

ftrvsub

Description Multiplies a single-precision floating-point vector (`vect1`) with a 4x4 matrix stored in the extension registers `(MTX)`, then subtracts `vect2` from the result. The result is stored in `vect3`.

```
vect3 = (vect1 x MTX) - vect2
```

Prototype
```
void ftrvsub(float vect1[4], float vect2[4],
             float vect3[4]);
```

Remarks Before using this function, you must first use the ld_ext intrinsic function to load MTX's data into the extension registers.

add4

Description Adds together the single-precision floating point vectors vect1 and vect2. The result is stored in vect3.

```
vect3 = vect1 + vect2
```

Prototype
```
void add4(float vect1[4], float vect2[4],
          float vect3[4]);
```

sub4

Description Subtracts the single-precision floating-point vector vect2 from vector vect1. The result is stored in vect3.

```
vect3 = vect1 - vect2
```

Prototype
```
void sub4(float vect1[4], float vect2[4],
          float vect3[4]);
```

mtrx4mul

Description Multiplies a single-precision floating-point 4x4 matrix (mtrx1) with the 4x4 matrix stored in the extension registers (MTX). The result is stored into mtrx2.

```
mtrx2 = mtrx1 x MTX
```

Prototype
```
void mtrx4mul(float mtrx1[4][4],
              float mtrx2[4][4]);
```

Remarks Before using this function, you must first use the ld_ext intrinsic function to load MTX's data into the extension registers.

mtrx4muladd

Description Multiplies a single-precision floating-point 4x4 matrix (mtrx1) with the matrix stored in the extension registers (MTX). Another matrix, mtrx2, is then added, and the result is stored into mtrx3.

```
mtrx3 = (mtrx1 x MTX) + mtrx2
```

Prototype
```
void mtrx4muladd(float mtrx1[4][4],
                 float mtrx2[4][4],
                 float mtrx3[4][4]);
```

Remarks Before using this function, you must first use the ld_ext intrinsic function to load MTX's data into the extension registers.

mtrx4mulsub

Description Multiplies a single-precision floating-point 4x4 matrix (mtrx1) with the matrix stored in the extension registers (MTX). Another matrix, mtrx2, is then subtracted, and the result is stored into mtrx3.

```
mtrx3 = (mtrx1 x MTX) - mtrx2
```

Prototype
```
void mtrx4mulsub(float mtrx1[4][4],
                 float mtrx2[4][4],
                 float mtrx3[4][4]);
```

Remarks Before using this function, you must first use the ld_ext intrinsic function to load MTX's data into the extension registers.

ld_ext

Description Loads the data of a 4x4 matrix into the floating-point extension registers.

Prototype
```
void ld_ext(float mtrx[4][4]);
```

Remarks No remarks

st_ext

Description Reads the floating-point extension registers and stores the matrix data into a 4x4 matrix.

Prototype
```
void st_ext(float mtrx[4][4]);
```

fabs

Description Finds the absolute value of a float.

Prototype
```
float fabs(float x);
```

fabsf

Description Finds the absolute value of a float using single-precision.

Prototype
```
float fabsf (float x);
```

sqrt

Description Finds the square root of a positive float.

Prototype
```
float sqrt(float x);
```

sqrtf

Description Finds the square root of a positive float using single-precision.

Prototype
```
float sqrtf(float x);
```

fsrra

Description Finds the reciprocal square root using single-precision.

Prototype
```
float fssca(float val);
```

fsca

Description Finds the sine and cosine of rad.

Prototype
```
void fsca(long rad,
          float *sinval,
          float *cosval);
```

Remarks `sinval` is the value of the sine.
`cosval` is the value of the cosine.

strcpy

Description Copies string s2 to string s1.

Prototype
```
char strcpy(char *s1,
            char *s2);
```

strcmp

Description Compares string s1 to string s2.

Prototype
```
int strcmp(char *s1
           char *s2);
```

Mnemonics for Inline Assembly

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

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 13.8 for an example. Here, we use the special `MOV.L` instruction to load the constant `12345678` into `R1`.

Listing 13.8 Example of using special instructions

```
asm int foo1() {
    MOV.L    12345678,R1;
    RTS;
    NOP;
}
```

The compiler actually expands the special instruction into the machine instructions shown in Listing 13.9.

Listing 13.9 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 13.10. 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 13.10 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 13.3 lists the inline assembly instructions supported by our compiler. They are similar to the regular assembler instructions, but '/' characters have changed to '_'. The instructions that we do not support in inline assembly are greyed out and marked as unsupported.

Table 13.3 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)"	
"BF"	"l"	
"BF_S"	"l"	
"BRA"	"m"	
"BRAF"	"Rn"	
"BSR"	"m"	unsupported
"BSRF"	"Rn"	
"BT"	"l"	
"BT_S"	"l"	

Mnemonic	Format	Support
"CLRMAC"	""	
"CLRS"	""	
"CLRT"	""	
"CMP_EQ"	"i,R0"	
"CMP_EQ"	"Rm,Rn"	
"CMP_GE"	"Rm,Rn"	
"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"	

Mnemonic	Format	Support
"FCNVDS"	"Fn"	
"FCNVSD"	"Fn"	
"FDIV"	"Fm,Fn"	
"FIPR"	"FVm,FVn"	unsupported
"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

Mnemonic	Format	Support
"FMOV"	"Xm,@(R0,Rn)"	unsupported
"FMOV"	"Xm,Xn"	unsupported
"FMOV"	"Xm,Dn"	unsupported
"FMOV"	"Dm,Xn"	unsupported
"FMUL"	"Fm,Fn"	
"FNEG"	"Fn"	
"FRCHG"	""	
"FSCHG"	""	
"FSQRT"	"Fn"	
"FSTS"	"Fn"	
"FSUB"	"Fm,Fn"	
"FTRC"	"Fn"	
"FTRV"	"XM,FVn"	unsupported
"JMP"	"@Rn"	
"JSR"	"@Rn"	unsupported
"LDC"	"Rn,GBR"	
"LDC"	"Rn,SR"	
"LDC"	"Rn,VBR"	
"LDC"	"Rn,SSR"	
"LDC"	"Rn,SPC"	
"LDC"	"Rn,DBR"	

Mnemonic	Format	Support
"LDC"	"Rn,Rb"	unsupported
"LDC.L"	"@Rn+,GBR"	
"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"	
"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"	

Mnemonic	Format	Support
"MOV.B"	"@(d4,Rm),R0"	
"MOV.B"	"@(R0,Rm),Rn"	
"MOV.B"	"@Rm+,Rn"	
"MOV.B"	"@Rm,Rn"	
"MOV.B"	"R0,@(d8,GBR)"	
"MOV.B"	"R0,@(d4,Rm)"	
"MOV.B"	"Rm,@(R0,Rn)"	
"MOV.B"	"Rm,@-Rn"	
"MOV.B"	"Rm,@Rn"	
"MOV.W"	"@(d8,GBR),R0"	
"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)"	
"MOV.W"	"R0,@(d4,Rm)"	
"MOV.W"	"Rm,@(R0,Rn)"	
"MOV.W"	"Rm,@-Rn"	
"MOV.W"	"Rm,@Rn"	
"MOV.L"	"@(d8,GBR),R0"	
"MOV.L"	"@(d8,PC),Rn"	
"MOV.L"	"@(d4,Rm),Rn"	
"MOV.L"	"@(R0,Rm),Rn"	
"MOV.L"	"@Rm+,Rn"	

Mnemonic	Format	Support
"MOV.L"	"@Rm,Rn"	
"MOV.L"	"R0,@(d8,GBR)"	
"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"	
"MOVT"	"Rn"	
"MUL.L"	"Rm,Rn"	
"MULS.W"	"Rm,Rn"	
"MULU.W"	"Rm,Rn"	
"NEG"	"Rm,Rn"	
"NEGC"	"Rm,Rn"	
"NOP"	""	
"NOT"	"Rm,Rn"	
"OCBI"	"@Rn"	
"OCBP"	"@Rn"	
"OCBWB"	"@Rn"	
"OR"	"i,R0"	
"OR"	"Rm,Rn"	
"OR.B"	"i,@(R0,GBR)"	
"PREF"	"@Rn"	
"ROTCL"	"Rn"	
"ROTCR"	"Rn"	

Mnemonic	Format	Support
"ROTL"	"Rn"	
"ROTR"	"Rn"	
"RTE"	""	
"RTS"	""	
"SETS"	""	
"SETT"	""	
"SHAD"	"Rm,Rn"	
"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"	
"STC"	"SR,=Rn"	
"STC"	"VBR,=Rn"	
"STC"	"SSR,=Rn"	
"STC"	"SPC,=Rn"	
"STC"	"DBR,=Rn"	

Mnemonic	Format	Support
"STC"	"Rb,=Rn"	unsupported
"STC.L"	"G,@Rn+"	
"STC.L"	"SR,@Rn+"	
"STC.L"	"VBR,@Rn+"	
"STC.L"	"SSR,@Rn+"	
"STC.L"	"SPC,@Rn+"	
"STC.L"	"DBR,@Rn+"	
"STC.L"	"Rb,@Rn+"	unsupported
"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"	

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

14

Overlays

Using overlays is a programming technique that allows a program to fit into memory that is smaller than the program itself, even in the absence of virtual memory.

To use overlays, you break your program into chunks of code that do not all have to be loaded at the same time. These chunks of code are compiled into overlays that link against each other. Your main program is then responsible for loading the individual overlays as they are needed.

For example, you might have an overlay that plays mpeg movies, an overlay that displays the main menu, and another overlay for the gameplay. Since these overlays run independently of each other, you can swap them in and out of memory as you need them.

The topics in this chapter include:

- Building an Overlay Project —an introduction to CodeWarrior's overlay feature.
- Overlay Notes —technical notes related to overlays.

Building an Overlay Project

CodeWarrior supports overlays with a special tab in the **Project** window labled **Overlays**. This feature allows you to easily create overlays for your game.

In this tutorial, we will build and test a program that contains two overlays.

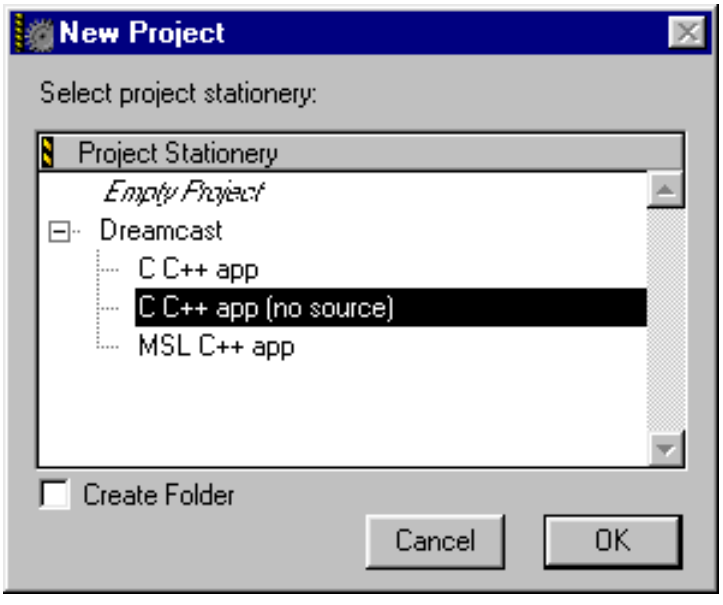
Follow these steps to build a project that uses overlays.

1. **Build the Project File**

a. Create a new project.

Launch CodeWarrior and select **New Project** from the **File** menu. Choose `Dreamcast` and `C C++ app (no source)` for the stationery as shown in Figure 14.1. Note that you may not want to create a new folder.

Figure 14.1 Choosing Stationery for the Overlay Tutorial



In the file dialog, locate the `Tutorial` folder which is typically located at

`Examples\Overlay\`

Enter the file name `overlay` and click **OK**. You should now see a project window named `overlay.mcp`.

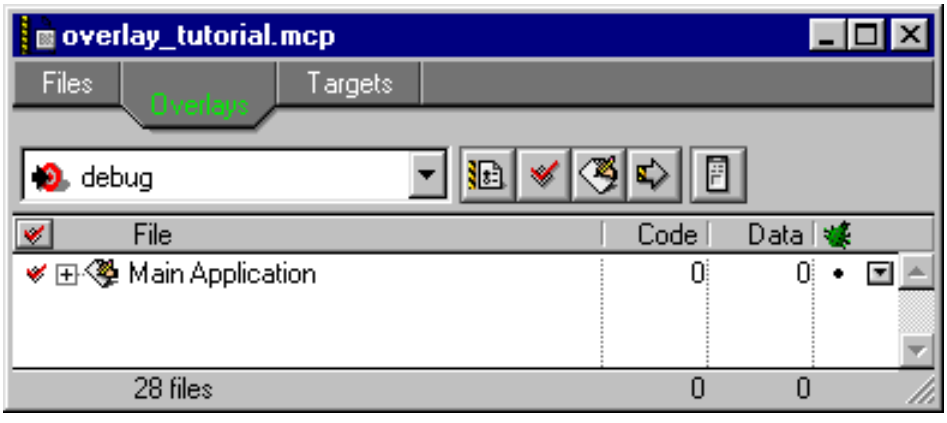
b. Add files to the project.

Add the source files `test.c`, `sbinit.c`, `mw_utils.c`, `njloop.c`, `overlay1.c` and `overlay2.c` on to the project window and under the `Sources` group. You should see the files listed in the file list.

c. Create the overlays.

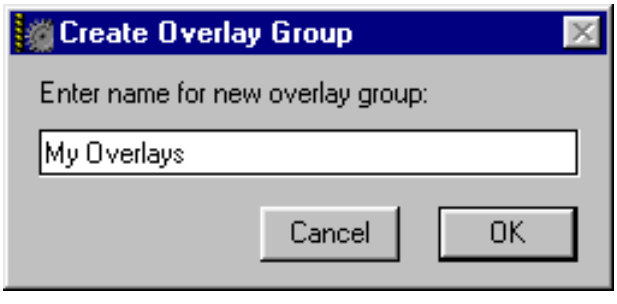
Click the **Overlays** tab at the top of the project window as shown in Figure 14.2 . Expand the Main Application list to show your source files.

Figure 14.2 Overlay Tab in Project Window



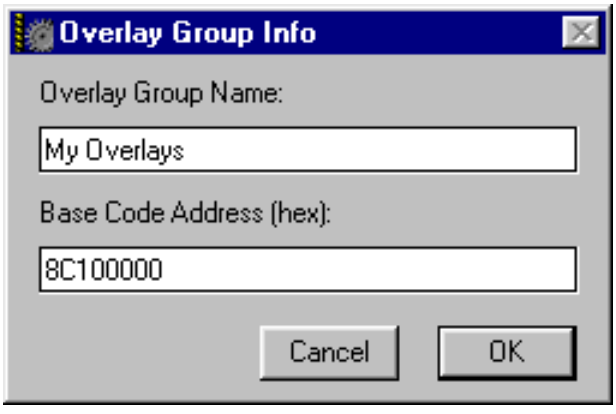
Select **Project > Create Overlay Group**. You should see a dialog as shown in Figure 14.3 . Type in the name `My Overlays` and click **OK**. This is the name of the new overlay group.

Figure 14.3 Create Overlay Group Dialog



You must now set the base code address for this overlay group. To set the base address, double-click the `My Overlays` group. You should see a dialog like the one in Figure 14.4 .

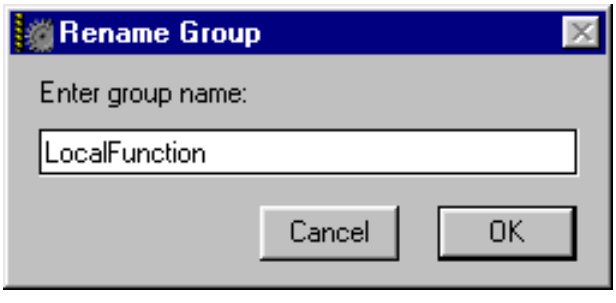
Figure 14.4 Setting the Base Code Address for an Overlay Group



The overlay code in the group `My Overlays` is loaded at the hex address entered in the **Base Code Address** field. For this example, enter `8C100000`. We chose this address because we know it is well above any memory space used by the rest of our application.

In the project window, expand the `My Overlays` group. You should see an overlay named `New Overlay`. Double-click the name or icon associated with this overlay to open the **Rename Group** dialog like the one in Figure 14.5 . Enter the name `LocalFunction` into the text field and click **OK**.

Figure 14.5 Rename Group (Overlay) Dialog



NOTE The name of an overlay is also the filename of the overlay as written to disk. In our source code, we load overlays into memory by their filename.

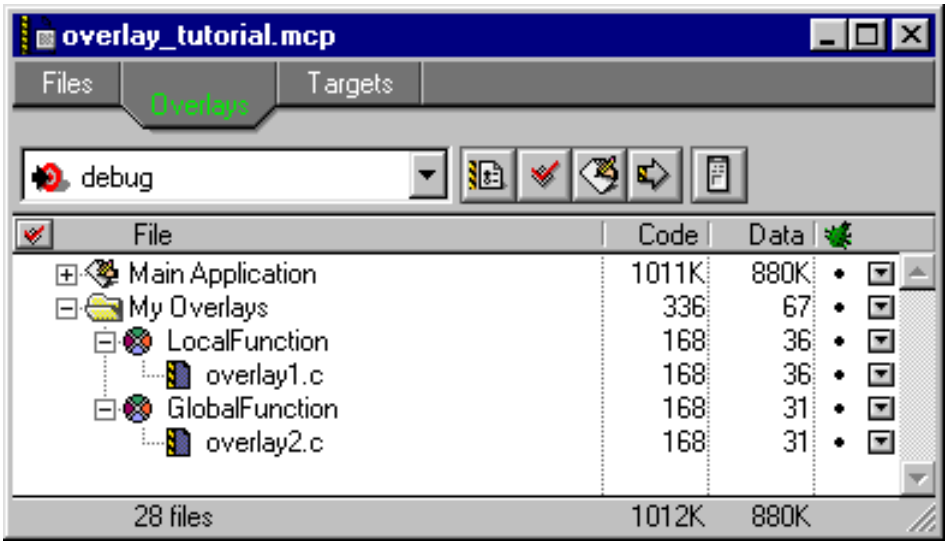
In the project window's file list, place the `Overlay1.c` source file into the `GlobalFunction` overlay by dragging the `Overlay1.c` icon to a position underneath the `LocalFunction` icon.

Now we need to create the second overlay of the `My Overlays` group. Click `My Overlays` in the project window and then choose **Project > Create New Overlay**. Enter the name `GlobalFunction` in the text field and click **OK**. Expand the new `GlobalFunction` overlay and drag the `overlay2.c` icon underneath the `GlobalFunction` icon in the file list.

d. You have successfully set up the overlays for this project.

Your project window should resemble the one in Figure 14.6 .

Figure 14.6 Project Window with Overlays



2. Set Up Other Project Settings

Now set the other project settings to prepare for building our application.

Open the **Edit > debug Settings > SH Target** panel. Change the **File Name** to `overlay.elf`. Open the **Target Settings > Target Settings** panel. Change the **Pre-linker** setting to **SH LCF Generator**. Since you created the overlay project using the stationery, you should not have to modify any other settings.

Save and close the target settings window.

3. Modifying Source Code for Overlays

We have to modify the source code to compile our code.

a. Inspect the overlay source files.

Click the **Files** tab in the Project window to view all the source files in the project. Double-click the `Overlay1.c` file to open it. It contains code for a simple arithmetic function named `func_ol1`. Close the file.

Open `Overlay2.c`. This file is slightly more interesting because it uses a global variable named `gVar`. The function `func_ol2` modifies this global variable, which is declared in `test.c`.

b. Edit test.c to work with overlays.

The code in your `test.c` will not compile. You must change a few constants to compiler and run the program.

Your code has to refer to the overlay by its position in the project's overlay list. You can determine an overlay's position by clicking the **Overlay** tab in the **Project** window. The overlays are `LocalFunction` and `GlobalFunction`, and they appear in that order. Thus, the index of `LocalFunction` is `0`, and the index of `GlobalFunction` is `1`.

In `test.c`, find the constant `OVERLAY1_NUMBER`. Change the placekeeper `***Index of first overlay***` to `0`. Change the placekeeper for the second overlay to `1`.

We now need to change the filenames of the overlays to match what you entered in the project. Find the definition of `OVERLAY1_FILENAME` and change it to:

```
#define OVERLAY1_FILENAME    "LOCALFUNCTION."
```

Change the definition of `OVERLAY2_FILENAME` to:

```
#define OVERLAY2_FILENAME    "GLOBALFUNCTION."
```

NOTE All filenames must be entered in uppercase letters. In addition, filenames without filename extensions must end with a trailing period ('.') character.

4. Examine how the overlays are loaded.

Down in `njUserMain()`, we call `func_ol1()` and `func_ol2()` in a loop. To call `func_ol1()`, we load its overlay file into memory using the `MWLoadOverlay()` utility function.

The address argument to `MWLoadOverlay()` is expressed as an offset from a table that the linker generates for you and includes in the main application. The name of this table is `overlay_section_address`. Each entry in this table corresponds to an overlay's base code address entered in an Overlay Group Info dialog as shown in Figure. There is an entry in this table for every overlay. The first entry in the table, index 0, always refers to the main application.

We check the return value of `MWLoadOverlay()` to see if the code successfully loaded. If so, we are safe to call the functions defined in that overlay code. Once we load the overlay, we also call the function `MWNotifyOverlayLoaded()`. This stub function tells the debugger that an overlay has been loaded and allows the debugger to make the adjustments necessary to debug this overlay. For more information on these functions, see "Metrowerks Utility Library" on page 149.

5. Compile the code.

Select **Project > Make** to compile the overlay tutorial program.

6. Test the overlays.

a. Write the overlay files to the GD-ROM emulator.

Before we can read our overlay files into memory, we must first write them to the GD-ROM. Launch the GDWorkshop, and create a GD-ROM image consisting of a dummy file and our two overlay files, `LocalFunction` and `GlobalFunction`.

Then activate the emulator by clicking the **Open/Close CD** button to close the simulated GD-ROM door.

NOTE To ensure that we meet the minimum length requirements for the GD-ROM, the dummy files we use must be at least 800 Kb in size. We need two dummy files for the first session, and one dummy file for the data section. The `gdworkshop.exe` program itself makes an excellent dummy file.

b. Start the debugger.

Select **Project > Enable Debugger**, and then choose **Project > Debug**. This downloads and runs our program on the Dreamcast. On your development computer, you should see the CodeWarrior debugger launch and stop at the beginning of `main()`.

c. Step through the program.

Single **Step** through the program until you come to `result=func_ol1(gCount)`. At this point, the `LocalFunction` overlay file has been loaded. **Step Into** this call to see the overlay code displayed in the Debugger. Step through and back out to `njUserMain()`. Notice the variable **`inVar`** now has a value of `1`.

Step through to the call to `func_ol2()`. If you step through this call, you should notice that the value of the global variable **`gVar`** is now `10`.

7. You have successfully built and tested a program using overlays.

Overlay Notes

These are various technical notes regarding how overlays are created and used. They include:

- Overlays and Exceptions
- Overlay Header
- GDWorkshop

Overlays and Exceptions

C++ exceptions are not supported when you use overlays.

Overlay Header

Listing 14.1 shows the format of the overlay header record that appears at the beginning of every overlay file. The debugger reads this 64 byte long header to identify the overlay. when the SH LCF generator is active, the overlay headers for your project are created automatically.

Listing 14.1 CodeWarrior Overlay Record Format

```
typedef struct overlayHeader
{
char          flag[3];      /* 'MWo' */
char          version;
unsigned long overlayID;    /* Same ID found in DWARF */
unsigned long loadAddress;  /* Address where to load the overlay*/
unsigned long TextSize;     /* Size of the executable part */
unsigned long DataSize;     /* Size of the data part */
unsigned long StaticInit;   /* Static init pointer */
unsigned long bssSize;      /* unused */
unsigned long entryPoint;   /* unused */
char          overlayName[32];
} OverlayHeader;
```

GDWorkshop

After recompiling any code that deals with overlays, you must replace the old copies of the files in the GD-ROM emulator with the newer copies.

If you do not replace the old files with the new files, you will encounter difficulties with your program, the CodeWarrior IDE, or both.

15

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:

Metrowerks Utility Library

To make programming for the Dreamcast as simple as possible, Metrowerks has written `mw_utils.c`. This library contains functions for loading and initializing overlays.

Using `mw_utils.c` requires that you:

1. add the `mw_utils.c` library source to your project
2. include the `mw_utils.h` header file in any file that calls a library function.

The functions in this library include:

- MWBload()
- MWNotifyOverlayLoaded()
- MWInitOverlay()
- MWLoadOverlay()

MWBload()

Description — Loads the entire contents of the named file into memory at the chose address.

Prototype
```
long MWBload(char *pfileName, void *address)
```

Parameters
```
pfileName - name of file to load
address   - memory address to place loaded data
```

Returns — Number of bytes read if successful (-1 otherwise).

MWNotifyOverlayLoaded()

Description — Informs the debugger that an overlay has been loaded.

Prototype
```
void MWNotifyOverlayLoaded
     (void *overlayLoadAddress)
```

Parameters
```
overlayLoadAddress - load address of the overlay
```

Remarks — `MWNotifyOverlayLoaded()` is a stub function that does not contain any actual code. However, when the debugger detects the presence of this function in your code, it will realize that an overlay has been loaded, read the header information at the start of the overlay, and restore the breakpoints for that overlay.

MWInitOverlay()

Description — Initializes memory for an overlay section.

Prototype
```
MWInitOverlay(void* address, signed long sizeByte)
```

Parameters
```
address  - memory address of loaded overlay
sizeByte - length of overlay in bytes
```

Remarks — Call this function after loading an overlay with `MWBload()`. This invalidates the CPU cache, clears the bss section, and calls the static initializer for the overlay.

MWLoadOverlay()

Description — Wraps `MWbload()`, `MWInitOverlay()`, and `MWNotifyOverlayLoaded()` into a single function.

Prototype
```
MWLoadOverlay(char* pFileName, void* address)
```

Parameters
```
pFileName - name of overlay file to load
address   - memory address to place loaded overlay
```

Returns `true` if successful.

Remarks This function strings together the functions commonly in loading an overlay. It loads the named overlay file into memory, initializes the overlay, and notifies the debugger that an overlay has been loaded.

Runtime Libraries

You may need to include the following runtime libraries in your project. The are located in the `Dreamcast Support` folder.

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 the `mw_pr ()` string printing function:

```
'mw output.lib'
```

Allocating Memory and Heaps

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.

16

Command Line Tools

CodeWarrior for Dreamcast includes a command line compiler, assembler, and linker for programmers who prefer to use command line tools. This chapter describes how to use the CodeWarrior for Dreamcast command line compiler and linker to build applications.

The topics in this chapter are:

NOTE Please read the Command LIne Tools Release Notes before using the tools.

Differences between Command Line Tools and IDE

The IDE-hosted tools and the command line tools differ in capability. These differences are as follows:

Overlay Support

You can not create projects with overlays using the command line tools.

Linker Command File Generator

The command line tools do not automatically create the `.lcf` linker command file. You must write your own `.lcf` file to link your project.

Locating the Command Line Tools

The command line tools are a set of three executable files:

- `mwccshx.exe`—Dreamcast compiler
- `mwldshx.exe`—Dreamcast linker
- `mwasmshx.exe`—Dreamcast assembler

These tools are located in the folder named `CodeWarrior\Tools\Command Line Tools`.

Command Line Switches

Under the IDE, linker settings and project settings are configured using preference panels. With command line tools, these settings are set according to switches and options you give on the command line.

For a complete list of command line switches for any tool component, use the `-help` option. For example, to obtain a complete list of switches for the command line tool compiler, you would type:

```
mwccshx -help
```

When using command line tools, you specify compiler and linker settings manually. In general, you need to use the following switches and options to compile Dreamcast applications.

Switches for the `mwasmshx` Assembler

```
-little
```

Switches for the mwccshx Compiler

```
-prefix prefix_dc.h
-inline off
-g
-v
-little
-ansi off
-ARM off
-bool off
-strict off
-wchar_t off
-proc SH4
-heapsize 32768
-stacksize 32768
-fp hard
-main SG_SEC
-map
```

Setting Up Environment Variables

Several environment variables are used at runtime to search for system paths and libraries. These variables can shorten the command lines for many tasks. All of the variables mentioned below are lists which are separated by semicolons (';').

NOTE It is not necessary to include quote marks when defining environment variables that include spaces. Windows will not strip out the quotes—leaving them in leads to "unknown directory" warnings in the command line tool. Use the following syntax when defining variables in batch files or at the command line:

```
set Folders=C:\First Path\Foo;D:\Second Path\Bar
```

C/C++ Compiler Variables

`MWCIncludes`—the named paths are added to the system path. Typically you might define this variable as follows:

```
set MWCIncludes=
CodeWarrior\Dreamcast Support\INCLUDE\;
```

```
CodeWarrior\Dreamcast Support\Shinobi\Lib\;
CodeWarrior\Dreamcast Support\Shc\INCLUDE\;
CodeWarrior\Dreamcast Support\Shinobi\INCLUDE\;
CodeWarrior\Dreamcast Support\Runtime\Runtime DC
```

Linker Variables

`MWLibraries`—the named paths are added to the system path. Typically, you might define this variable as follows:

```
set MWLibraries=
CodeWarrior\Dreamcast Support\Shinobi\Lib\;
CodeWarrior\Dreamcast Support\Runtime\Runtime DC
```

`MWLibraryFiles`—the named files are added to the end of the link order. Typically, you might define this variable as follows:

```
set MWLibraryFiles=ninja.elf.lib;
Shinobi.elf.lib;sh4nlfzz.elf.lib;
Nindows.elf.lib;MSLRuntimeDC.LIB
```

Compiling and Linking

The compiler invokes the linker automatically. The link order is determined by the order in which files are listed on the command line. Keep in mind that you still need a valid linker command file to link your code.

We have included two examples to illustrate the usage of the command line tools. These may be found in the folder named:

```
Examples\Command Line Tools\
```

For the interest of simplicity, assume for a moment that all of the libraries and source code files used by the `teapot` example are located in the same folder as the command line tools themselves (this allows us to present an example without long path names). If this were the case, we could create a batch file with the following commands to compile and link our project into an executable named `teamake1.elf`.

```
mwasmshx -little global32_cw.src
```

```
mwccshx -prefix prefix_dc.h -O4,p -inline off
-g -little -ansi off -ARM off -bool off
-strict off -wchar_t off -proc SH4
-heapsize 32768 -stacksize 32768 -fp hard
-main SG_SEC -v -o teamake1.elf -map
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 ninja.elf.lib Shinobi.elf.lib
sh4nlfzz.elf.lib Nindows.elf.lib MSLRuntimeDC.LIB
model.c njloop.c sbinit.c t009.c test.c
global32_cw.o linker.lcf
```

17

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

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 en counter 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.

Index

Symbols

A

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

CodeWarrior

Targeting Dreamcast

Credits

writing lead: Roger Wong

engineering: Aaron Smith, Guohua Cao, Laurent Visconti, Nick Havens, Shoji Ueda, Takashi Kashima, Toshiaki Koasa, and Xin Li

frontline warriors: David Wilson, and CodeWarrior users everywhere