

**CodeWarrior™
Development Studio
for
Nintendo DS™
Targeting Manual
Version 1.2**

Metrowerks and the Metrowerks logo are registered trademarks of Metrowerks Corporation in the United States and/or other countries. CodeWarrior is a trademark or registered trademark of Metrowerks Corporation in the United States and/or other countries. ARM, Thumb, ARM Powered and StrongARM are registered trademarks of ARM Limited. ARM7 is a trademark of ARM Limited. All other trade names and trademarks are the property of their respective owners.

Copyright © 2004 Metrowerks Corporation. ALL RIGHTS RESERVED.

No portion of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, without prior written permission from Metrowerks. Use of this document and related materials are governed by the license agreement that accompanied the product to which this manual pertains. This document may be printed for non-commercial personal use only in accordance with the aforementioned license agreement. If you do not have a copy of the license agreement, contact your Metrowerks representative or call 1-800-377-5416 (if outside the U.S., call +1-512-996-5300).

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

How to Contact Metrowerks

Corporate Headquarters	Metrowerks Corporation 7700 West Parmer Lane Austin, TX 78729 U.S.A.
World Wide Web	http://www.metrowerks.com
Technical Support	E-mail: support@noa.com

Table of Contents

1	Introduction	9
	Read the Developer Notes	9
	Documentation Overview	9
	Adobe Acrobat PDF Files	9
	Microsoft HTML Help CHM Files	10
	Context-sensitive Help	10
	Other Resources	10
2	Getting Started	11
	Installing Nintendo DS Development Tools	11
	Uninstalling the CodeWarrior IDE	13
	CodeWarrior Development Tools	14
	Overview of the CodeWarrior IDE	14
	CodeWarrior Debugger	14
	CodeWarrior Development Process	15
	Projects	15
	Editing	16
	Compiling	16
	Linking	16
	Debugging	17
	Viewing Preprocessor Output	17
	Checking Syntax	17
	Disassembling	17
3	Tutorials	19
	Converting a Makefile-based Project to a CodeWarrior Project	19
	Debugging a Project	25
	Creating Release and ROM Output Files	27
4	Target Settings	29
	Target Settings Overview	29
	Opening the Target Settings Window	29

Table of Contents

Default Settings for Stationery	30
Target Settings Panels	31
ARM Project Panel	32
ARM Assembler Panel	33
ARM Processor Panel	36
Binary Converter Settings Panel	42
Nintendo CodeGen Panel	43
BatchRunner PostLinker Panel	47
BatchRunner Preprocessor Panel	48
ELF Disassembler Panel	49
ARM Linker Panel	53
NITRO LCF Prelinker Panel	58
Nitro MakeRom Postlinker Panel	60
ARM Debugger Settings Panel	62
NITRO Debugger Setting Panel	67
5 Debugging Procedures and Utilities	69
Configuring a Project for Debugging	70
Debugger Settings Panel	70
Remote Debugging Panel	72
ARM/Thumb Mode and Disassembly	73
Load/Save/Fill Memory Operations	73
Load/Save Memory	74
Fill Memory	76
Debugging Stand-alone ELF (.nef) Files	80
Multi-Core Debugging Options	80
Displaying the ARM7 Thread Window	80
Stopping and Running Simultaneous Processes	81
Cache Viewer	81
System Browser (Processor and Thread Browser)	83
Using the Profiler	84
Using Watchpoints	85
Activating DS Flash Card and GBA Cartridge Slots	86
Writing Emulator Memory to DS Flash Card	86
Writing User Contents to Backup Device	87

View ROM Header	88
One Time PROM	89
6 Assembler	91
Label Syntax	91
Debugging Assembly Source	92
.function	92
.line	92
.file	93
GNU Assembler Compatibility	93
7 C and C++ Compiler	99
Number Formats	100
Integer Formats	100
Floating-Point Formats	101
Declaration Specifiers	101
Register Variables	102
Pragmas	102
allow_byte	104
ASHLA	104
avoid_byte	105
dead_stripping	105
define_section	105
generic_symbol_names	106
interworking	107
little_endian	108
profile	108
section	108
thumb	108
warn_byte	108
Attribute Syntax	109
__attribute__((aligned(<i>item</i>)))	109
Typedef Declaration Examples	110
Struct Member Examples	110
__attribute__((interrupt (<i>flagname</i>)))	111

Table of Contents

8	Linker Issues	113
	Linker Command File Prelinker	113
	Activating the Prelinker	113
	Project Settings Read by the Prelinker	113
	Prelinker Defaults	114
	Limitations	114
	Overlay Support	114
	Binary Converter/Compiler	115
	Link Order	116
	Deadstripping Unused Code and Data	117
9	Linker Command File Specification	119
	Structure of Linker Command Files	119
	Memory Segment	120
	Sections Segment	120
	Closure Blocks	121
	Linker Command File Syntax	122
	Alignment	123
	Arithmetic Operations	123
	Comments	124
	Deadstrip Prevention	124
	Exception Tables	124
	Expressions, Variables and Integral Types	125
	File Selection	126
	Function Selection	127
	Stack and Heap	127
	Static Initializers	128
	Writing Data to Memory	128
	Writing Data to Memory from a File	129
	Alphabetical Keyword Listing	131
	. (location counter)	132
	ADDR	132
	ALIGN	133
	ALIGNALL	133

EXCEPTION	134
FORCE_ACTIVE	134
GROUP	135
INCLUDE	135
KEEP_SECTION	136
LITERAL	136
MEMORY	136
OBJECT	138
OVERLAYID	138
REF_INCLUDE	138
SECTIONS	139
SIZEOF	140
SIZEOF_ROM	140
WRITEB	141
WRITEH	141
WRITES	141
WRITEW	142
 10 Inline Assembly and Intrinsic Functions	 143
Inline Assembly	143
Inline Assembly Syntax	144
Labels	147
Comments	147
Preprocessor Comments and Macros	147
Stack Frame	148
Specifying Operands	149
Using Local Variables and Arguments	153
Intrinsic Functions	155
Buffer Manipulation	155
Pseudo-Instructions	155
DCD	156
LDA	156
LDCONST	156
Optimization Control Directives	157

Table of Contents

11 Overlays	159
Defining Overlay Structures	159
Creating Overlay Groups in the Project Window	159
Creating the Linker Command File with the Prelinker	161
Creating Overlay Structures Using the Command Line Linker	162
Overlay Management	162
Determining Valid Intersegment Calls	163
12 Libraries	167
Runtime Libraries	167
Mandatory Floating Point Libraries	167
MSL C/C++	169
MSL Extras	169
A Command-Line Tools	171
bintoelf.exe	171
elftobin.exe	172
mwasmarm.exe	173
mwccarm.exe	173
mwldarm.exe	175
Project Options	175
Linker Options	176
Linker CodeGen Options	177
ELF Disassembler Options	179
Index	183

Introduction

This manual describes how to use the CodeWarrior™ Integrated Development Environment (IDE) to develop software for the Nintendo DS™ platform.

This chapter has these sections:

- [Read the Developer Notes](#)
- [Documentation Overview](#)
- [Other Resources](#)

Read the Developer Notes

If you have not read the CodeWarrior release notes, please do so now. They contain important information about new features, bug fixes, and incompatibilities. You can find them on the CodeWarrior CD-ROM, in the Release Notes folder.

Documentation Overview

We provides documentation in these formats:

- [Adobe Acrobat PDF Files](#)
- [Microsoft HTML Help CHM Files](#)
- [Context-sensitive Help](#)

Adobe Acrobat PDF Files

The CodeWarrior documentation in PDF format is in

CodeWarrior\Help\PDF

where *CodeWarrior* is the path to your CodeWarrior installation.

To view these documents, you must have the Adobe® Acrobat® Reader™ software. You can download the software from this URL:

<http://www.adobe.com/acrobat>

Microsoft HTML Help CHM Files

On Windows systems, the CodeWarrior documentation in Microsoft HTML Help CHM format is in

CodeWarrior\Help

where *CodeWarrior* is the path to your CodeWarrior installation.

To view these documents, start the CodeWarrior IDE and select **Help > Online Manuals**.

Context-sensitive Help

There is context-sensitive help for many of the options in the CodeWarrior fIDE windows.

To use context-sensitive help, perform one of these tasks:

- Click the question-mark symbol at the top of a window, then click the item
- Right-click the item to open a contextual menu, then select **What's This?** from the menu

Other Resources

This manual does not discuss how to program for the Nintendo DS platform or how to use the Nintendo DS runtime libraries. These topics are covered in the documentation that came with your Nintendo DS development kit. Other programming resources that you may be interested in are listed below:

Nintendo Software Development Support Group

<http://www.warioworld.com>

Nintendo SDSG News Server (Nintendo SDSG account required)

NNTP Address: news.sdsg.nintendo.com

NNTP Port: 563

Application Binary Interface (ABI) for the ARM Architecture.

<http://www.arm.com/products/DevTools/ABI.html>

ARM Architecture Reference Manual (ISBN: 0201737191)

Getting Started

This chapter explains how to install the CodeWarrior™ IDE and provides an overview of the tools. This chapter has these sections:

- [Installing Nintendo DS Development Tools](#)
- [Uninstalling the CodeWarrior IDE](#)
- [CodeWarrior Development Tools](#)
- [CodeWarrior Development Process](#)

Installing Nintendo DS Development Tools

- [Install Nintendo Nintendo DS SDK](#)
- [Install IS-NITRO-DEBUGGER](#)
- [Install CodeWarrior Development Tools](#)
- [Install Nintendo Ensata Emulator](#)

Install Nintendo Nintendo DS SDK

Following Nintendo's installation instructions, install the SDK to an appropriate folder on your hard drive. For example, C:\NitroSDK

In previous releases, you had to create an environment variable NITROSDK_ROOT that pointed to this SDK folder. Although the new CodeWarrior installer now creates this variable for you, you will still have to create it yourself if you install an older release:

1. Open the Windows Control Panel
2. Double-click **System**
3. Select **Advanced** tab
4. Click **Environment Variables** button
5. Go to the **System Variables** section, and click the **New** button.
6. Create variable named NITROSDK_ROOT with value C:\NitroSDK
7. Click **OK** button as many times as necessary to return to Windows.

CAUTION You must reboot your system after creating this environment variable.

Install IS-NITRO-DEBUGGER

Following Nintendo's installation instructions, install the IS-NITRO-DEBUGGER software package. The latest version of this software package is located on the Warioware Software Development Support Group website:

<https://www.warioworld.com/nitro/>

CAUTION You must reboot your system after you install the IS-NITRO-DEBUGGER software package, even if the installer does not prompt you to reboot.

Install CodeWarrior Development Tools

Before installing this product, you must uninstall any previous version of the product on your computer. To uninstall a previous version of this product, refer to [“Uninstalling the CodeWarrior IDE”](#).

NOTE Only a user with Administrator privileges may install the CodeWarrior Development Tools for Nintendo DS. If other users require access to these tools, you must grant them full rights to the Cross_Tools directory in the CodeWarrior IDE installation.

TIP If you are installing the CodeWarrior IDE on a computer with multiple operating systems, the installation process affects the currently running operating system. As you start the installation process, be sure you are running the operating system on which you plan to use the CodeWarrior IDE.

1. Run the CodeWarrior installer.
 - a. Double-click the drive letter corresponding to the drive on your computer that contains the CodeWarrior CD-ROM. An installation menu appears.

NOTE If you disabled auto-run, you need to manually launch the installation program `Launch.exe` on the CD-ROM in order to see the installation menu.

- b. Click **Launch the installer**. The installation wizard runs and guides you through the setup process.
 - c. Follow the on-screen instructions in the installation program until the **Select NITRO SDK Location** page appears.
 - d. If the installer is unable to locate an installation of the NITRO SDK on your system, click the **Browse** button to locate the NITRO SDK folder yourself.

- e. Click **Next** button to continue the installation process.
- f. Follow the remaining on-screen instructions in the installation program.
2. Reboot computer - upon reboot, the CodeWarrior IDE installer completes the installation procedure.

Install Nintendo Ensata Emulator

The latest version of this software package is located on the Warioware Software Development Support Group website:

<https://www.warioworld.com/nitro/>

1. Following Nintendo's installation instructions, install the Ensata Emulator to an appropriate folder on your hard drive.
2. Locate and open the CodeWarrior Ensata debugger initialization file `est_cw_debugger.ini` in a text editor. This file is located in `{CodeWarrior}\bin\Plugins\Support\Nitro\IS`
3. Change the variable `ensata_path` to that of your ensata executable. For example:

```
[control]
ensata_path=C:\NitroSDK\ensata\Release\ensata.exe
```

4. Save the file back to disk.

Uninstalling the CodeWarrior IDE

You should uninstall the CodeWarrior IDE before you upgrade to a newer release. Follow these steps to uninstall the CodeWarrior IDE:

1. Open the Windows **Control Panel**.
2. Run **Add or Remove Programs** from the Windows Control Panel.
3. Select the CodeWarrior installation from the list of installed programs.
4. Click **Change/Remove**.
5. Follow the on-screen instructions to complete the uninstall process.

CAUTION	If you have multiple CodeWarrior IDE installations and you uninstall one of them, the process unregisters dynamically linked library (DLL) files shared among the remaining installations. This process can cause the remaining installations to stop working.
----------------	--

To restore the functionality of the remaining installations, you need to

run the `regservers.bat` file. This file resides in the `\bin` directory of each remaining installation. Run one of the remaining `regservers.bat` files to re-register the shared DLL files.

CodeWarrior Development Tools

Programming for the Nintendo DS platform is much like programming for any other target platform in CodeWarrior IDE. If you have never used the CodeWarrior IDE, then you should read these sections:

- [Overview of the CodeWarrior IDE](#)
- [CodeWarrior Debugger](#)

Overview of the CodeWarrior IDE

The CodeWarrior IDE helps you write, compile, and debug your software. The CodeWarrior IDE has a project manager, source-code editor, compilers, linkers, and a debugger.

The project manager may be new to those more familiar with command-line development tools. The project manager organizes all files and settings related to your project. It lets you see the contents of your project at a glance, and simplifies navigation among your source-code files.

A project can contain multiple build targets. A *build target* is a separate build (with its own settings) that uses some or all of the files in the project. For example, you can have a debug version and a release version of your software as separate build targets in the same project.

For more information about how the CodeWarrior IDE compares to a command-line environment, see “[CodeWarrior Development Process](#)”. That section explains how various parts of the CodeWarrior IDE implement the features of a command-line development system based on makefiles.

The CodeWarrior IDE has an extensible architecture that uses plug-in compilers and linkers to target various operating systems and microprocessors.

For more information about the CodeWarrior IDE, refer to the *IDE User's Guide*.

CodeWarrior Debugger

The CodeWarrior debugger controls the execution of your program and allows you to see what is happening internally as your program runs.

You use the debugger to find problems in your program. The debugger can execute your program one statement at a time and suspend execution when control reaches a specified

point. When the debugger stops a program, you can view the chain of function calls and examine or change the values of variables and registers.

For general information about the debugger, including its basic features and user interface, refer to the *IDE User's Guide*.

CodeWarrior Development Process

While working with the CodeWarrior IDE, you will proceed through the development stages familiar to all programmers: writing code, compiling and linking code, and debugging code. For complete information on performing tasks like editing, compiling, debugging, and linking code, refer to the *IDE User's Guide*.

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

- [Projects](#)
- [Editing](#)
- [Compiling](#)
- [Linking](#)
- [Debugging](#)
- [Viewing Preprocessor Output](#)
- [Checking Syntax](#)
- [Disassembling](#)

Projects

The CodeWarrior *project* is analogous to a **makefile**, or a collection of makefiles. A CodeWarrior project can contain multiple *build targets*. For example, you can configure a project to build both a debug version and a release version of your executable file.

A major difference between the CodeWarrior IDE and make is that make works backwards from object files to source-code files (*backward chaining*). In contrast, the CodeWarrior IDE works forward from source-code files to object files (*forward chaining*).

Another major difference is that make defines each step of the build process (such as source to object, object to library, library to executable file) and there may be an arbitrary number of steps during a build. By contrast, the CodeWarrior IDE uses a fixed build model for each target: build sub-targets, precompile, compile, pre-link, link, and post-link.

Getting Started

CodeWarrior Development Process

The CodeWarrior IDE lists all the project's files in the project window. The input files include source-code files, object-code files, libraries, scripts, and sub-project files. You might also find header files and documentation files in a project, so that you can find all files in one convenient place. The IDE ignores extraneous files during the build process.

The CodeWarrior IDE also lets you add source-code files with unsupported filename extensions to your project. You can use the CodeWarrior IDE to associate the unsupported filename extensions to a CodeWarrior plug-in compiler. For details, refer to the *IDE User's Guide*.

You can add or remove a project's files. You can assign files to one or more different targets in the project, to help you simplify managing files common to multiple targets.

The CodeWarrior IDE manages all the dependencies between files automatically, and tracks which files changed since the last build. When you rebuild, the IDE recompiles only those files that changed.

Editing

The CodeWarrior IDE has an integrated text editor. It reads and writes text files in UNIX, Mac OS, and MS-DOS/Windows formats.

To edit a source-code file, or any other text file in a project, just double-click the file's name in the project window.

The editor window has navigational (code browsing) features that let you switch between related files, locate a particular function, mark a location in a file, or go to a specific line of code.

Compiling

To compile a source-code file, it must be among the files in the active build target. If it is, you select its name in the project window and select **Project > Compile**.

To compile all files in the active build target that you modified since the last compile, select **Project > Bring Up To Date**.

In command-line environments, a binary file stores object code compiled from a source-code file. The CodeWarrior IDE stores and manages object files transparently.

You can also disassemble Extended Linker Format (ELF) files in the CodeWarrior IDE. The IDE generates a `.dump` file that has the object code from a disassembled `.elf` file.

Linking

To link object code into a final binary file, select **Project > Make**. This command brings the current project up to date, then links the resulting object code into a final output file.

You use the CodeWarrior IDE to control the linker. You do not need to specify a list of object files. The CodeWarrior IDE keeps track of all object files automatically. Arrange files in the **Overlays** page of the project window to control the order in which the IDE links them.

Debugging

To tell the compiler and linker to generate debugging information for all items in your project, make sure **Enable Debugger** is selected in the **Project** menu.

If you want to only generate debug information on a file-by-file basis, click in the debug column for that file. The Debug column is located in the project window, to the right of the Data column.

When you are ready to debug your project, choose **Debug** from the **Project** menu.

Viewing Preprocessor Output

To view preprocessor output, select the file's name in the project window and select **Project > Preprocess**. A new window opens and shows you what the preprocessed file looks like. You can use this feature to investigate bugs caused by macro expansion or other subtleties of the preprocessor.

Checking Syntax

To check the syntax of a file in your project, select the file's name in the project window and select **Project > Check Syntax**. If the IDE detects syntax or compilation errors in the selected file, a message window appears and shows information about the errors.

Disassembling

To disassemble a compiled file or an ELF file in your project, select the file's name in the project window and select **Project > Disassemble**. After disassembling the file, the CodeWarrior IDE creates a `.dump` file that contains the disassembled file's object code in DWARF (Debugging With Attribute Record Format). The `.dump` file's contents appear in a new window.

Getting Started

CodeWarrior Development Process

Tutorials

These tutorials demonstrate some of the features of the CodeWarrior IDE. In the first tutorial, we create and compile one of the sample projects from the Nintendo DS SDK. In the second tutorial, we load the project into the debugger and step through the program to demonstrate some of the features of the debugger.

- [Converting a Makefile-based Project to a CodeWarrior Project](#)
- [Debugging a Project](#)
- [Creating Release and ROM Output Files](#)

Converting a Makefile-based Project to a CodeWarrior Project

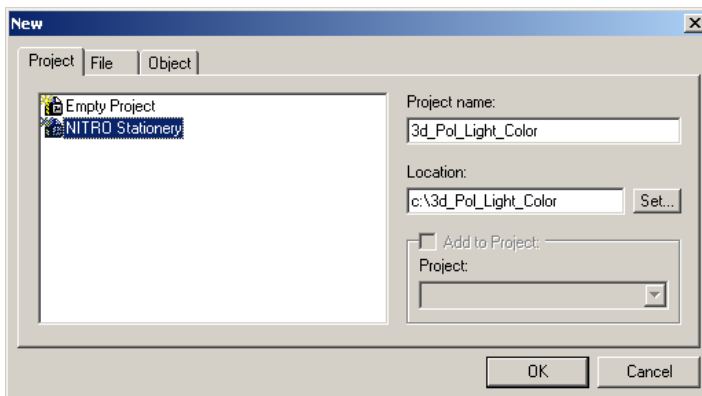
In this tutorial, we use the CodeWarrior IDE to create a project for and to compile one of the makefile samples included in the Nintendo DS SDK. We examine the makefile to identify the libraries and source files, then create a CodeWarrior project based on this information.

1. Launch the CodeWarrior IDE.
2. Create a new C application project from stationery.
 - a. Select **File > New - New** dialog box appears ([Figure 3.1](#))

Tutorials

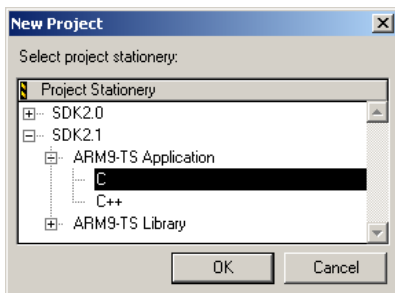
Converting a Makefile-based Project to a CodeWarrior Project

Figure 3.1 New dialog box



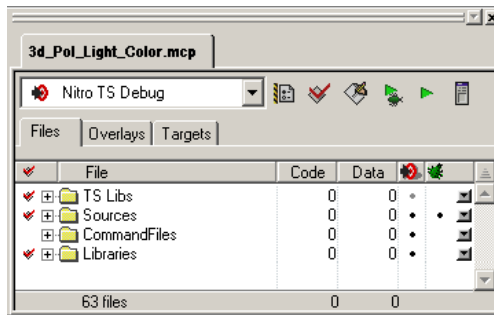
- b. Select **NITRO Stationery**
- c. For **Project Name**, enter 3d_Pol_Light_Color
- d. For **Location**, enter C:\3d_Pol_Light_Color
- e. Click **OK** button - **New Project** dialog box appears ([Figure 3.2](#))

Figure 3.2 New Project dialog box



- f. Select **SDK2.1 > ARM9-TS Application > C**
- g. Click **OK** button - IDE creates project; **3d_Pol_Light_Color.mcp** project window appears ([Figure 3.3](#))

Figure 3.3 3d_Pol_Light_Color.mcp project window



3. Examine the makefile to identify all source code and library components.

Although the project stationery comes preloaded with the standard SDK libraries, you must add special libraries yourself.

- a. Open the **makefile** file in the folder
 {NitroSDKFolder}\build\demos\gx\UnitTours\3D_Pol_LightColor
- b. Locate the line `LLIBRARIES = libDEMO.a`. This line indicates that we must add the `libDEMO.a` library file to our project.
- c. Locate the line `SRCS = main.c`. This line indicates that we must add the `main.c` source file to our project.

Figure 3.4 Gathering information from the Makefile

```
#-----
# Project: NitroSDK - GX - demos - UnitTours/3D_Pol_LightColor
# File:      Makefile
#-----

SUBDIRS      =
LINCLUDES    =      ../DEMOLib/include
LLIBRARY_DIRS =      ../DEMOLib/lib/$(NITRO_BUILDTYPE)
LLIBRARIES   =      libDEMO.a

#-----
SRCS         =      main.c
TARGET_BIN   =      main.srl
```

Library File

Source File

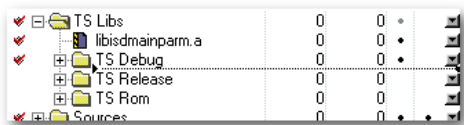
4. Add the source file to the project.
 - a. Locate the **main.c** file in the folder
 {NitroSDKFolder}\build\demos\gx\UnitTours\3D_Pol_LightColor\src
 - b. Copy this **main.c** file to your `C:\3d_Pol_Light_Color` folder, replacing the existing file.

Tutorials

Converting a Makefile-based Project to a CodeWarrior Project

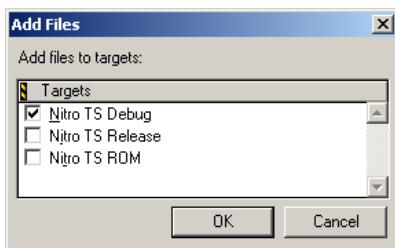
5. Add the library file and library path to the project.
 - a. Open the folder
{NitroSDKFolder}\build\demos\gx\UnitTours\DEMOLib\lib\ARM9-TS\Debug
 - b. Drag the **libDEMO.a** file to the IDE project window and drop it under the **TS Libs** > **TS Debug** file group ([Figure 3.5](#)) - **Add Files** dialog box appears

Figure 3.5 Drop library under the TS Debug file group



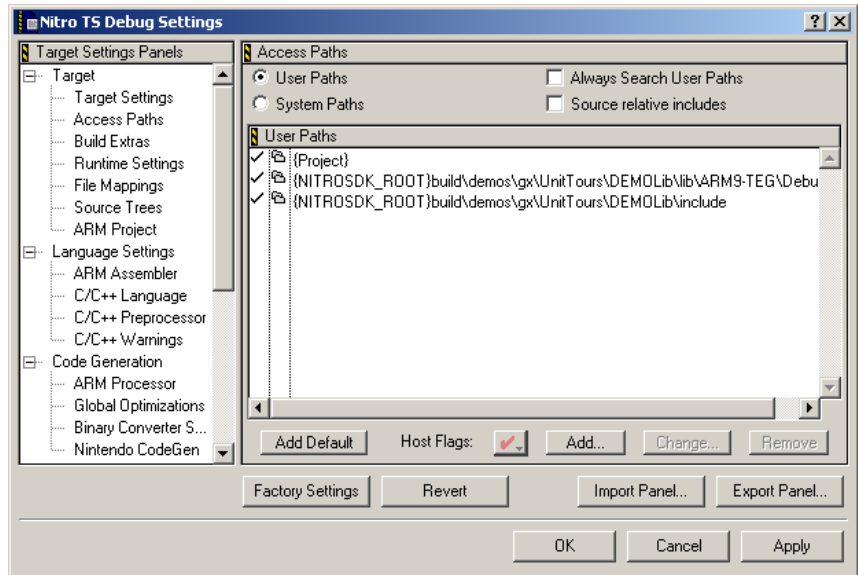
- c. Select only the **Nitro Debug** checkbox ([Figure 3.6](#))

Figure 3.6 Add Files dialog box



- d. Click **OK** button - **libDEMO.a** is added to the Nitro Debug target
 - e. Repeat steps c - f for the Release and ROM libraries, adding them to their respective Nitro targets.
 - f. Select **Edit > Nitro Debug Settings** - Target Settings window appears
 - g. Select the **Target > Access Paths** target settings panel ([Figure 3.7](#))

Figure 3.7 Access Paths target settings panel

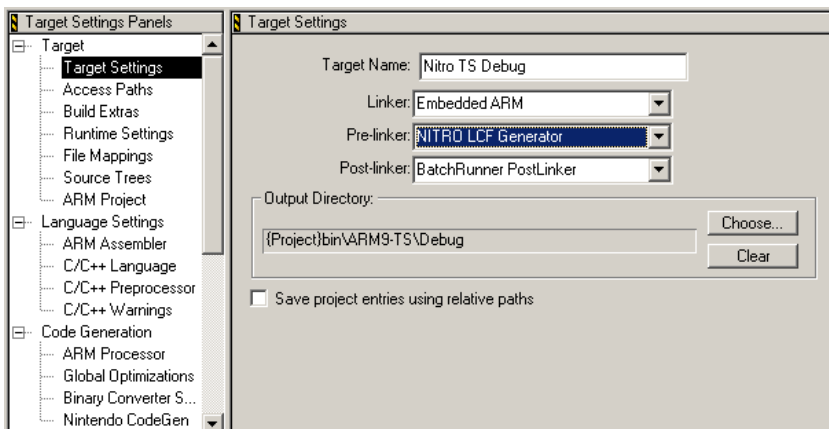


- h. Click the **Add** button - **Browse for Folder** dialog box appears
 - i. Set **Path Type** to NITROSDK_ROOT
 - j. Select the
`{NITROSDK_ROOT}build\demos\gx\UnitTours\DEMOLib\Include`
 folder
 - k. Click **OK** button - path is added.
6. Adjust the linker command file settings.
 - a. Select the **Target** > **Target Settings** panel appears
 - b. Set **Pre-linker** to **NITRO LCF Generator** ([Figure 3.8](#))

Tutorials

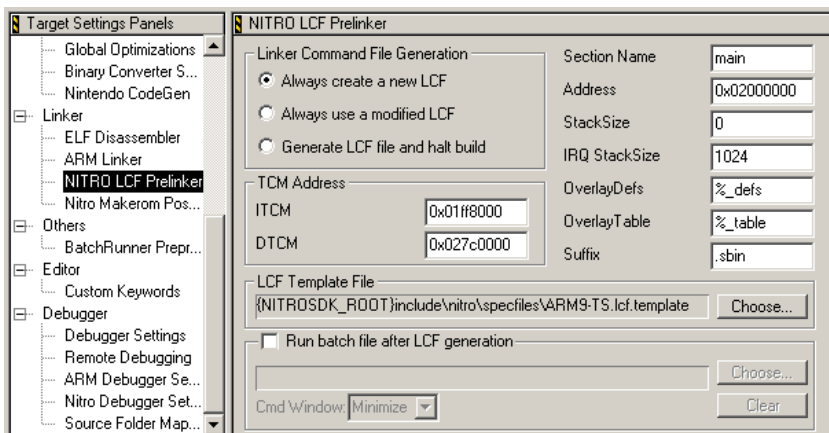
Converting a Makefile-based Project to a CodeWarrior Project

Figure 3.8 Nitro Debug Target Settings panels



- c. Select the **Linker > Nitro LCF Prelinker** settings panel
- d. Verify that **Address** is set at 0x02000000

Figure 3.9 NITRO LCF Prelinker target settings panel



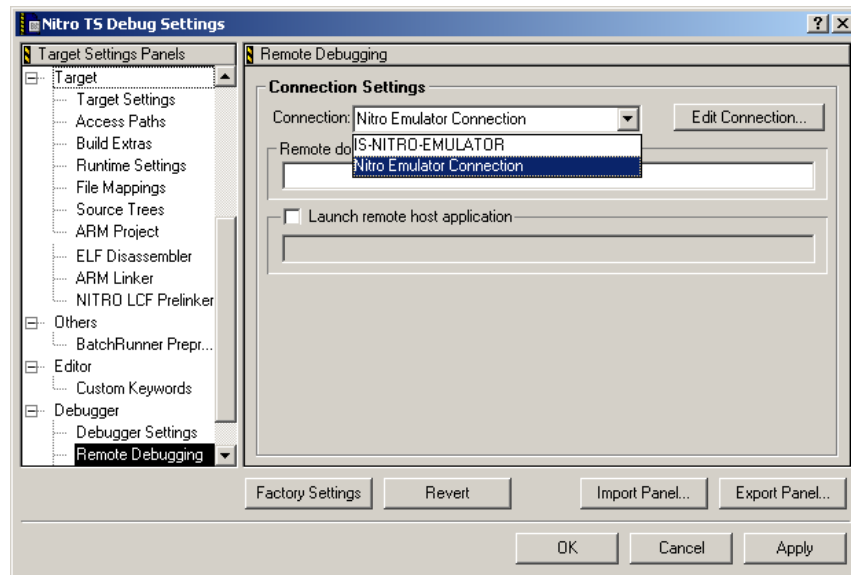
- e. Click **OK** button - target settings window closes
7. Build the project by selecting **Project > Make** - IDE builds the project.

Debugging a Project

In this tutorial, we configure the CodeWarrior IDE to load our sample project into the debugger, and we step through the program to demonstrate some of the debugger features. This quick tour introduces some important IDE elements used for debugging the Nintendo DS game platform. For a full description of the CodeWarrior IDE, refer to the *CodeWarrior IDE Users Guide*.

1. Set the debugger target to the emulator or to the hardware connection
 - a. Select **Edit > Nitro Debug Settings** - target settings window appears
 - b. Select the **Debugger > Remote Debugging** panel
 - c. Set the **Connection** to either **Nitro Emulator Connection** (software emulator) or **IS-NITRO-EMULATOR** (hardware target) ([Figure 3.10](#))

Figure 3.10 Remote Debugging panel

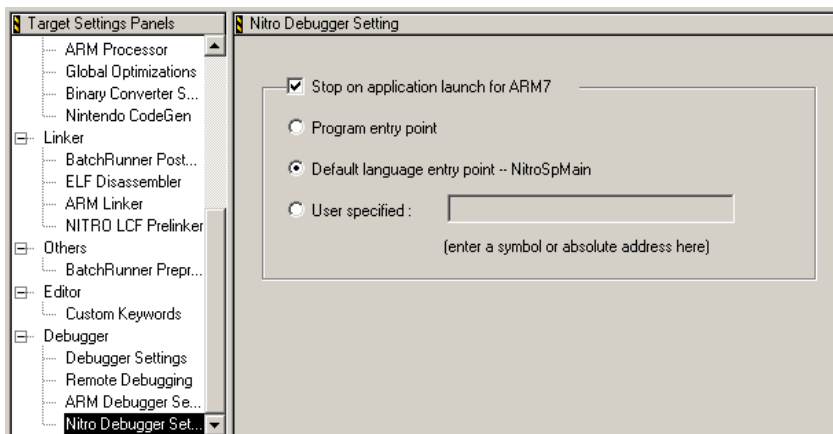


- d. If using the hardware target, select the **Debugger > Nitro Debugger Settings** panel and uncheck the **Stop on application** checkbox ([Figure 3.11](#)) - this hides the ARM7 thread window as this sample only uses the ARM9 core.

Tutorials

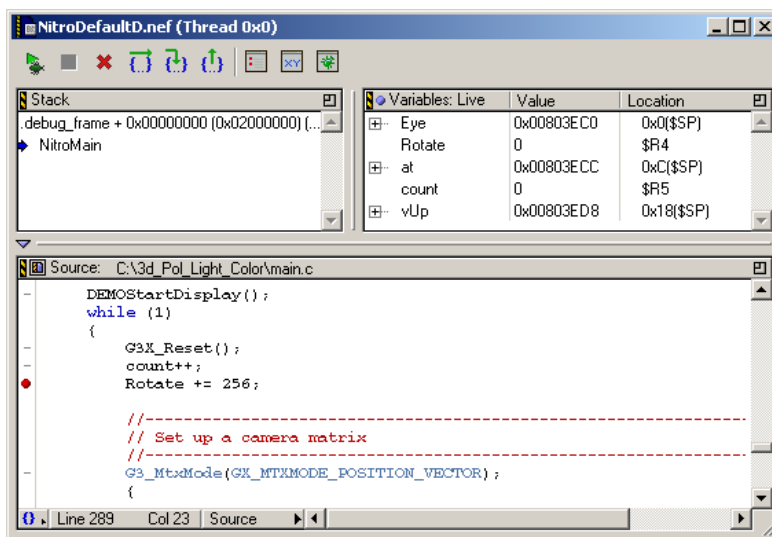
Debugging a Project

Figure 3.11 NITRO Debugger Settings panel



- e. Click **OK** button - Target Settings window closes
2. Start a debug session by selecting **Project > Debug** - thread window appears
3. Click in the left column ([Figure 3.12](#)) to set a breakpoint in the source code at the line that reads:
`Rotate += 256;`

Figure 3.12 Setting a breakpoint






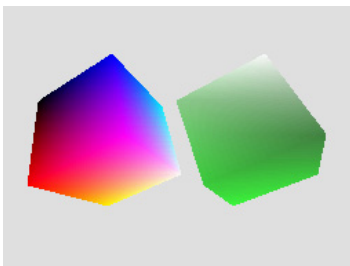
4. Click the Run button  in the thread window to run the program until the breakpoint - variables in the variables pane update.
5. Click the Step Over button  a few times to step through the source code.
6. Click the Run button  again to continue program execution - program output appears in either the LCD or the emulator display

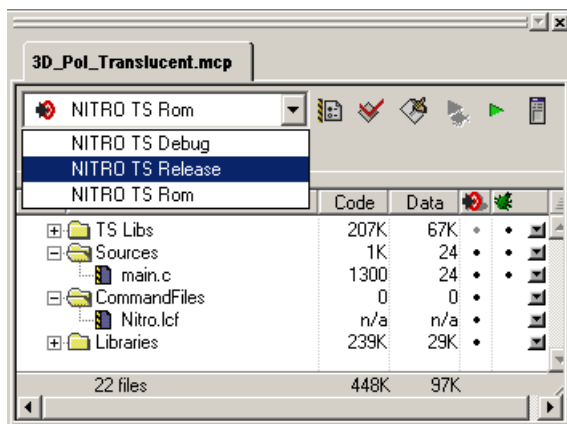
Figure 3.13 Program output



Creating Release and ROM Output Files

The stationery projects include targets for creating debug, release, and ROM versions of your project. Select your desired target from the target list and re-compile your project.

Figure 3.14 Selecting Release or ROM Targets



Tutorials

Creating Release and ROM Output Files

- **Debug**

Contains debugger symbolics and unoptimized code that lets you debug your project.

- **Release**

Increases compiler optimizations to reduce code size and execution time.

- **ROM**

Similar to Release target, but also removes debugger symbolics.

NOTE The release version of your output file is optimized. As such, the debugger may not be able to accurately map source code to the corresponding assembly code.

ROM versions lacks debugger symbolics; you can run them, but you cannot debug them.

Along with the `.nef` output file, each target is set by default to also creates a `.srl` ROM file according to the `.rsf` file specifications. For more information, see [“Nitro MakeRom Postlinker Panel”](#).

Target Settings

This chapter describes the target settings panels specific to projects targeting Nintendo DS hardware and simulated hardware. For an explanation of other settings panels, refer to the *IDE User's Guide*.

This chapter contains these topics:

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

Target Settings Overview

Changes that you make to target settings can significantly affect the size and performance of your application. Before you build a project, make sure to adjust target settings appropriately.

A project may contain more than one build target. The CodeWarrior IDE maintains separate target settings for each build target in a project. You must configure each build target separately. You can use the **Import Panel** button and the **Export Panel** button to transfer settings from one build target to another. For more information about importing and exporting panels, see the *IDE User's Guide*.

- [Opening the Target Settings Window](#)
- [Default Settings for Stationery](#)

Opening the Target Settings Window

To open the **Target Settings** window:

1. Bring forward the project window that has the build-target settings that you want to change.
2. Make sure that the build target for which you want to change settings is the current build target.

The current build target appears in the list box in the project-window toolbar.

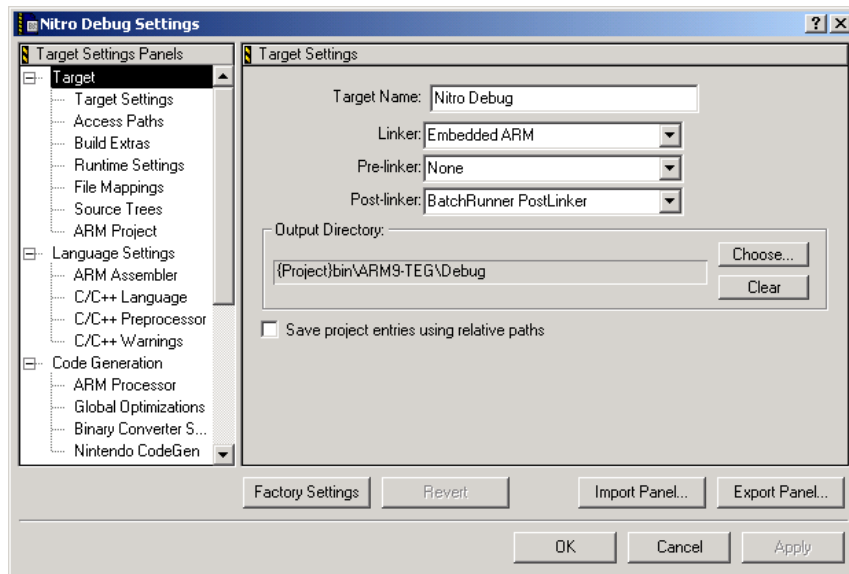
Target Settings

Target Settings Overview

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

The **Target Settings** window opens ([Figure 4.1](#)). The left hand pane of the **Target Settings** window lists several settings panels. The linker that you specify determines the other settings panels that are available.

Figure 4.1 Target Settings window



Default Settings for Stationery

If you created your CodeWarrior project by using project stationery, many of the target settings have default values appropriate for your project.

You should review the settings panels to ensure that the default configuration options are appropriate for your target hardware and development environment.

Target Settings Panels

This section describes Nintendo DS-specific target settings. The Nintendo DS-specific panels are:

- [ARM Project Panel](#)
- [ARM Assembler Panel](#)
- [ARM Processor Panel](#)
- [Binary Converter Settings Panel](#)
- [Nintendo CodeGen Panel](#)
- [BatchRunner PostLinker Panel](#)
- [ARM Linker Panel](#)
- [NITRO LCF Prelinker Panel](#)
- [Nitro MakeRom Postlinker Panel](#)
- [BatchRunner Preprocessor Panel](#)
- [ARM Debugger Settings Panel](#)
- [NITRO Debugger Setting Panel](#)

Settings panels of more general interest are discussed in other CodeWarrior manuals. [Table 4.1](#) lists these panels, and where you can find more information about them.

Table 4.1 Where to Find Information on Other Settings Panels

Panel	Manual
Target Settings	IDE Users Guide
Access Paths	IDE Users Guide
Build Extras	IDE Users Guide
Runtime Settings	IDE Users Guide
File Mappings	IDE Users Guide
Source Trees	IDE Users Guide
C/C++ Language	C Compilers Reference
C/C++ Preprocessor	C Compilers Reference
C/C++ Warnings	C Compilers Reference
Global Optimizations	IDE Users Guide

Table 4.1 Where to Find Information on Other Settings Panels

Panel	Manual
Custom Keywords	IDE Users Guide
Debugger Settings	IDE Users Guide
Remote Debugging	IDE Users Guide

ARM Project Panel

The ARM Project Panel ([Figure 4.2](#)) determines the type of project and the output-file name. [Table 4.2](#) describes the options in the panel.

Figure 4.2 ARM Project Panel

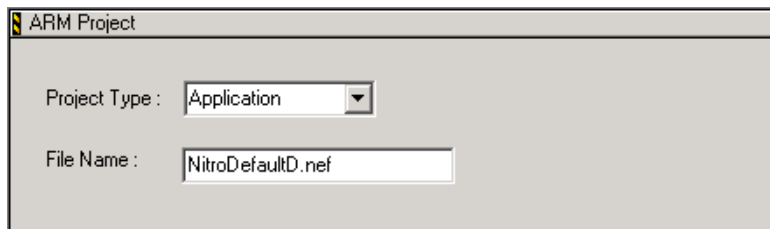


Table 4.2 ARM Project Panel Options

Option	Explanation
Project Type	Use to specify the type of output file for this build target: <ul style="list-style-type: none">• Application—an application executable file• Library—a shared library or dynamic library file• Partial linking—a partially linked object file that you can use in a later link operation Command line options: mwldarm.exe -application -library -partial
File Name	Enter the output filename. Command line options: mwldarm.exe -o <i>filename</i>

ARM Assembler Panel

The ARM Assembler panel ([Figure 4.3](#)) lets you specify how the assembler compiles ARM assembly language into object code in ARM Object Format. The linker can link this object code with these items:

- object code that the assembler produces
- object code that the compiler produces
- object libraries

Figure 4.3 ARM Assembler Panel

The screenshot shows the 'ARM Assembler' panel with the following settings:

- Source Format:**
 - ☐ Labels must end with ':'
 - ☒ Directives begin with '.'
 - ☐ Allow space in operand field
 - ☐ Case sensitive identifiers
- Architecture:**
 - ☒ Set from ARM Processor panel
 - Initial state: ARM
 - Floating point: Software
 - Architecture: ARM 4T
- ☐ GNU compatible syntax
- ☐ ARM ADS compatible syntax
- ☐ Generate listing file
- Prefix file: (empty text box)

Table 4.3 ARM Assembler Panel Options

Option	Explanation
Labels must end with ':'	<p>Check to have the assembler accept only labels that end with a colon.</p> <p>Clear to have the assembler accept labels regardless of an ending colon.</p> <p>Commandline: mwasmarm.exe -[no]colons</p>
Directives begin with '.'	<p>Check to have the assembler accept only directives that begin with a period.</p> <p>Clear to have the assembler accept directives regardless of a beginning period.</p> <p>Commandline: mwasmarm.exe -[no]period</p>
Allow space in operand field	<p>Check to have the assembler accept only operand fields that do not have spaces.</p> <p>Clear to have the assembler accept operand fields regardless of spaces.</p> <p>Commandline: mwasmarm.exe -[no]space</p>
Case sensitive identifiers	<p>Check to have the assembler consider the case of identifier characters. For example, the assembler recognizes <code>anIdentifier</code> and <code>anidentifier</code> as two distinct identifiers.</p> <p>Clear to have the assembler ignore the case of identifier characters. For example, the assembler treats <code>anIdentifier</code> and <code>anidentifier</code> as the same identifier.</p> <p>Commandline: mwasmarm.exe -[no]case</p>

Table 4.3 ARM Assembler Panel Options (*continued*)

Option	Explanation
Set from ARM Processor panel	<p>Check to have the IDE select configure the Initial state, Floating point, and Architecture settings according to the settings you specify in the ARM Processor panel.</p> <p>Clear to configure the Initial state, Floating point, and Architecture settings independently of the settings in the ARM Processor panel.</p>
Initial state	<p>Specifies how the assembler interprets the first line of assembly source (you may change how subsequent instructions are interpreted by using the <code>.arm</code> and <code>.thumb</code> directives to override the initial state).</p> <ul style="list-style-type: none"> ARM—assemble source instructions as ARM code Thumb—assemble source instructions as Thumb code <p>Select the expected processor state at execution time. This setting applies only to architectures and processors that support the Thumb instruction set.</p> <p>This option does not generate code that switches the processor state. You must ensure that the processor is in the state that you expect.</p> <p>Commandline: thumb: mwasmarm.exe -16 ARM: mwasmarm.exe -32</p>
Floating point	<p>Specifies the type of floating-point unit (FPU) for the target hardware architecture:</p> <p>The assembler might display error messages or warnings if the selected FPU architecture is not compatible with the target architecture.</p> <p>Commandline: mwasmarm.exe -fpu none soft</p>
Architecture	<p>Specifies the target hardware architecture.</p> <p>Commandline: mwasmarm.exe -proc arm4t arm5TE</p>

Table 4.3 ARM Assembler Panel Options (*continued*)

Option	Explanation
GNU compatible syntax	Instructs the assembler to accept GNU-style assembly syntax. See “GNU Assembler Compatibility” for details. Commandline: mwasmarm.exe -[no]gnu
ARM ADS compatible syntax	Instructs the assembler to accept ARM ADS-compatible syntax extensions. Commandline: mwasmarm.exe -[no]ads
Generate listing file	Instructs the assembler to generate a disassembly output file. The disassembly output file contains the file source, along with line numbers, relocation information, and macro expansion. Commandline: mwasmarm.exe -list
Prefix file	Specifies a prefix file that you want the assembler to include at the top of each assembly file. Commandline: mwasmarm.exe -include <i>filename</i>

ARM Processor Panel

The ARM Processor panel ([Figure 4.4](#)) lets you configure processor-related settings for your project.

Figure 4.4 ARM Processor Panel

ARM Processor

Target Architecture: Target Endian:

Floating-Point Model:

☐ Position-independent code ☐ Generate Thumb instructions

☐ Position-independent data ☒ Support ARM/Thumb interworking

☐ Make string constants read-only ☐ Support ARM Shared Library Architecture

☒ Permit dead-stripping of functions ☐ Generate code for profiling

Use .sdata section for objects up to bytes long

Table 4.4 ARM Processor Panel Options

Option	Explanation
Target Architecture	<p>Use to specify the target hardware architecture or processor name. The compiler can take advantage of the extra instructions that the selected architecture provides and optimize the code to run on a specific processor.</p> <p>The inline assembler might display error messages or warnings if it assembles some processor-specific instructions for the wrong target architecture.</p> <p>Command line options: mwccarm.exe -proc arm720t arm7tdmi arm946e v4t v5te</p>
Target Endian	<p>Use to specify the byte order of the target hardware architecture:</p> <ul style="list-style-type: none"> • Little—little endian; right-most bytes (those with a higher address) are most significant • Big—big endian; left-most bytes (those with a lower address) are most significant <p>Command line options: mwccarm.exe -little -big</p>

Table 4.4 ARM Processor Panel Options (*continued*)

Option	Explanation
Floating-Point Model	<p>Use to specify the type of floating-point unit (FPU) for the target hardware architecture. The Software model is a software-based FPU library.</p> <p>The assembler might display error messages or warnings if the selected FPU architecture is not compatible with the target architecture.</p> <p>Commandline: mwccarm.exe -fp soft</p>
Position-independent code	This option is not used for Nintendo DS development.
Position-independent data	This option is not used for Nintendo DS development.
Make string constants read-only	<p>Instructs the compiler to place string constants into the <code>.rodata</code> section.</p> <p>Commandline: mwccarm.exe -readonlystrings</p>
Permit dead-stripping of functions	<p>Check to allow dead-stripping optimizations at the function level.</p> <p>Clear to prevent dead-stripping optimizations at the function level.</p>
Generate Thumb instructions	<p>Check to have the processor generate Thumb code instructions.</p> <p>Clear to prevent the processor from generating Thumb code instructions.</p> <p>The IDE enables this setting only for architectures and processors that support the Thumb instruction set.</p> <p>Commandline: mwccarm.exe -[no]thumb</p>

Table 4.4 ARM Processor Panel Options (*continued*)

Option	Explanation
Support ARM/Thumb interworking	<p>Instructs the linker to generate suitable interworking veneers when it links the assembler output. You must enable this option if you write ARM code that you want to interwork with Thumb code or vice versa. The only functions that need to be compiled for interworking are the functions that are called from the other state. You must ensure that your code uses the correct interworking return instructions.</p> <p>If the Generate Thumb Instructions option is checked, this option is automatically enabled, and cannot be changed.</p> <p>Disable this option if you write ARM code that you do not want to interwork with Thumb code or vice versa.</p> <p>Commandline: mwccarm.exe -[no]interworking</p>
Support ARM Shared Library Architecture	<p>This option is not used for Nintendo DS development.</p> <p>Commandline: mwccarm.exe -[no]ashla</p>
Generate code for profiling	<p>Check to have the processor generate code for use with a profiling tool.</p> <p>Clear to prevent the processor from generating code for use with a profiling tool.</p> <p>Commandline: mwccarm.exe -[no]profile</p>
Use .sdata section for objects up to n bytes long	<p>Enter the maximum number of bytes (n) of an object length for which the processor uses a .sdata section.</p> <p>Commandline: mwccarm.exe -sdatathreshold <i>long</i></p>

Small Data and Third Party Libraries

The Small Data text box specifies the largest size, in bytes, of items that the linker stores in small data address space (.sdata and .sbss). For example, a value 16 in this box means that the linker will store all non-constant data items of 16 or fewer bytes in this space.

NOTE The `__declspec (sdata)` identifier forces a global data item into small data address space regardless of the data's actual size. This identifier's format is: `__declspec (sdata) longlong largeVariable;`

Programs need just one instruction to access a value in small data address space, making data access faster than it is for items stored in regular data address space (.data and .bss). For rapid program execution, you want as many data items as possible stored in small data address space.

However, the small data address space is fixed at 64 kilobytes, so an overflow error is possible. In case of such an error, you must reduce the value in the Small Data text box or modify the `__declspec (sdata)` identifier.

Another possible cause of small data overflow is adding precompiled, third-party libraries to your project. Such libraries already can contain small data; adding enough such libraries can exhaust the small data address space.

NOTE If you are creating precompiled libraries for other developers to use, your Small Data value should be 0. This will prevent your libraries from causing small-data overflows for the other developers.

To determine exactly your total small-data usage, you must add the small data that your code uses to the small data of all your precompiled libraries.

Calculating Your Small Data

Method One:

1. Create a link map file.
2. In the link map, look up all the .sdata and .sbss size values.
3. Add these values.

Method Two:

1. Create a link map file.
2. Put all your .sdata and .sbss items into a dedicated section of the linker command file.
3. In the link map, look up the size of the dedicated section.

Calculating Library Small Data

For each precompiled library:

1. Bring up the library name in the Project window.
2. Right-click on the library name. This brings up a context-sensitive menu.
3. Click the menu item Disassemble. The window displays the disassembled file.

4. Find the Section Header Table.
5. Read the .sdata and .sbss sizes from the table.

Another possibility: the library vendor may be able to tell you how much small data the library contains.

Binary Converter Settings Panel

The Binary Converter Settings panel ([Figure 4.5](#)) invokes the Binary Converter/Compiler, a plug-in that can convert any binary file into an ELF-format object file that you can link into your application.

See “[Binary Converter/Compiler](#)” for information relating to the Binary Converter/Compiler and its settings file.

Figure 4.5 Binary Converter Settings Panel

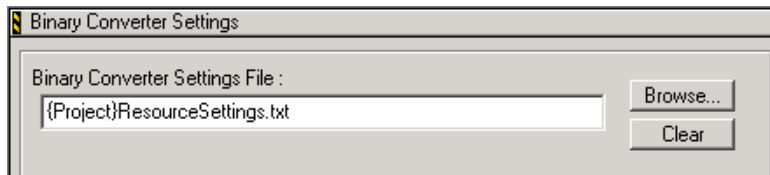


Table 4.5 Binary Convert Settings Panel Options

Option	Explanation
Binary Converter Settings File	Specifies an optional settings file for the Binary Converter.

Nintendo CodeGen Panel

The Nintendo CodeGen panel ([Figure 4.6](#)) lets you generate code that complies with the hardware restrictions on byte read/write access. These restrictions depend on the cache settings:

- Cache enabled
 `strb` is not allowed, `swpb`, `ldrb`, and `ldrswb` are allowed
- Cache disabled
 None of the 8-bit transfer instructions, `strb`, `swpb`, `ldrb`, or `ldrswb`, are allowed.

Figure 4.6 Nintendo CodeGen Panel

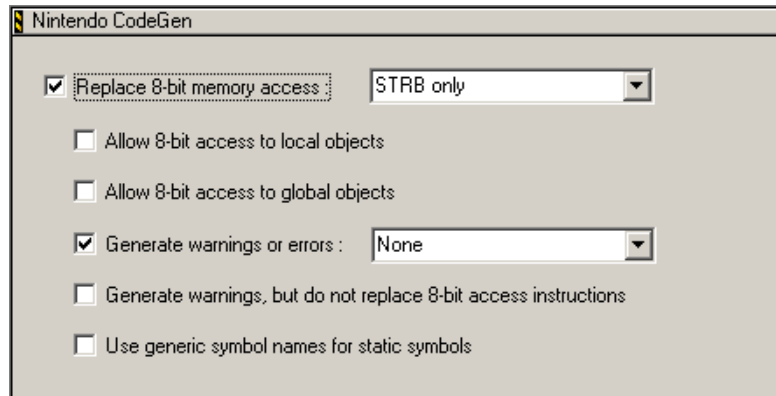


Table 4.6 Nintendo CodeGen Panel Options

Option	Explanation
Replace 8-bit Memory Access	<p>Replaces 8-bit memory access with 16-bit memory access for the instruction types you specify in the list box:</p> <ul style="list-style-type: none"> • No replacement No memory restrictions. All instructions are allowed. Compiler generates 8-bit accesses. • STRB only Use this option when the main memory cache is enabled. The compiler replaces <code>strb</code> instructions with <code>swpb</code>, <code>ldrb</code>, or <code>ldrsh</code> equivalents. • All instructions Use this option when the main memory cache is disabled. The compiler replaces <code>strb</code>, <code>ldrb</code>, and <code>ldrsh</code>, and does not generate <code>swpb</code>. <p>Checking this checkbox and setting it to a value other than No Replacement enables the other options in this panel.</p> <p>Declaration specifiers:</p> <ul style="list-style-type: none"> • <code>__declspec(byte_access)</code> Objects with this type modifier may be safely accessed with 8-bit access instructions • <code>__declspec(no_byte_access)</code> Objects with this type modifier may not be accessed with 8-bit access instructions <p>Pragma: <code>#pragma avoid_byte none strb all</code></p> <p>Command line options: <code>mwccarm.exe -avoid_byte none strb all</code></p>
Allow 8-bit access to local objects	<p>Use this option when the stack is placed in the internal memory. The compiler does not replace <code>strb</code> and <code>ldrb</code> if the compiler determines that the memory being accessed is on the stack.</p> <p>Pragma: <code>#pragma allow_byte local</code></p> <p>Command line options: <code>mwccarm.exe -allow_byte none local global both</code></p>

Table 4.6 Nintendo CodeGen Panel Options (*continued*)

Option	Explanation
Allow 8-bit access to global objects	<p>Use this option when global data is placed in the internal memory.</p> <p>Pragma: #pragma allow_byte global</p> <p>Command line options: mwccarm.exe -allow_byte none local global both</p>
Generate warnings or errors	<p>Replacing 8-bit memory-accesses is expensive in terms of code size, speed, and the number of registers required. The compiler can notify you when such replacements occur, based upon the list box setting:</p> <ul style="list-style-type: none"> • None Do not generate any warning or error messages on replacement. • Warning-All Generates warning messages for each replacement • Error-All Generates an error on 8-bit memory accesses. <p>Pragma: #pragma -warn_byte none all error</p> <p>Command line options: mwccarm.exe -warn_byte none all error</p>

Table 4.6 Nintendo CodeGen Panel Options (*continued*)

Option	Explanation
Generate warnings, but do not replace 8-bit access instructions	<p>Compiles your code without replacing any instructions, but generates warning messages as if you had compiled for a cache-restricted configuration. This is helpful to check which routines can be moved to a restricted memory configuration without performance penalties.</p> <p>This option has no command line equivalent</p>
Use generic symbol names for static symbols	<p>Obfuscates the names of static symbols within binaries (such as libraries) as protective measure against unauthorized persons disassembling the binary. Such a disassembly can reveal the names of static symbols and may expose internal structures and other proprietary details.</p> <p>The compiler replaces static symbol names with generic ones, for example “my_local” becomes “@1234”. This feature only works when debugging is disabled.</p> <p>Pragma: #pragma generic_symbol_names on off</p> <p>Command line options: mwccarm.exe -generic_symbol_names on off</p>

BatchRunner PostLinker Panel

The BatchRunner Postlinker ([Figure 4.7](#)) lets you run a batch file or other program after the IDE has compiled and linked all your project files.

Figure 4.7 BatchRunner PostLinker Panel

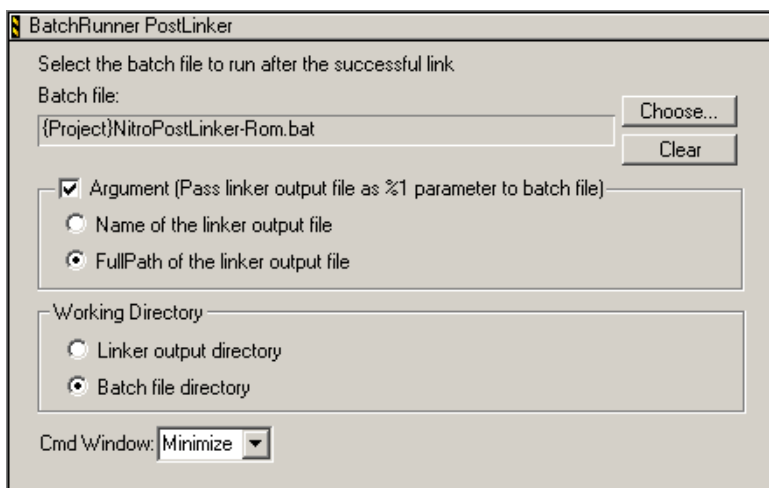


Table 4.7 BatchRunner PostLinker Panel Options

Option	Explanation
Select the Batch File...	Use to specify the batch file or program that will be run after the linker successfully creates an output file.
Argument (Pass linker output file as %1 parameter to batch file)	Passes an argument to the batch file as a %1 parameter. <ul style="list-style-type: none"> Name of the Linker Output File Passes only the name of the linker output file Full path of the Linker Output File Passes the name and path of the linker output file

Table 4.7 BatchRunner PostLinker Panel Options (*continued*)

Option	Explanation
Working Directory	<p>Specifies the working directory — the directory in which the batchfile is run.</p> <ul style="list-style-type: none">• Linker Output Directory Sets the working directory to the linker output directory• Batch File Directory Sets the working directory to the directory the batch file resides in
Cmd Window	<p>Determines the behavior of the command window that is opened while a batch file executes.</p> <ul style="list-style-type: none">• Minimize• Show• Hide

BatchRunner Preprocessor Panel

The BatchRunner Preprocessor panel ([Figure 4.8](#)) lets you run a batch file during the preprocessor stage of the build process, on source files that match a specific filename extension.

The batch file is only invoked if a matching source file has been updated since the last build — the BatchRunner Preprocessor verifies the dependency status of the source files that are defined in the File Mappings panel. The batch file is invoked with two arguments: %1, the full path of the source file; %2, the full path of the required preprocessed output file.

If the batch file returns a 0 value, the IDE adds the preprocessed output file to the project and compiles the output file. If the batch file returns a non-zero value, the IDE display an error message and halts the build process.

Figure 4.8 BatchRunner Preprocessor Panel

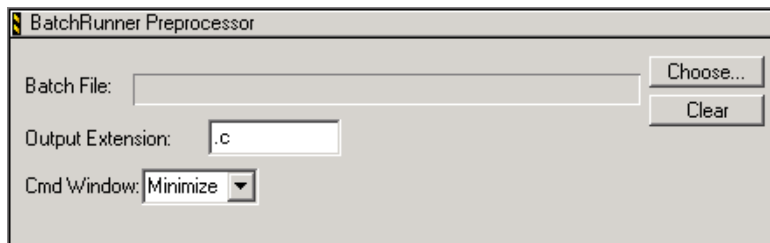


Table 4.8 BatchRunner Preprocessor Panel Options

Option	Explanation
Batch File	Specifies the batch file that will be invoked.
Output Extension	Specifies the filename extension for pattern matching.
Cmd Window	Determines the behavior of the command window that is opened while a batch file executes. <ul style="list-style-type: none">• Minimize• Show• Hide

ELF Disassembler Panel

The ELF Disassembler panel ([Figure 4.9](#)) lets you configure the information that the disassembler shows in disassembled object code.

Figure 4.9 ELF Disassembler panel

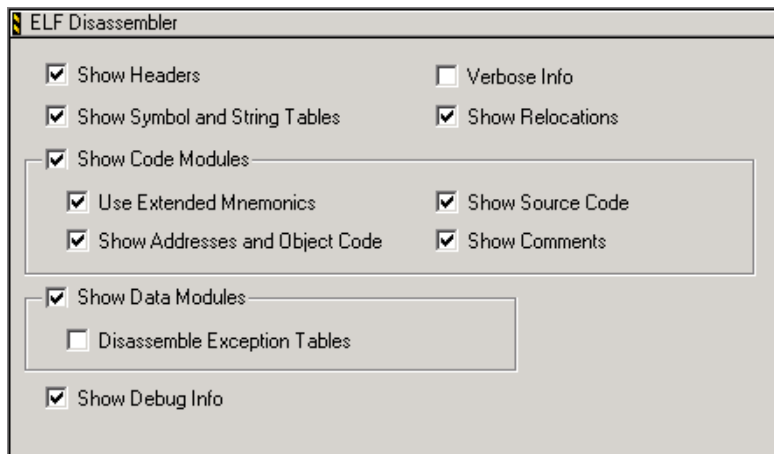


Table 4.9 ELF Disassembler Panel Options

Option	Explanation
Show Headers	<p>Check to have the disassembler show disassembly headers.</p> <p>Clear to prevent the disassembler from showing disassembly headers.</p> <p>Command line options: mwldarm.exe -show [no]headers</p>
Show Symbol and String Tables	<p>Check to have the disassembler show symbol and string tables.</p> <p>Clear to prevent the disassembler from showing symbol and string tables.</p> <p>Command line options: mwldarm.exe -show [no]names</p>
Verbose Info	<p>Check to have the disassembler show detailed disassembly information.</p> <p>Clear to have the disassembler show less detailed disassembly information.</p> <p>Command line options: mwldarm.exe -show [no]verbose</p>
Show relocations	<p>Check to have the disassembler show information about relocated symbols.</p> <p>Clear to prevent the disassembler from showing information about relocated symbols.</p> <p>Command line options: mwldarm.exe -show [no]relocs</p>

Table 4.9 ELF Disassembler Panel Options (*continued*)

Option	Explanation
Show Code Modules	<p>Check to have the disassembler show information about code modules. Checking this checkbox enables the Use Extended Mnemonics, Show Addresses and Object Code, Show Source Code, and Show Comments options.</p> <p>Clear to prevent the disassembler from showing information about code modules. Clearing this checkbox disables the Use Extended Mnemonics, Show Addresses and Object Code, Show Source Code, and Show Comments options.</p>
Use Extended Mnemonics	<p>Check to have the disassembler use extended mnemonic information to show code modules.</p> <p>Clear to prevent the disassembler from using extended mnemonic information to show code modules.</p> <p>Command line options: mwldarm.exe -show [no]extended</p>
Show Addresses and Object Code	<p>Check to have the disassembler show information about addresses and object code.</p> <p>Clear to prevent the disassembler from showing information about addresses and object code.</p> <p>Command line option: mwldarm.exe -show [no]code</p>
Show Source Code	<p>Check to have the disassembler show source code.</p> <p>Clear to prevent the disassembler from showing source code.</p> <p>Command line options: mwldarm.exe -show [no]source</p>
Show Comments	<p>Check to have the disassembler show comments.</p> <p>Clear to prevent the disassembler from showing comments</p> <p>Command line options: mwldarm.exe -show [no]comments</p>

Table 4.9 ELF Disassembler Panel Options (*continued*)

Option	Explanation
Show Data Modules	<p>Check to have the disassembler show information about data modules. Checking this checkbox enables the Disassemble Exception Tables option.</p> <p>Clear to prevent the disassembler from showing information about data modules. Clearing this checkbox disables the Disassemble Exception Tables option.</p> <p>Command line options: mwldarm.exe -show [no]data</p>
Disassemble Exception Tables	<p>Check to have the disassembler disassemble exception tables.</p> <p>Clear to prevent the disassembler from disassembling exception tables.</p> <p>Command line options: mwldarm.exe -show [no]exceptions</p>
Show debug info	<p>Check to have the disassembler show debugging-table information.</p> <p>Clear to prevent the disassembler from showing debugging-table information.</p> <p>Command line options: mwldarm.exe -show [no]debug</p>

ARM Linker Panel

The ARM Linker panel ([Figure 4.10](#)) lets you configure linker-related settings. To process your output with more than one linker or post-linker, you can use dependent subprojects or subtargets to select additional linkers or post-linkers. For more information about creating subprojects and subtargets, see *The IDE User's Guide*.

Figure 4.10 ARM Linker Panel

The screenshot shows the 'ARM Linker' panel with the following settings:

- ☒ Generate Symbolic Info
 - ☒ Store Full Path Names
- ☒ Generate Link Map
 - ☐ List Unused Objects
 - ☐ Show Transitive Closure
- ☐ Generate S-Record File
 - Max. Record Length: 252
 - EOL Character: MAC
- ☐ Disable Dead-stripping
- ☐ Suppress Warning Messages
- ☒ Generate Symbol Table
 - ☐ Sort Symbols by Value
 - ☐ Mapping Symbols First
- ☐ Generate X-Record File
 - Max. Record Length: 252

Entry Point:

Force Active Symbols

Table 4.10 ARM Linker Panel Options

Option	Explanation
Generate Symbolic Info	<p>Instructs the linker to generate and include debugging information (DWARF symbolics) in the output file.</p> <p>Enabling this option also enables the Store Full Path Names checkbox.</p> <p>Command line options: mwldarm.exe -g -sym on</p>
Store Full Path Names	<p>Instructs the linker to include the full path names of source files in the generated symbolic information.</p> <p>Command line options: mwldarm.exe -sym full[path]</p>
Generate Link Map	<p>Instructs the linker to generate a link map file, a text file that shows which files provide the definition for each object and function in the 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.</p> <p>Enabling this option also enables the List Unused Objects and Show Transitive Closure checkboxes.</p> <p>Command line options: mwldarm.exe -map</p>
List Unused Objects	<p>Instructs the linker to include unused objects in the link map file.</p> <p>Command line options: mwldarm.exe -map unused</p>

Table 4.10 ARM Linker Panel Options (*continued*)

Option	Explanation
Show Transitive Closure	<p>Instructs the linker to add a recursive list of the objects referenced by main() to the link map file. Here is some sample code and the resulting recursive list in the link map file.</p> <p>Command line options: mwldarm.exe -map closure</p> <pre>void foo(){ int a = 1001; } void foo1(){ int b = 1002; } void NitroMain(void){ foo(); foo1(); }</pre> <p># Link map of _start 1] _start (func, overload) found in crt0.o 2] NitroMain (func, global) found in main.c 3] foo (func, global) found in main.c 3] foo1 (func, global) found in main.c</p>
Generate S-Record File	<p>Instructs the linker to generate an S-record file. The S-record format, developed by Motorola, Inc., is a file format for transmitting binary files to target systems.</p> <p>Enabling this option also enables the Max. Record Length and EOL Character checkboxes.</p> <p>Command line options: mwldarm.exe -srec</p>
Max. Record Length	<p>Specifies the maximum length in bytes for the generated S-record file. This text box becomes available after you check the Generate S-Record File checkbox.</p> <p>Command line options: mwldarm.exe -sreclength <i>length</i></p>

Table 4.10 ARM Linker Panel Options (*continued*)

Option	Explanation
EOL Character	<p>Specifies the host-platform End Of Line (EOL) character convention:</p> <ul style="list-style-type: none"> • MAC—Apple Macintosh platform • DOS—Microsoft MS-DOS (default) • UNIX—UNIX operating system <p>Command line options: mwldarm.exe -srecol mac dos unix</p>
Disable Dead-stripping	<p>Prevents the linker from removing unused code and data.</p> <p>Command line options: mwldarm.exe -[no]deadstrip</p>
Suppress Warning Messages	<p>Prevents the IDE from displaying linker warning messages.</p> <p>Command line options: mwldarm.exe -warnings off on</p>
Generate Symbol Table	<p>Instructs the linker to generate an ELF symbol table, as well as a list of relocations, in the ELF executable file.</p> <p>Enabling this option also enables the Sort Symbols by Value and Mapping Symbols First checkboxes.</p> <p>Command line options: mwldarm.exe -symtab</p>
Sort Symbols by Value	<p>Sorts the symbols in the generated symbol table according to their values.</p> <p>Command line options: mwldarm.exe -symtab sort</p>
Mapping Symbols First	<p>Check to list mapping symbols first in the symbol table.</p> <p>Clear to prevent listing mapping symbols first in the symbol table.</p> <p>Command line options: mwldarm.exe -symtab mapsymfirst</p>

Table 4.10 ARM Linker Panel Options (*continued*)

Option	Explanation
Generate X-Record File	<p>Instructs the linker to generate a Hex-record (X-record) file. The Hex-record file, developed by Intel Corp., is a hexadecimal object-file format that some cross assemblers and utilities support.</p> <p>Enabling this option also enables the Max. Record Length checkbox.</p>
Max. Record Length	<p>Specifies the maximum length in bytes for the generated X-record file. This text box becomes available after you check the Generate X-Record File checkbox.</p>
Entry Point	<p>Specifies the entry-point symbol for the linker. The default symbol is <code>__startup</code>.</p> <p>Command line options: <code>mwldarm.exe -main <i>symbol</i></code></p>
Force Active Symbols	<p>Instructs the linker to include certain symbols in the link, even if those symbols are not referenced. It is a way to make symbols immune to deadstripping. For example, enter the symbol of an unreferenced copyright string in order to keep it in the final executable file. This option is equivalent to the compiler pragma and linker directive, <code>force_active</code>.</p> <p>To list multiple symbols, delimit multiple symbols using a single space between them as a separator.</p> <p>If using C++ code, you must specify the mangled name of the symbol instead. To find the mangled name, disassemble the source file and search for the symbol's unmangled name. The mangled version of the symbol name will be in close proximity.</p> <p>Command line options: <code>mwldarm.exe -force_active <i>symbol1</i>[,<i>symbol2</i>...]</code></p>

NITRO LCF Prelinker Panel

The NITRO LCF Prelinker panel ([Figure 4.11](#)) controls the behavior of the Linker Command File generator. This panel is only available if you select **NITRO LCF Generator** as a prelinker in the Target Settings panel.

Figure 4.11 NITRO LCF Prelinker Panel

The screenshot shows the 'NITRO LCF Prelinker' dialog box. It contains several sections for configuring linker settings:

- Linker Command File Generation:** Three radio buttons are present: 'Always create a new LCF' (selected), 'Always use a modified LCF', and 'Generate LCF file and halt build'.
- TCM Address:** Two text boxes for 'ITCM' (0x01ff8000) and 'DTCM' (0x027c0000).
- Section Name:** A text box containing 'main'.
- Address:** A text box containing '0x02004000'.
- StackSize:** A text box containing '0'.
- IRQ StackSize:** A text box containing '1024'.
- OverlayDefs:** A text box containing '%_defs'.
- OverlayTable:** A text box containing '%_table'.
- Suffix:** A text box containing '.sbin'.
- LCF Template File:** A text box containing '{NITROSDK_ROOT}\include\nitro\specfiles\ARM9-TS.lcf.template' and a 'Choose...' button.
- Run batch file after LCF generation:** A checkbox that is currently unchecked.
- Batch File:** A text box for a batch file path and a 'Choose...' button.
- Cmd Window:** A dropdown menu set to 'Minimize' and a 'Clear' button.

Table 4.11 NITRO LCF Prelinker Panel Options

Option	Explanation
Linker Command File Generation	<p>These settings determine when and if a linker command file may be generated. Creating a new LCF destroys any previous LCF. If your old LCF has a custom setting you wish to preserve, select the Always Use a Modified LCF radio button.</p> <ul style="list-style-type: none"> • Always Create a New LCF Forces the LCF generator to create a new LCF each time you Make your project. This is the default setting. • Always Use a Modified LCF Disables the LCF generator • Generate LCF File and Halt Build Forces the LCF generator to create a new LCF when you Make your project, but it halts the Make process after the LCF has been created. This lets you customize the LCF before you invoke the linker.
ITCM	Specifies the starting address of the ITCM memory segment defined in the LCF's MEMORY section. The default value is 0x01ff8000.
DTCM	Specifies the starting address of the DTCM memory segment defined in the LCF's MEMORY section. The default value is 0x027c0000.
Address	Specifies the starting address of the main memory segment, within the LCF's MEMORY section. The correct value for the TS hardware and the software emulator target is 0x02000000. The correct value for the TEG hardware is 0x02004000.
Section Name	Specifies a section name for the main segment.
StackSize	Specifies the stack size defined by the LCF symbol SDK_SYS_STACKSIZE. The default value is 0 (max size).
IRQ Stack Size	Specifies the IRQ stack size defined by the LCF symbol SDK_IRQ_STACKSIZE. The default value is 1024.
OverlayDefs	Specifies the filename of the overlay definitions file. The Suffix will be appended to the filename. You can use % to use the section name in the filename. The default value is %_defs

Table 4.11 NITRO LCF Prelinker Panel Options

Option	Explanation
OverlayTable	Specifies the filename of the overlay table file. The Suffix will be appended to the filename. You can use % to use the section name in the filename. The default value is %_table
Suffix	Specifies the suffix of the binary output files that the linker generates for each memory segment. The default value is .sbin
LCF Template File	Specifies the LCF template file that the prelinker will use with the SDK makelcf tool. Click the Browse button to select a different LCF template file.
Run Batch File After LCF Generation	<p>Runs the specified batch file after generating the LCF. To specify the batch file, click the Choose button.</p> <p>The working directory for batch files run from this panel is the same directory as the executed batch file.</p>
Cmd Window	<p>Determines the behavior of the command window that is opened while a batch file executes.</p> <ul style="list-style-type: none">• Minimize• Show• Hide

Nitro MakeRom Postlinker Panel

The Nitro MakeRom Postlinker Panel ([Figure 4.12](#)) lets you specify the options for the SDK compstatic and makerom tools to create a ROM image of your project. Please consult your SDK documentation for descriptions of the compstatic and makerom tool options.

Special Notes:

- [Working Directory](#)
- [Using FS Functions](#)

Working Directory

The working directory for the makerom tool is set to the **Output Directory** as specified in the **Target > Target Settings** panel.

Using FS Functions

To use SDK FS functions from a ROM image, you must:

1. define the files in the RomSpec section of the .rsf file
2. specify the .rsf file in the **makeroom tool options**

You can do this with any CodeWarrior project, including those made from stationery. For an example, refer to the CodeWarrior project at

{CW}\Example\NITRO\SDK2.1\ARM9-TS\RomFileSample

Figure 4.12 Nitro MakeRom Postlinker Panel

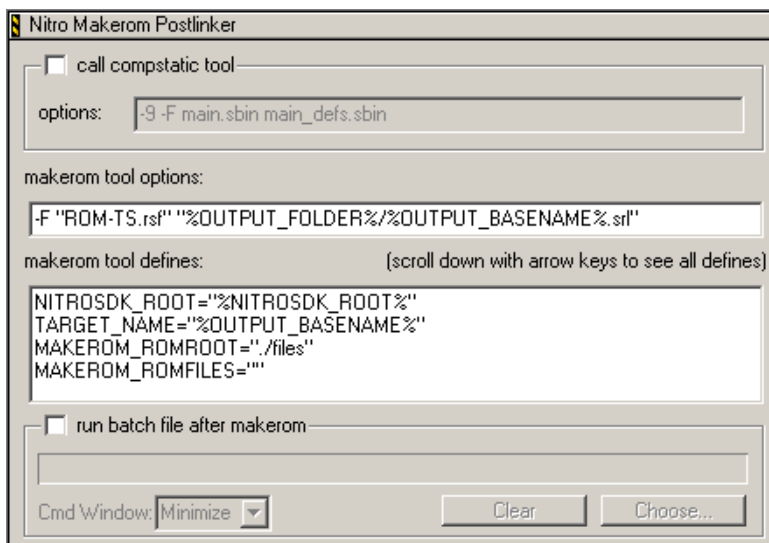


Table 4.12 NITRO MakeRom Postlinker Panel

Option	Explanation
call compstatic tool	Calls the SDK compstatic tool that compresses the static region of the ROM image. This reduces the memory footprint.
options	Lets you specify the options for the SDK compstatic tool. This text field is only active when the call compstatic tool option is checked.

Table 4.12 NITRO MakeRom Postlinker Panel

Option	Explanation
makerom tool options	<p>Let you specify the options for the SDK makerom tool. You may use the following predefined keywords</p> <ul style="list-style-type: none"> • %NITROSDK_ROOT% path of your NitroSDK source tree • %OUTPUT_FILENAME% main.nef • %OUTPUT_FULLPATH% c:\foo\bar\main.nef • %OUTPUT_FOLDER% c:\foo\bar • %OUTPUT_BASENAME% main
makerom tool defines	<p>Lets you set the values of the defines used in the .rsf file. Options you specify in this edit box are added as -D options for the makerom tool.</p> <p>You may use the same predefined keywords as described above for the makerom tool options.</p>
run batch file after makerom	<p>Runs the specified batch file after running the SDK makerom tool. To specify the batch file, click the Choose button.</p> <p>The working directory for batch files run from this panel is the Output Directory specified in the Target > Target Settings panel.</p>
Cmd Window	<p>Determines the behavior of the command window that is opened while a batch file executes.</p> <ul style="list-style-type: none"> • Minimize • Show • Hide

ARM Debugger Settings Panel

The ARM Debugger Settings panel ([Figure 4.13](#)) lets you configure debugger-related settings. You may specify:

- the target processor
- an initialization file for the target processor
- a memory-configuration file for the target processor

- additional debugger settings

NOTE If your project targets a ROM area, you cannot download data sections to read-only memory, so clear all checkboxes in the **Program Download Options** group. If your project targets a RAM area, check the checkboxes for the items that you want to download.

Figure 4.13 ARM Debugger Settings Panel

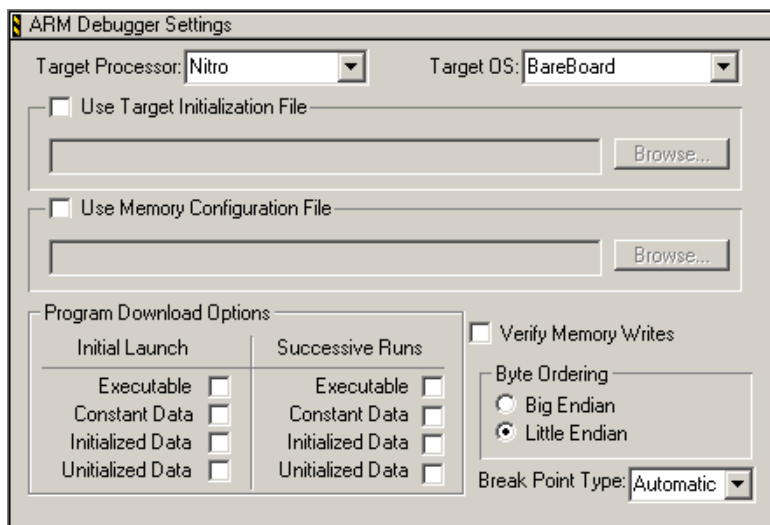


Table 4.13 ARM Debugger Settings Panel Options

Option	Explanation
Target Processor	Use to specify the target processor architecture. The debugger configures the appropriate register views according to the target processor that you specify. For NINTENDO DS development, you must select Nitro .
Target OS	Use to specify the operating system on which you debug the application.

Target Settings

Target Settings Panels

Table 4.13 ARM Debugger Settings Panel Options (*continued*)

Option	Explanation
Use Target Initialization File	<p>Check to have the debugger execute an initialization file before downloading the image file to the Target Processor. Checking this checkbox enables the text box and Browse button below it. Enter in the text box the absolute path to the file, or click the Browse button to select the file.</p> <p>Clear to prevent the debugger from executing an initialization file before downloading the image file to the Target Processor. Clearing this checkbox disables the text box and Browse button below it.</p> <p>You can find standard initialization files in</p> <pre>CodeWarrior\ARM_EABI_Support\ Initialization_Files\</pre> <p>where <i>CodeWarrior</i> is the path to your CodeWarrior installation.</p>
Use Memory Configuration File	<p>Check to have the debugger use a memory configuration file in order to access memory on the Target Processor. Checking this checkbox enables the text box and Browse button below it. Enter in the text box the absolute path to the file, or click the Browse button to select the file.</p> <p>Clear to have the debugger treat all memory locations as accessible. Clearing this checkbox disables the text box and Browse button below it.</p> <p>The memory configuration file has information about the location of read-only and read/write memory regions in the Target Processor.</p> <p>You can find standard memory-configuration files in</p> <pre>CodeWarrior\ARM_EABI_Support\ Initialization_Files\</pre> <p>where <i>CodeWarrior</i> is the path to your CodeWarrior installation. Memory-configuration files have the extension <code>.mem</code></p>

Table 4.13 ARM Debugger Settings Panel Options (*continued*)

Option	Explanation
Executable	<ul style="list-style-type: none"> • Initial Launch—Check to download executable code and text sections of the program to the Target Processor the first time you debug a project after opening the IDE. Clear to turn off this feature. • Successive Runs—Check to download executable code and text sections of the program to the Target Processor each time you debug a project after initial launch. Clear to turn off this feature.
Constant Data	<ul style="list-style-type: none"> • Initial Launch—Check to download constant data sections of the program to the Target Processor the first time you debug a project after opening the IDE. Clear to turn off this feature. • Successive Runs—Check to download constant data sections of the program to the Target Processor each time you debug a project after initial launch. Clear to turn off this feature.
Initialized Data	<ul style="list-style-type: none"> • Initial Launch—Check to download initialized data sections of the program to the Target Processor the first time you debug a project after opening the IDE. Clear to turn off this feature. • Successive Runs—Check to download initialized data sections of the program to the Target Processor each time you debug a project after initial launch. Clear to turn off this feature.
Uninitialized Data	<ul style="list-style-type: none"> • Initial Launch—Check to download uninitialized data sections of the program to the Target Processor the first time you debug a project after opening the IDE. Clear to turn off this feature. • Successive Runs—Check to download uninitialized data sections of the program to the Target Processor each time you debug a project after initial launch. Clear to turn off this feature.

Table 4.13 ARM Debugger Settings Panel Options (*continued*)

Option	Explanation
Verify Memory Writes	<p>Check to have the debugger verify memory-write operations during the download process.</p> <p>Clear to have the debugger perform the download process without verifying memory-write operations.</p> <p>The debugger reads the section of memory that it writes to the target board. This way, the debugger verifies that it correctly wrote all sections to the board. This process allows you to verify that the write operation was successful, and that neither runaway code nor the program stack modified the written sections.</p>
Big Endian	Select if the Target Processor uses big-endian (BE) byte order (left-most bytes are most significant: B3 B2 B1 B0).
Little Endian	Select if the Target Processor uses little-endian (LE) byte order (right-most bytes are most significant: B0 B1 B2 B3).
Break Point Type	<p>Use to specify the breakpoint mode that the debugger sets during debugging sessions:</p> <ul style="list-style-type: none"> • Automatic—the debugger determines whether to set breakpoints in ARM mode or Thumb mode • ARM—the debugger always sets breakpoints in ARM mode • Thumb—the debugger always sets breakpoints in Thumb mode <p>If your project generates symbolics information, specify Automatic to have the debugger automatically set the mode. Otherwise, if you specify ARM or Thumb mode, the debugger always sets that mode, even if the code is for the other mode.</p>

NITRO Debugger Setting Panel

The NITRO Debugger Setting panel lets you open a second thread window for the ARM7 processor when you start a debug session. It lets you specify where to stop the ARM7 application for multi-core debugging ([Figure 4.14](#)).

Figure 4.14 Nitro Debugger Setting panel

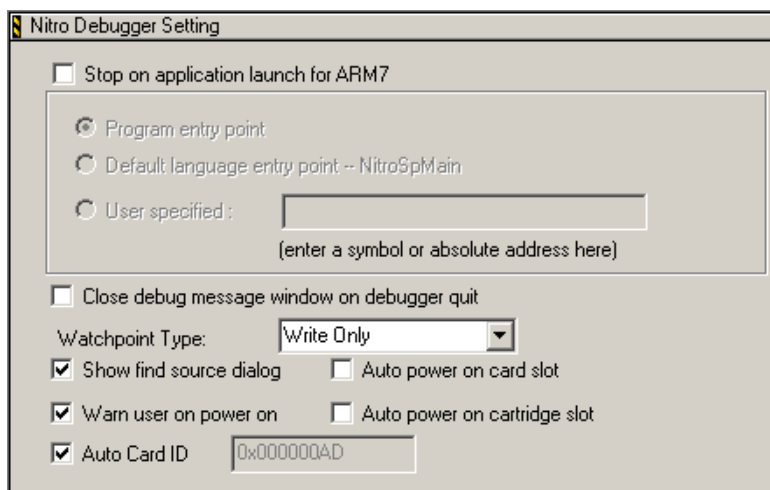


Table 4.14 Nitro Debugger Setting Panel Options

Option	Explanation
Stop on application launch for ARM7	Opens a thread window for the ARM 7 processor at the start of the debug session, and lets you specify the stop point
Program Entry Point	Specifies the stop point as the program entry point, the exact point from which the ELF program starts executing.
Default Language Entry Point - NitroSpMain	Specifies the stop point as the NitroSpMain function
User Specified	Specifies the stop point as a user-defined symbol or memory address
Close debug message window on debugger quit	Closes the debug message window after you quit the debugger. Otherwise, the message window remains open even after the debugger window is closed.

Target Settings

Target Settings Panels

Table 4.14 Nitro Debugger Setting Panel Options (*continued*)

Option	Explanation
Watchpoint Type	Lets you set the watchpoint type for the debug session. Note: This option only affects the hardware debug target. The Ensata emulator watchpoint must be set from the Nitro Emulator Connection definition in the Edit > Preferences > Remote Connections panel.
Show find source dialog	If the code you are debugging does not have any source code associated with it in the project, the debugger can prompt you to specify the source file it should use for source-level debugging.
Warn user on power on	Displays a reminder that a card or cartridge should never be inserted or removed while the power is on.
Auto power on card slot	Automatically power on the card slot when you start a debug session. The default value is off.
Auto power on cartridge slot	Automatically power on the cartridge slot when you start a debug session. The default value is off.
Auto Card ID	Lets the debugger detect whether your binary is for One Time PROM or for Mask ROM, and sets the Card ID accordingly. The automatically set CARD ID value may be different from the ID of a mass production ROM. If unchecked, you must enter the Card ID into the text field.

Debugging Procedures and Utilities

This chapter is a supplement to the Debugger chapter of the *IDE User's Guide*. The information here only describes the aspects of the CodeWarrior debugger that are specific to Nintendo DS development.

NOTE Many of the debugger utilities described in this chapter require an active debug session. To activate a CodeWarrior debugger session, open your project and select **Project > Debug**.

This chapter has these topics:

- [Configuring a Project for Debugging](#)
- [ARM/Thumb Mode and Disassembly](#)
- [Load/Save/Fill Memory Operations](#)
- [Debugging Stand-alone ELF \(.nef\) Files](#)
- [Multi-Core Debugging Options](#)
- [Cache Viewer](#)
- [System Browser \(Processor and Thread Browser\)](#)
- [Using the Profiler](#)
- [Using Watchpoints](#)
- [Activating DS Flash Card and GBA Cartridge Slots](#)
- [Writing Emulator Memory to DS Flash Card](#)
- [Writing User Contents to Backup Device](#)
- [View ROM Header](#)
- [One Time PROM](#)

Configuring a Project for Debugging

This section explains how to configure your project in order to debug a program on a simulator or on development hardware. These target settings panels control the way the debugger works:

- [Debugger Settings Panel](#)
- [Remote Debugging Panel](#)

Debugger Settings Panel

This panel configures application stopping and data-update intervals. [Figure 5.1](#) shows the panel. [Table 5.1](#) explains the panel options that apply to Nintendo DS development.

Figure 5.1 Debugger Settings panel

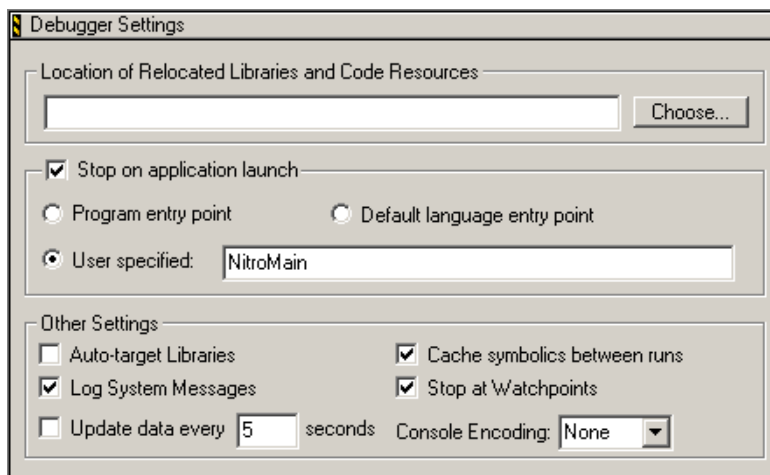


Table 5.1 Debugger Settings panel—options

Option	Explanation
Stop on application launch	<p>Check to set a temporary breakpoint that halts program execution. Checking this checkbox enables the option buttons below it. Select the option button for the point at which you want to halt the program:</p> <ul style="list-style-type: none"> • Program entry point—the point of launch of the program, before any statements execute • Default language entry point—the default entry point for the programming language. For example, the default entry point for the C++ language is the <code>main()</code> function. • User specified—the function or symbol that you specify by name. Select this option button to enable its associated text box. Enter in the text box the name of the function or symbol at which you want to halt program execution. The text box has the sample entry <code>NitroMain</code>. <p>Clear to have the debugger continue executing program statements until it arrives at a breakpoint.</p>
Cache symbolics between runs	<p>Check to have the debugger cache symbolics information after killing a process. Caching this information improves performance on successive runs and preserves the information from one debugging session to the next.</p> <p>Clear to prevent the debugger from caching symbolics information. The debugger discards all symbolics information after each debugging session ends.</p>
Update data every <i>n</i> seconds	<p>Enter the number of seconds <i>n</i> that the debugger waits before updating the information in debugging-session windows.</p>

Remote Debugging Panel

The Remote Debugging panel ([Figure 5.2](#)) lets you specify the remote connection that the debugger should use to connect to the target. The CodeWarrior IDE ships with two pre-configured remote debugger settings:

- IS-NITRO-EMULATOR — IS hardware emulator
- Nitro Emulator Connection — Ensata software emulator

Figure 5.2 Remote Debugging Panel

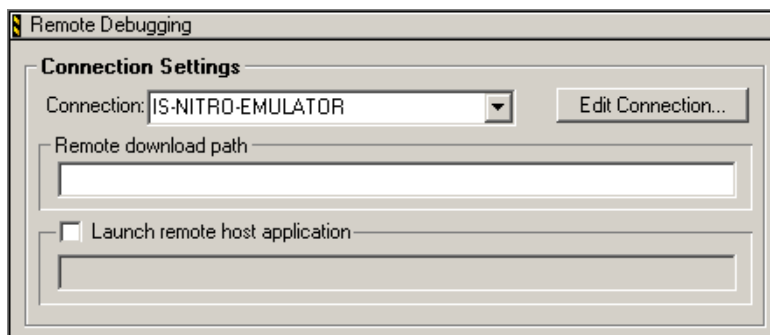


Table 5.2 Remote Debugging panel—options

Option	Explanation
Connection	<p>Use to specify the remote connection that the debugger uses for the current build target.</p> <p>You use the Remote Connections preference panel in the IDE Preferences window to define the remote connections that appear in this list box.</p>

Table 5.2 Remote Debugging panel—options (*continued*)

Option	Explanation
Remote download path	Enter the absolute path to the directory in which to store downloaded files. This option does not apply to bareboard development.
Launch remote host application	Check to have the IDE launch an application that serves as a host application on the remote computer. Checking this checkbox enables the text box below it. Enter the absolute path to the remote application: Clear to prevent the IDE from launching a host application on the remote computer. This option does not apply to bareboard development.

ARM/Thumb Mode and Disassembly

When the debugger disassembles a block of memory, it tries to accurately display ARM and thumb instructions by determining the state:

- if symbolics exist, the debugger gets the state from the symbolics file matching the address,
- if symbolics do not exist, the debugger gets the current state from the hardware CPSR Register T bit.

In other words, if there are no debugger symbolics for the memory block, the debugger assumes that the memory block is in the same state as the code at the program counter (PC). If this assumption is incorrect, the debugger can be fooled into displaying thumb instructions as ARM instructions, or vice versa. This affects the disassembly displayed in the source view as well as in the memory window.

Load/Save/Fill Memory Operations

- [Load/Save Memory](#)
- [Fill Memory](#)

Load/Save Memory

You use this command to have the IDE load and save a file into target memory. You specify the location in target memory at which the IDE loads and saves the file, and the size of that file.

To use this command, follow these steps:

1. Start a debug session.
2. Select **Debug > ARM > Load/Save Memory**.

The **Load/Save Memory** dialog box ([Figure 5.3](#)) appears.

3. Specify the parameters of the load or save operation.
4. Perform the operation.

Click the **OK** button to begin the operation. The **Progress** text box shows the current status of the operation. Click the **Cancel** button to stop the current operation. Click the Cancel button a second time to close the Load/Save Memory dialog box.

When you perform the [Load Memory](#) operation, the IDE performs these tasks:

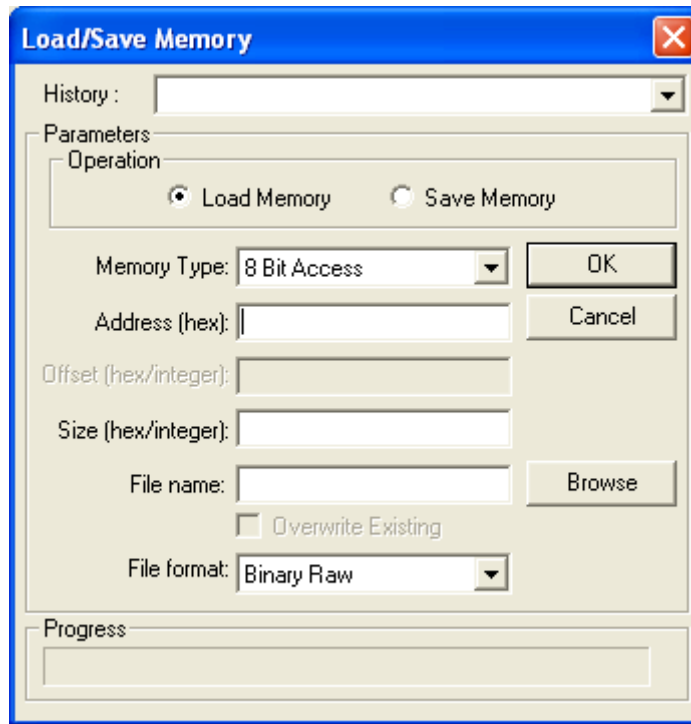
- reads data from the file that you specified
- writes the data to target memory until it reaches the end of the file or the size that you specified

When you perform the [Save Memory](#) operation, the IDE performs these tasks:

- reads memory from the target, piece by piece
- writes the memory to the file that you specified

TIP If you perform a [Save Memory](#) operation followed by a [Load Memory](#) operation, both with the same starting address and size, the memory on the target does not change. If you examine the file that the IDE writes, it looks exactly as it does in the memory view.

Figure 5.3 Load/Save Memory Dialog Box



[Table 5.3](#) explains the options in the Load/Save Memory dialog box.

Table 5.3 Load/Save Memory Dialog Box—Options

Option	Explanation
History	Use this list box to specify a load/save operation that you previously performed. The list box shows the 10 most recent operations. Each operation shows the starting address and the associated size.
Load Memory	Click this option button to specify that you want to perform a load-memory operation.
Save Memory	Click this option button to specify that you want to perform a save-memory operation.
Memory Type	Use this list box to specify the appropriate bit length for memory access during the load or save operation.

Debugging Procedures and Utilities

Load/Save/Fill Memory Operations

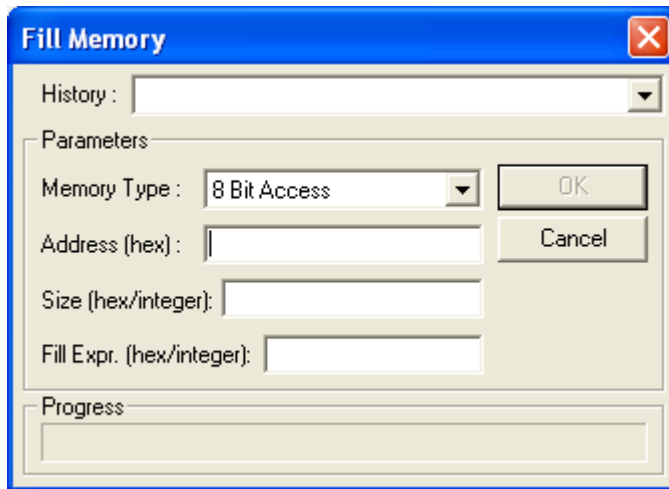
Table 5.3 Load/Save Memory Dialog Box—Options (*continued*)

Option	Explanation
Address	<p>Enter the starting address in memory at which you want to perform the load or save operation.</p> <p>Prefix your entry with 0x to have the IDE treat it as a hexadecimal value. Otherwise, the IDE treats the entry as a decimal value.</p>
Offset	<p>This option is not used for Nintendo DS development.</p>
Size	<p>Enter the number of bytes that you want to load or save to the target. Note that endianness has no effect on the actual memory that the IDE writes.</p> <p>Prefix your entry with 0x to have the IDE treat it as a hexadecimal value. Otherwise, the IDE treats the entry as a decimal value.</p>
File name	<p>Enter the full path to the file from which you want to load memory or to which you want to save memory. Click the Browse button to use a dialog box to specify the file.</p>
Overwrite Existing	<p>Select to have the IDE overwrite current data in the specified file.</p> <p>Clear to have the IDE append information to the current data in the specified file.</p> <p>Click the Save Memory option button to enable this checkbox.</p>
File Format	<p>Use to specify the appropriate data format of the file that you enter in the File name text box.</p>

Fill Memory

This command fills target memory with the result of an expression. You specify the location at which the IDE starts filling memory, and the size of that fill, in the Fill Memory dialog box ([Figure 5.4](#)).

Figure 5.4 Fill Memory Dialog Box



To use this command, follow these steps:

1. Start a debug session.
2. Select **Debug > ARM > Fill Memory**.

The **Fill Memory** dialog box appears.

3. Specify the parameters of the fill-memory operation.
4. Perform the operation.

Click the **OK** button to begin the operation. The **Progress** text box shows the current status of the operation. Click the **Cancel** button to stop the current operation. Click the Cancel button a second time to close the Fill Memory dialog box.

The IDE treats the [Fill Expr.](#) in different ways:

- If you prefix the expression with 0x, the IDE treats the set of characters as a hexadecimal value. For example, if you enter 0xBEEF, the IDE fills target memory with the hexadecimal value 0xBEEF. Entering 0xBE 0xEF has the same effect. The IDE ignores spaces in the hexadecimal number.
- If you do not prefix the expression with 0x, the IDE treats the set of characters as a decimal value and writes the equivalent hexadecimal value to target memory. For example, if you enter 10, the IDE fills target memory by writing the hexadecimal value 0x0a to each byte on the target. The IDE writes the minimum number of hexadecimal characters required to represent the decimal value. For example, entering 1024 causes the IDE to write 0x1000, not 0x00001000.

Debugging Procedures and Utilities

Load/Save/Fill Memory Operations

- If you enclose the expression in quotation marks ("*expression*"), the IDE treats the set of characters as a literal ASCII string. Use this method to include space characters in the filled memory.

[Table 5.4](#) shows various examples of how the IDE interprets the value that you enter in the [Fill Expr.](#) text box. If the expression results in a value that exceeds the size that you specify, the IDE truncates the value so that it fits the specified size. The IDE grays out the **OK** button if the current [Fill Expr.](#) entry is invalid.

Table 5.4 Fill Expression—Examples

To fill memory with this....	...Enter this as the Fill Expr.	Resulting Memory
All zeros	0 or 0x0	00000000000000000000
Any hex number	0x12345678 or 0x12 0x34 0x56 0x78	12345678123456781234
Any single ASCII character (0x)	0x00 (ASCII number in hex format)	00000000000000000000
Any single ASCII character (no 0x)	FF	FFFFFFFFFFFFFFFFFFFF
Any string with spaces (quoted)	"abc b"	61626320636162632063
Any string without spaces	qqq	Error. You must enclose strings in quotation marks, like this: "qqq"
A hex string or number string without 0x or spaces	def	0def0def0def0def0def
Hex number with zeroes	0x00001000	00001000000010000000
Odd number of hex digits	0x123	01230123012301230123
Set of Hex Numbers	0x12345678 0xabcdef11	12345678abcdef1112345
Hex number with zeroes	0x0234	02340234023402340234
String as a number (must enclose in quotation marks)	"1234"	31323334313233343132

Table 5.4 Fill Expression—Examples (*continued*)

To fill memory with this....	...Enter this as the Fill Expr.	Resulting Memory
Combo string + hex	ab 0x1 (IDE ignores space)	ab01ab01ab01ab01ab01
Combo string with space + hex	"ab " 0x1 (IDE uses space inside quotation marks)	61622001616220016162
Invalid value	1s00000 or 0xzz	Error. The IDE grays out the OK button until you correct the invalid value.

[Table 5.5](#) explains the options in the Fill Memory dialog box.

Table 5.5 Fill Memory Dialog Box—Options

Option	Explanation
History	Use this list box to specify a fill-memory operation that you previously performed. The list box shows the 10 most recent operations. Each operation shows the starting address and the associated size.
Memory Type	Use this list box to specify the appropriate bit length for memory access during the fill operation.
Address	Enter the starting address at which you want to fill memory. Prefix your entry with 0x to have the IDE treat it as a hexadecimal value. Otherwise, the IDE treats the entry as a decimal value.

Table 5.5 Fill Memory Dialog Box—Options (*continued*)

Option	Explanation
Size	<p>Enter the number of bytes that you want to fill in target memory. Note that endianness has no effect on the actual memory that the IDE writes.</p> <p>Prefix your entry with 0x to have the IDE treat it as a hexadecimal value. Otherwise, the IDE treats the entry as a decimal value.</p>
Fill Expr.	<p>Enter a set of characters that you want to have the IDE use to fill the target memory. Starting at the address you specify, the IDE repeatedly writes these characters until it reaches the size (total number of bytes) you specify.</p> <p>Note that endianness has no effect on the actual memory that the IDE writes. The IDE does not perform byte-swapping of the characters that you enter.</p>

Debugging Stand-alone ELF (.nef) Files

To work with an Extended Linker Format (ELF) file, you can drag and drop it into the IDE. The IDE creates a default project that contains the ELF file. You can use this project to download the ELF file to the target hardware and debug it.

NOTE You can only debug ELF binaries that contain debugger symbolics.

Multi-Core Debugging Options

The Nintendo DS system contains two ARM processors, an ARM7 and an ARM9. As each can run code independently from the other, the CodeWarrior IDE also lets you debug each independently.

- [Displaying the ARM7 Thread Window](#)
- [Stopping and Running Simultaneous Processes](#)

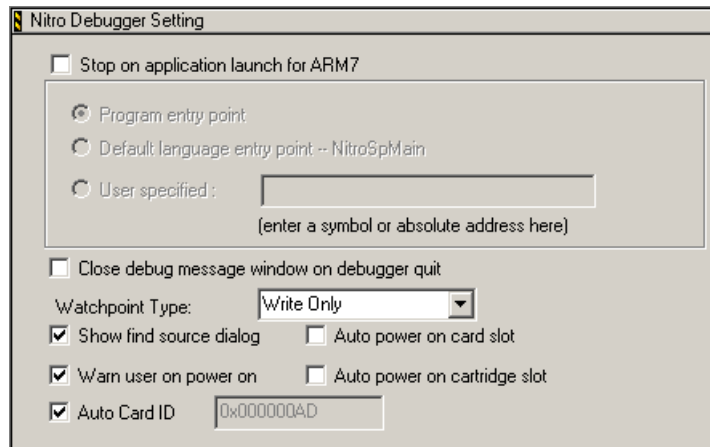
Displaying the ARM7 Thread Window

When we tell the CodeWarrior IDE to launch the debugger, the IDE displays the ARM9 thread window. We can instruct the IDE to display the ARM7 thread window at the same

time by indicating in the Nitro Debugger Setting panel ([Figure 5.5](#)) that we wish to stop the ARM7 code upon launch.

An alternative method of launching a thread window for the ARM7 processor is to open the system browser and right-click the ARM7 processor to **Attach to Process** (see [“System Browser \(Processor and Thread Browser\)”](#).)

Figure 5.5 Nitro Debugger Setting panel



Stopping and Running Simultaneous Processes

The break and run commands in the thread window only affect individual threads and processes. You can also start and stop individual processes in the system browser window (see [“System Browser \(Processor and Thread Browser\)”](#).)

If you wish to stop or resume all the processes at the same time, then you must use the options under the **Multi-Core Debug** menu, **Run All**, **Stop All**, or **Kill All**.

Cache Viewer

The cache viewer ([Figure 5.6](#)) lets you view and modify the contents of the Nitro ARM9 instruction and data cache. To open the Cache Viewer window:

1. Click the ARM9 thread window to give it focus.
2. Select **Data > View Cache > Nitro ARM9 Instruction Cache** or **Data > View Cache > Nitro ARM9 Data Cache**

Figure 5.6 The Cache Viewer window

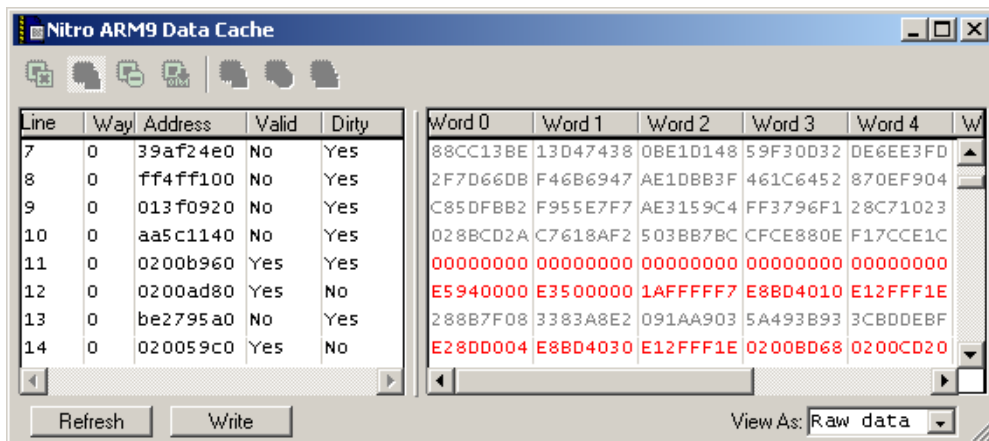


Table 5.6 Cache Status Descriptions

Valid	Dirty	Description
No	Yes	Invalid - the cache line does not match main memory and must be discarded.
Yes	No	Valid -the cache line matches main memory.
Yes	Yes	Dirty - the cache line contains new data that you must write back to main memory.


To change the value of the cache contents, double-clicking any word value and enter the new value.

The **Write** button commits your changes to the cache.

The **Refresh** button updates the display with the current contents of the cache.

The **View As** listbox lets you view the cache contents as either disassembly or raw hex data.

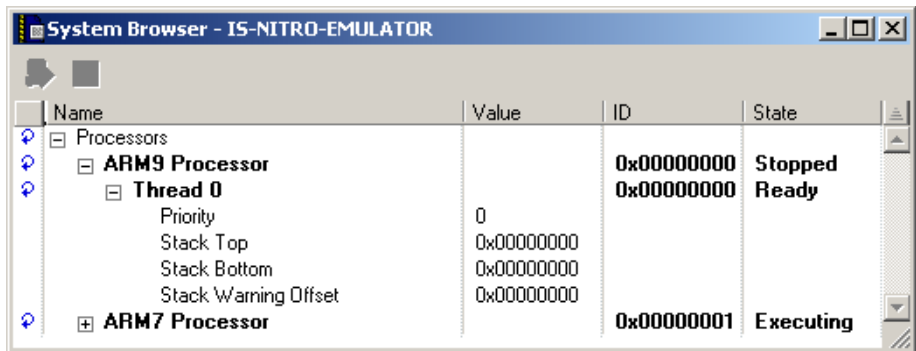
To invalidate the cache, click the **Invalidate Cache**  button.

To flush the cache to memory, click the **Flush Cache**  button.

System Browser (Processor and Thread Browser)

The System Browser ([Figure 5.7](#)) lets you view and control the state of each processor and program thread. To open the System Browser window, select **View >System > IS NITRO Debugger**.

Figure 5.7 System Browser window



Click the hierarchy control box to the left of a Processor or Thread name to view the contents.

Right-click a processor or thread to open a drop-down menu:

- Processor drop-down menu options
 - Attach to Process
Attaches the debugger to the selected process. You may also double-click the processor name to attach to process.
 - Run
Executes the selected process if it is currently stopped.
 - Kill
Kills the selected process.
- Thread drop-down menu options
 - Stack Crawl Window
Opens a debugger thread window (also called a stack crawl window) for the selected thread.
 - Run
Executes the selected thread if it is currently stopped

- Break
Halts the selected thread if it is currently executing.

Using the Profiler

The CodeWarrior IDE includes a simple profiler that you can use to analyze the timing of your code. To activate the profiler:

1. Check the **Generate code for profiling** checkbox in the [ARM Processor Panel](#)
2. Use `#pragma profile on` directive before the function definition and the `#pragma profile off` directive after the function definition
3. Use the `-profile` option or the `#pragma` directives with the command-line compiler
4. Add the profiler libraries to your project. These libraries are in:
`CodeWarrior\ARM_EABI_Support\Profiler\Lib\`
where *CodeWarrior* is the path to your CodeWarrior installation.
5. In your source code, you must `#include` the header file `Profiler.h`. This file is in:
`CodeWarrior\ARM_EABI_Support\Profiler\include\`

These functions calls control the profiler:

- `ProfilerInit` initializes the profiler
- `ProfilerClear` removes existing profiling data
- `ProfilerSetStatus` turns profiling on (1) and off (0)
- `ProfilerDump ("filename")` dumps the profile data to a profiler window or to a file named *filename*
- `ProfilerTerm` exits the profiler

[Listing 5.1](#) shows an example of using these calls in source code.

Listing 5.1 Using the Profiler in Source Code

```
#include "profiler.h"

void main()
{
    InitHeap();
    OS_InitTick();      // initialize NITRO OS Tick feature

    // initialize Profiler
    ProfilerInit(collectDetailed, bestTimeBase, 20, 20);
    ProfilerClear();
    // enable profile
    ProfilerSetStatus(1);

    Function_To_Be_Profiled();

    //disable profiler
    ProfilerSetStatus(0);
    // output profiler result
    ProfilerDump("profiledump");
    // terminate profiler
    ProfilerTerm();

    OS_Terminate();
}
```

The profiler libraries use an external function called `getTime` to measure the actual execution time. The source-code file `timer.c` shows a semihosting example of using the `getTime` function. This file is in

`CodeWarrior\ARM_EABI_Support\Profiler\Support\`
where *CodeWarrior* is the path to your CodeWarrior installation.

The `timer.c` code is well suited for execution in simulators. If you want to execute the code on an Nintendo DS board instead, you can modify the code to take advantage of hardware-based timers.

Using Watchpoints

Watchpoint operations are conducted as described in the *IDE Users Guide*, with the following exceptions:

- Nintendo DS debugging only supports one watchpoint.

Debugging Procedures and Utilities

Activating DS Flash Card and GBA Cartridge Slots

- If you are using the IS NITRO EMULATOR as your debug target, you can set the watchpoint properties in the NITRO Debugger settings panel. Please see [“NITRO Debugger Setting Panel”](#).
- If you are using the Nitro Emulator (Ensata) as your debug target, you can set the watchpoint properties by editing the **Nitro Emulator Connection**. Please see [“Remote Debugging Panel”](#).

Activating DS Flash Card and GBA Cartridge Slots

There are two ways to activate a DS Flash Card or GBA Cartridge from within the CodeWarrior IDE.

- Automatically

Check the **Auto Power on card slot** and **Auto power on cartridge slot** options in the Nitro Debugger Settings panel

- Manually

Start a CodeWarrior debug session.

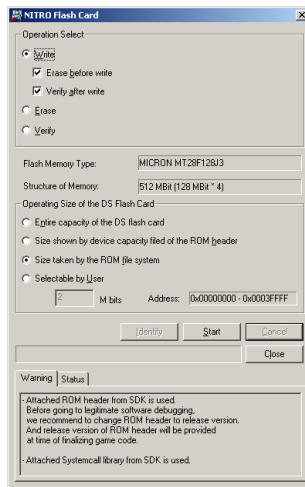
Select **NITRO > Enable External Card** or **NITRO > Enable Cartridge Slot** — an audible clicking noise signals the successful activation of the slot.

CAUTION	Never insert or remove a card or cartridge while the IS NITRO EMULATOR is on. Doing so may damage your hardware.
----------------	--

Writing Emulator Memory to DS Flash Card

The **NITRO Flash Card** dialog box ([Figure 5.8](#)) lets you write the contents of emulation memory to the DS Flash card, or delete the existing contents of the flash card. To access the NITRO Flash Card dialog box, select **Nitro > Flash....**

Figure 5.8 NITRO Flash Card Dialog Box



The options in this panel are described in the IS-NITRO Debugger help document, in the sections *Operation > Debugger > Using the Nintendo DS flash card*, and *Explanation of Functions > The Dialog Boxes > The [Nintendo DS Flash Card] Dialog Box*.

Writing User Contents to Backup Device

The **Backup Device** dialog box ([Figure 5.9](#)) lets you write the contents of a backup device on the flash card to a binary file, which you may restore at a later time. This function can also be used to load a ROM image binary into the backup device so that you can run your program on a consumer Nintendo DS unit for test purposes.

To access the **Backup Device** dialog box, select **Nitro > Backup Memory**.

Figure 5.9 Backup Device Dialog Box

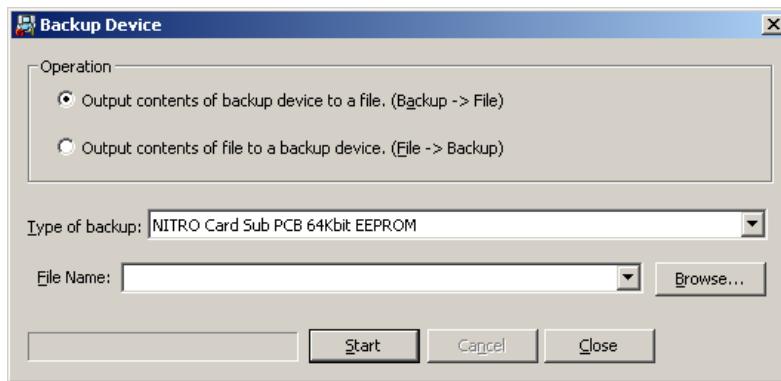


Table 5.7 Backup Device Dialog Box—options

Option	Explanation
Output contents of backup device to a file.	Select this option to transfer data from a flash card to a file.
Output contents of file to a backup device	Select this option to transfer data from a file to the flash card.
Type of backup	Specify the model of your flash card from the list.
File Name	Specify the filename for the transfer operation.

View ROM Header

To view the ROM header information, select **NITRO > View Rom Header** ([Figure 5.10](#)).

Figure 5.10 View ROM Header Dialog Box

ROM Registration Data

Software Title: NITRO

Initial Code: NTRJ Maker Code: 01 Product Code: 0x00

Device Type: 0x00 Device Code: 0x00 ROM Version: 0x00

ROM Control Type: 0x00066000 ROM Binary: 0x001B0FFB Mask ROM: 0x051E

Nintendo Logo Data: 0x2474

ROM Registration CRC:

ARM9 Static Module

ROM Offset: 0x00004000 Entry Address: 0x02000950

RAM Address: 0x02000000 ROM Size: 0x000094D8

ARM7 Static Module

ROM Offset: 0x0000C000 Entry Address: 0x02380000

RAM Address: 0x02380000 ROM Size: 0x0002B118

File Name Table

ROM Offset: 0x0003E0D0 Table Size: 0x00000009

File Allocation Table

ROM Offset: 0x00036000 Table Size: 0x00000000

Overlay Header Table

ARM9 ROM Offset: 0x00000000 ARM9 Table Size: 0x00000000

ARM7 ROM Offset: 0x00000000 ARM7 Table Size: 0x00000000

Notice:

OK

One Time PROM

The Nitro SDK 2.1 makerom tool has an option to make a One Time PROM binary. There is an example One Time PROM CodeWarrior project in:

```
{CW}\Examples\NITRO\SDK2.1\ARM9-TS\SDK-demos\card
```

To make valid One Time PROM binaries with CodeWarrior:

Set the RomSpeedType parameter in the Makerom postlinker panel:

```
MAKEROM_ROMSPEED= "1TROM"
```

To run or debug a One Time PROM binary with a suitable Card ID:

Enable the Set Card ID automatically option in the NITRO Debugger Setting panel.

Assembler

This chapter describes aspects of the CodeWarrior Nintendo DS stand-alone assembler that are not documented in the *Assembler Reference* manual. The sections in this chapter are:

- [Label Syntax](#)
- [Debugging Assembly Source](#)
- [GNU Assembler Compatibility](#)

Label Syntax

The label and =label instructions have different functions.

- `ldr r1, label`
Loads the contents of the address label
- `ldr r1,=label`
Loads the actual address value of the label

Listing 6.1 Example usage of label and =label

main.c

```
extern void asmfunc(void;

int main()
{
    asmfunc();
}
```

asmfunc.asm

```
.public asmfunc
.public __SP_INIT
SVC_Stack_Size .equ 0x100
.text
asmfunc:
    ldr r1,=@SVC_Stack
    ldr r0,=@USR_Stack
    ldr r1,=@SVC_Stack
    ldr r0,=@USR_Stack
```

Assembler

Debugging Assembly Source

```
    ldr r2, [r1]
    ldr r3, [r0]
    mov pc, lr
@SVC_Stack
    .long __SP_INIT
@USR_Stack
    .long __SP_INIT-SVC_Stack_Size
```

Debugging Assembly Source

Some assembler directives have a slightly different usage than the explanations given in the *Assembler Reference*:

- [.function](#)
- [.line](#)
- [.file](#)

.function

To ensure that the debugger can display your assembly source code while debugging, you must use the `.function` directive to identify the NitroMain function.

Listing 6.2 Identifying NitroMain Using the .function Directive

```
.function "NitroMain", NitroMain, NitroMain_end-NitroMain
NitroMain:
    and r1, r2, #1
    orr r1, r2, #1
    tst r1, r2
    mov r0, r2
lp:
    b lp
NitroMain_end:
```

.line

By default, the assembler generates all the necessary source line information for debugging purposes. If you wish to exercise line-by-line control over the source line information, you may use the `.line` directive. If there is at least one `.line` directive in your code, the assembler generates source line information only for the code located immediately after the directive. All other source line information will be suppressed.

.file

If you do not have `.line` directives in your code, using a `.file` directive will prevent the assembler from generating DWARF `.debug_line` information, and lines in the source file will not appear during debugging.

If you do have `.line` directives in your code, it is not necessary to use the `.file` directive, as the `.function` directive will provide source-level debugging.

GNU Assembler Compatibility

The Codewarrior Assembler supports several GNU-format assembly language extensions.

- [GNU Compatible Syntax option](#)
- [Supported Extensions](#)
- [Unsupported Extensions](#)

GNU Compatible Syntax option

Only in cases where GNU's assembler format conflicts with that of the CodeWarrior assembler does the **GNU Compatible Syntax** option have any effect (see "[ARM Assembler Panel](#)"). Specifically:

- Defining Equates
Whether defined using `.equ` or `.set`, all equates can be re-defined.
- Ignored directives
The `.type` directive ignored.
- Undefined Symbols
Undefined symbols are automatically treated as imported
- Arithmetic Operators
`<` and `>` mean left-shift and right-shift instead of less than and greater than.
`!` means bitwise-or-not instead of logical not.
- Precedence Rules
Precedence rules for operators are changed to be compatible with GNU rather than with C.
- Local Labels
Local labels with multi-number characters are supported (example: `"1000:"`). There is no limit on the number of digits in the label name. Multiple instances of the label are allowed. When referenced, you get the nearest one - forwards or backwards depending on whether you append `'f'` or `'b'` to the number.

Assembler

GNU Assembler Compatibility

- Numeric Constants

Numeric constants beginning with 0 are treated as octal.

- Semicolon Use

Semicolons can be used as a statement separator.

- Unbalanced Quotes

A single unbalanced quote can be used for character constants. For example: `.byte 'a`

Supported Extensions

Some GNU extensions are always available, regardless whether you set the **GNU compatible syntax** option in the ARM Assembler settings panel (see [“ARM Assembler Panel”](#)). Specifically:

- Lines beginning with `#` `*` or `;` are always treated as comment, even if the comment symbol for that assembler is something different.
- Escape characters in strings extended to include `\xNN` for hex digits and `\NNN` for octal.
- Binary constants may begin with `0b`.
- Supports the GNU macro language, with macros defined by:

```
.macro      name, arg1 [=default1], arg2...s1
...
.endm
```

Arguments may have default values as shown, and when called may be specified by value or position. See the GNU documentation for details.

- New or enhanced directives (see GNU documentation for details)

Table 6.1 Supported GNU Assembler Directives

Directive	Description	Comment
<code>.abort</code>	End assembly	Supported
<code>.align N,[pad]</code>	Align	Now accepts optional padding byte
<code>.app-file name</code>	Source name	Synonym for <code>.file</code>
<code>.balign[w] N,[pad]</code>	Align	Align to N (with optional padding value)
<code>.comm name,length</code>	Common data	Reserve space in BSS for global symbol
<code>.def</code>	Debugging	Accepted but ignored
<code>.desc</code>	Debugging	Accepted but ignored

Table 6.1 Supported GNU Assembler Directives

Directive	Description	Comment
.dim	Debugging	Accepted but ignored
.eject	Eject page	Accepted but ignored
.endr	End repeat	See .irp, .irpc
.endef	Debugging	Accepted but ignored
.fill N,[size],[val]	Repeat data	Emit N copies of width 'size', value 'val'
.hword val..	Half-word	Synonym for .short
.ident	Tags	Accepted but ignored
.ifnotdef name	Conditional	Synonym for .ifndef
.include name	Include file	Now accepts single, double or no quotes
.int val..	Word	Synonym for .long
.irp name,values	Repeat	Repeat up to .endr substituting values for name
.irpc name,chars	Repeat	Repeat up to .endr substituting chars for name
.lcomm name,length	Local common	Reserve length bytes in bss
.lflags	Ignored	Accepted but ignored
.ln lineno	Line number	Synonym for .line
.list	Listing on	Switch on listing
.local name	Local macro var	Declare name as local to macro
.macro name, args..	Macros	Supports Gnu syntax, default values, etc
.nolist	Listing off	Disable listing
.org pos,fill	Origin	Now allows fill value to be specified
.p2align[wl] N[,pad]	Align	Align to 2**N, using pad value 'pad'
.psize	Page size	Accepted but ignored
.rept N	Repeat	Repeat block up to .endr N times
.sbttl	Subtitle	Accepted but ignored

Table 6.1 Supported GNU Assembler Directives

Directive	Description	Comment
<code>.scl</code>	Debugging	Accepted but ignored
<code>.size name,N</code>	Set size	Set size of name to N
<code>.skip N[,pad]</code>	Space	Skip N bytes, pad with 'pad'
<code>.space N[,pad]</code>	Space	Skip N bytes, pad with 'pad'
<code>.stabd</code>	Debugging	Accepted but ignored
<code>.stabs</code>	Debugging	Accepted but ignored
<code>.str "string"</code>	Constant string	Synonym for <code>.asciz</code>
<code>.string "string"</code>	Constant string	Synonym for <code>.asciz</code>
<code>.tag</code>	Debugging	Accepted but ignored
<code>.title</code>	Title	Accepted but ignored
<code>.type</code>	Debugging	Ignored in Gnu mode
<code>.val</code>	Debugging	Accepted but ignored
<code>.word</code>	Word	Synonym for <code>.long</code>

Unsupported Extensions

Among the GNU extensions that the CodeWarrior Assembler does not support are:

- Sub-sections (such as `".text 2"`). The sub-section number will be ignored.
As a workaround, you can create your own sections with the `.section <name>` directive. You may have an arbitrary number of text subsections with the names `.text1`, `.text2`, etc.
- Assignment to location counter (such as `". = .+4"`)
As a workaround, you can advance the location counter with `.space <expr>`
- Empty expressions defaulting to 0. Example:
`".byte ,"` equivalent to `".byte 0,0"`
There is no workaround for this. You must always supply the arguments.
- `.linkonce` directive

The linker automatically detects logically-identical sections, and uses the following factors to determine whether to keep only one or both in the final image:

- the binding of the symbols associated with each section
 - the location of these two sections. For example, are the sections in the same overlay or overlay group? Is one in main, and the other in an overlay group? For more information, see [“Determining Valid Intersegment Calls”](#).
- `.octa`

We do not support 16-byte numbers directly. As a workaround, you may use consecutive `.long` directives to build a large number in memory.

- `.quad`

We do not support eight-byte numbers directly. As a workaround, you may use consecutive `.long` directives to build a large number in memory.

Assembler

GNU Assembler Compatibility

C and C++ Compiler

This chapter describes the CodeWarrior C/C++ NINTENDO DS compiler. The sections in this chapter are:

- [Number Formats](#)
- [Declaration Specifiers](#)
- [Register Variables](#)
- [Pragmas](#)
- [Attribute Syntax](#)

This chapter contains references to Appendix A of the “Reference Manual,” of *The C Programming Language, Second Edition* (Prentice Hall) by Kernighan and Ritchie. [Table 7.1](#) lists other useful compiler and linker documentation.

Table 7.1 Other compiler and linker documentation

For this topic	Refer to
How the CodeWarrior IDE implements the C/C++ language	<i>C Compilers Reference</i>
Using C/C++ Language and C/C++ Warnings settings panels	<i>C Compilers Reference</i> , “Setting C/C++ Compiler Options” chapter
Controlling the size of C++ code	<i>C Compilers Reference</i> , “C++ and Embedded Systems” chapter
Using compiler pragmas	<i>C Compilers Reference</i> , “Pragmas and Symbols” chapter
Initiating a build, controlling which files are compiled, handling error reports	<i>IDE User’s Guide</i> , “Compiling and Linking” chapter
Information about a particular error	<i>Error Reference</i> , which is available online

Number Formats

This section explains how the CodeWarrior C/C++ compilers implement integer and floating-point types for NINTENDO DS processors. You also can read `limits.h` for more information on integer types, and `float.h` for more information on floating-point types.

This section has these topics:

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

Integer Formats

[Table 7.2](#) shows the size and range of the integer types for the NINTENDO DS compiler.

Table 7.2 NINTENDO DS Integer Types

For this type	Option setting	Size	Range
bool	n/a	8 bits	true or false
char	Use Unsigned Chars is off (see language preferences panel in the <i>C Compilers Reference</i>)	8 bits	-128 to 127
	Use Unsigned Chars is on	8 bits	0 to 255
signed char	n/a	8 bits	-128 to 127
unsigned char	n/a	8 bits	0 to 255
short	n/a	16 bits	-32,768 to 32,767
unsigned short	n/a	16 bits	0 to 65,535
int	n/a	32 bits	-2,147,483,648 to 2,147,483,647
unsigned int	n/a	32 bits	0 to 4,294,967,295
long	n/a	32 bits	-2,147,483,648 to 2,147,483,647
unsigned long	n/a	32 bits	0 to 4,294,967,295

Table 7.2 NINTENDO DS Integer Types (*continued*)

For this type	Option setting	Size	Range
long long	n/a	64 bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long	n/a	64 bits	0 to 18,446,744,073,709,551,615

Floating-Point Formats

[Table 7.3](#) shows the sizes and ranges of the floating point types for the NINTENDO DS compiler.

Table 7.3 NINTENDO DS Floating Point Types

Type	Size	Range
float	32 bits	1.17549e-38 to 3.40282e+38
short double	64 bits	2.22507e-308 to 1.79769e+308
double	64 bits	2.22507e-308 to 1.79769e+308
long double	64 bits	2.22507e-308 to 1.79769e+308

Declaration Specifiers

A declaration specifier has the form:

```
__declspec(qualifierName) dataType symbolName
```

where *qualifierName* is the declspec qualifier, *dataType* is the data type of the symbol, and *symbolName* is the name of the symbol.

[Listing 7.1](#) show examples of using `__declspec()`.

Listing 7.1 Using `__declspec()`

```
__declspec(sbss) int    Large_Array_In_Small_Data_Section[1000];
__declspec(data) bool   Small_Object_In_Large_Data_Section;
__declspec(weak) void   replaceableFunction() {...}
```

The valid declaration specifiers are listed in [Table 7.4](#).

Table 7.4 Declaration specifiers

This declaration specifiers	...does this
<code>__declspec(data)</code>	stores symbol in the .data section.
<code>__declspec(sdata)</code>	stores symbol in the .sdata section.
<code>__declspec(byte_access)</code>	type modifier; objects of this type may be safely accessed with 8-bit access instructions.
<code>__declspec(no_byte_access)</code>	type modifier; objects of this type may not be accessed with 8-bit access instructions.
<code>__declspec((interrupt("flag")))</code>	sets the interrupt flag as one of the following: <ul style="list-style-type: none">• IRQ• FIQ• SWI• ABORT• UNDEF
<code>__declspec(weak)</code>	creates a weak link symbol whose definition can be overwritten by a duplicate symbol (for example, from a library)

Register Variables

The compiler automatically allocates local variables and parameters to registers based on to how frequently they are used and how many registers are available. If you are optimizing for speed, the compiler gives preference to variables used in loops.

The compiler also gives preference to variables declared as `register`, but does not automatically assign them to registers. For example, the compiler is more likely to place a variable from an inner loop in a register than a variable declared `register`. See also: K&R, §A4.1, §A8.1

Pragmas

[Table 7.5](#) lists the pragmas supported for NINTENDO DS development that are documented in the *C Compilers Reference*. There are other pragmas, unique to Nintendo DS development, that are documented later in this chapter ([Table 7.6](#)).

Table 7.5 Pragmas for NINTENDO DS Development

align_array_members	ANSI_strict
ARM_conform	auto_inline
bool	check_header_flags
cplusplus	cpp_extensions
dont_inline	dont_reuse_strings
enumsalwaysints	exceptions
extended_errorcheck	fp_contract
global_optimizer	has8bytebitfields
ignore_oldstyle	longlong
longlong_enums	mark
no_register_save_helpers	once
only_std_keywords	optimize_for_size
optimizewithasm	peephole
pop	precompile_target
push	readonly_strings
require_prototypes	RTTI
static_inlines	syspath_once
trigraphs	unsigned_char
unused	warning_errors
warn_emptydecl	warn_extracomma
warn_hidevirtual	warn_illpragma
warn_implicitconv	warn_possunwant
warn_unusedarg	warn_unusedvar
wchar_type	

Table 7.6 Additional Pragmas for Nintendo DS Development

#pragma	Corresponding macro	Corresponding __option()
allow_byte		
ASHLA		ASHLA
avoid_byte		
dead_stripping		dead_stripping
define_section <i>sectname</i> <i>istring</i> [<i>ustring</i>] [<i>addrmode</i>] [<i>accmode</i>]		
generic_symbol_names		
interworking	__APCS_INTERWORKING	interworking
little_endian	! __BIG_ENDIAN	littleendian
profile		profile
section <i>sectname</i> (begin end)		
thumb	__thumb	thumb
warn_byte		

allow_byte

This pragma allows 8-bit access to local and global objects.

```
#pragma allow_byte none|local|global|both|reset
```

ASHLA

```
#pragma ASHLA on|off|reset
```

This pragma determines whether to enable ARM Shared Library Architecture support.

Use the `reset` option to have the IDE use the settings in the [ARM Processor Panel](#) to determine whether to enable ARM Shared Library Architecture support.

avoid_byte

This pragma replaces 8-bit memory access with 16/32-bit memory access for the 8-bit instruction types you specify

```
#pragma avoid_byte none|strb|all|reset
```

dead_stripping

```
#pragma dead_stripping on|off|reset
```

This pragma determines whether to pool constants from all functions in a file. The NINTENDO DS linker deadstrips unused code and data only from files compiled by the CodeWarrior C/C++ compiler. The linker never deadstrips assembler relocatable files and C/C++ object files built by other compilers. Deadstripping is particularly useful for C++ programs. Libraries (archives) 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. The linker always ignores completely unreferenced object files.

There are, however, situations where there are symbols that you don't want dead-stripped even though they are never used. See [“Force Active Symbols”](#) for information on how to prevent dead-stripping of unused symbols.

Use the `reset` option to have the IDE use the settings in the [ARM Processor Panel](#) to determine whether to enable deadstripping support.

define_section

```
#pragma define_section sectname istring [ ustring ]  
[ addrmode ] [ accmode ]
```

This pragma defines a section. You can use the [section](#) pragma to reference the section in your source code.

sectname

The *sectname* parameter is the identifier by which you reference the section in your source code.

istring

The *istring* parameter is the section name string for *initialized* data assigned to the section.

`ustring`

The optional *ustring* parameter is the Executable and Linkable Format (ELF) section name for *uninitialized* data assigned to the section.

`addrmode`

The optional *addrmode* parameter indicates how the linker addresses the section. You set this parameter to one of the following values:

- `abs32`
32-bit absolute addressing mode. Use for applications where all memory addresses are known at link time.
- `pcrel32`
32-bit PC-relative mode. Use for position-independent code (PIC) sections.
- `sbrel32`
32-bit Static Base register (SB) -relative mode. Use for position-independent data (PID) sections.
- `sbrel12 - 12-bit SB relative`
12-bit SB-relative mode. Use for small data sections. Sections addressed via this mode must be placed within 4Kb of the address pointed by the SB register. These "small data" sections allow the compiler to generate more efficient sequences for loading and storing global variables.

`accmode`

The optional *accmode* parameter indicates the attributes of the section. You set this parameter to one of the following values:

- `R`
- `RW`
- `RX`
- `RWX`

generic_symbol_names

`#pragma generic_symbol_names on|off`

This pragma obfuscates the names of static symbols within binaries (such as libraries) as protective measure against unauthorized persons disassembling the binary. Such a disassembly can reveal the names of static symbols and may expose internal structures and other proprietary details.

The compiler replaces static symbol names with generic ones, for example "my_local" becomes "@1234". This feature can only work when debugging is disabled.

interworking

```
#pragma interworking on|off|reset
```

This pragma determines whether to support interworking. *Interworking* involves changing between ARM and Thumb state. You can use interworking when you write code for ARM architectures that support the 16-bit Thumb instruction set. Interworking works with code that follows the ARM/Thumb Procedure Call Standard (ATPCS).

When you enable interworking in your project, ARM routines can return to a Thumb-state caller, and Thumb routines can return to an ARM-state caller. The linker generates the code, or *veneers*, that changes between ARM and Thumb state

If you select the interworking option, you can call a routine in another module without considering which instruction set it uses. The linker handles creating veneers and patching call sites. This works for compiled or assembled code.

For ARM processors based on v5t and later cores, when you enable interworking:

- the compiler generates `bl` for general subroutine calls and `blx` for calls by way of function pointers. This variant of the `blx` instruction has a register operand.
- the linker looks up all the `bl` instructions. If the caller and callee are not in the same mode (for example, one is in ARM mode and the other is in Thumb mode), the linker replaces the `bl` instruction with `blx`. This variant of the `blx` instruction has a memory-address operand.

For ARM processors based on cores earlier than v5t, when you enable interworking:

- the compiler generates these items:
 - a `bl` instruction for the general subroutine call
 - a `bx` instruction for calls by way of function pointers from ARM functions
 - a `bl __call_via_rX` instruction for calls by way of function pointers from Thumb functions. The runtime library provides the `__call_via_rX()` functions, which essentially contain `bx rX`.
- the linker generates the appropriate veneer code when the caller and callee are not in the same mode

Use the `reset` option to have the IDE use the settings in the [ARM Processor Panel](#) to determine whether to enable interworking support.

Use the `off` option to disable interworking. For example, disable interworking if your architecture does not use the Thumb instruction set, or you write the necessary assembler code that handles changes between ARM and Thumb states.

C and C++ Compiler

Pragmas

NOTE The CodeWarrior linker automatically generates veneers for long-branch capability

little_endian

```
#pragma little_endian on|off|reset
```

This pragma determines whether to use little-endian data format for the target.

profile

```
#pragma profile on|off|reset
```

This pragma determines whether to support simple profiling.

section

```
#pragma section sectname begin|end
```

This pragma marks the beginning or end of the user-specified section named *sectname*.

thumb

This pragma determines whether to generate Thumb code.

```
#pragma thumb on|off|reset
```

warn_byte

This pragma generates error and warning messages when handling 8-bit access instructions

```
#pragma warn_byte none|all|error|reset
```

Attribute Syntax

This section explains code syntax for setting attributes::

- `__attribute__((aligned(item)))`
- `__attribute__((interrupt (flagname)))`

`__attribute__((aligned(item)))`

You can use `__attribute__ ((aligned(?)))` in several situations:

- Variable declarations
- Struct, union, or class definitions
- Typedef declarations
- Struct, union, or class members

NOTE Substitute any power of 2 up to 2^{13} (8192) for the question mark (?).

NOTE Aligned (n) is only for alignments greater than the object's minimum alignment.

Variable Declaration Examples

This section shows variable declarations that use `__attribute__ ((aligned(?)))`.

The following variable declaration aligns V1 on a 16-byte boundary:

```
int V1[4] __attribute__ ((aligned (16)));
```

The following variable declaration aligns V2 on an 8-byte boundary:

```
int V2[4] __attribute__ ((aligned (8)));
```

Struct Definition Examples

This section shows struct definitions that use `__attribute__ ((aligned(?)))`.

The following struct definition aligns all definitions of struct S1 on an 8-byte boundary:

```
struct S1 { short f[3]; }  
    __attribute__ ((aligned (8)));  
struct S1 s1;
```

The following struct definition aligns all definitions of `struct S2` on a 32-byte boundary:

```
struct S2 { short f[3]; }  
    __attribute__ ((aligned (32)));  
struct S2 s2;
```

NOTE You must specify a minimum alignment of at least 4 bytes for structs; specifying a lower number for the alignment of a struct causes alignment exceptions.

Typedef Declaration Examples

This section shows typedef declarations that use `__attribute__((aligned(?)))`. These declarations only work on global symbols.

The following typedef declaration aligns all definitions of `T1` on an 8-byte boundary:

```
typedef int T1 __attribute__ ((aligned (8)));  
T1 t1;
```

The following typedef declaration aligns all definitions of `T2` on an 32-byte boundary:

```
typedef int T2 __attribute__ ((aligned (32)));  
T2 t2;
```

Struct Member Examples

This section shows struct member definitions that use `__attribute__((aligned(?)))`. These definitions only work on global symbols.

The following struct member definition aligns all definitions of `struct S3` on an 8-byte boundary, where `a` is at offset 0 and `b` is at offset 8:

```
struct S3 {  
    char a;  
    int b __attribute__ ((aligned (8)));  
};  
struct S3 s3;
```

The following struct member definition aligns all definitions of struct `S4` on a 4-byte boundary, where `a` is at offset 0 and `b` is at offset 4:

```
struct S4 {  
    char a;  
    int b __attribute__ ((aligned (2)));  
};          /*2 has no effect, see note below*/  
struct S4 s4;
```

NOTE Specifying `__attribute__ ((aligned (2)))` does not affect the alignment of `S4` because 2 is less than the natural alignment of `int`.

`__attribute__ ((interrupt (flagname)))`

```
__attribute__  
((interrupt("IRQ" | "FIQ" | "SWI" | "ABORT" | "UNDEF")))
```

You can use `__attribute__ ((interrupt (flagname)))` to set the attributes of the interrupt flag named *flagname*. You set this parameter to one of these values: `IRQ`, `FIQ`, `SWI`, `ABORT`, `UNDEF`

Alternatively, you can use this syntax

```
__declspec ((interrupt("IRQ" | "FIQ" | "SWI" | "ABORT" | "UNDEF")))
```


Linker Issues

This section describes some of the background information on the NINTENDO DS linker and how it works. This section has these topics:

- [Linker Command File Prelinker](#)
- [Binary Converter/Compiler](#)
- [Link Order](#)
- [Deadstripping Unused Code and Data](#)

Linker Command File Prelinker

Linker command files can be difficult to create. For your convenience, we include an LCF prelinker as a pre-linker option that can generate a linker command file based upon data collected from the project and target settings.

The topics in this section are:

- [Activating the Prelinker](#)
- [Project Settings Read by the Prelinker](#)
- [Prelinker Defaults](#)
- [Limitations](#)
- [Overlay Support](#)

Activating the Prelinker

To use the prelinker, set the LCF prelinker in the target settings. Go to the **Target Settings** preference panel. Select the **NITRO LCF Generator** from the **Pre-Linker** pull-down menu.

Next, go to the **NITRO LCF Prelinker** preference panel. Click the **Always create a new LCF** checkbox.

Project Settings Read by the Prelinker

As the programmer, it is your responsibility to fill in the correct information in your project settings and target settings panels so that the LCF prelinker can read them and create the linker command file.

Linker Issues

Linker Command File Prelinker

The following is the complete list of project and target settings read by the LCF prelinker:

- Overlay group names

The group names are used in the overlay header and the memory block.

- Overlay base code addresses

The base code addresses are used in the memory block to position the section. If 0xFF is used for the base code address, the prelinker uses AFTER to dynamically calculate the address of the section instead of interpreting it as a direct memory address.

- Project Files List

The files list is used in the sections block.

- Project Overlays List

The overlays list determines link order and is used in the sections block.

Prelinker Defaults

The LCF prelinker assumes certain values for some memory addresses and symbols. We list these values here:

- Section name for main segment: main
- Address: 0x02000000
- Stack Size: 0 (means as large as possible)
- IRQ StackSize: 1024
- OverlayDefs filename: %_defs
- OverlayTable filename: %_table
- Suffix of binary output files: .sbin

Limitations

The linker command file created by the prelinker is basic. Special memory allocations and sectioning must be inserted by hand.

Overlay Support

The prelinker automatically generates all necessary overlay sections. For more information about overlays, see [“Overlays”](#).

Binary Converter/Compiler

The Binary Converter/Compiler converts binary files into an ELF-format object file. This lets you link binary files directly into your application. By default, the Binary Compiler converts data files into an ELF file with these properties:

- 1-byte alignment
- little endian
- .data section
- an output filename of filename_extension.o. (Periods and spaces in your original filename are converted to underscores.)
- symbols for the start and end of data that can be accessed from C source:
extern char _binary_filename_extension[];
extern char _binary_filename_extension_end[];

Some of this can be customized on a filetype-basis by creating a ResourceSettings.txt file:

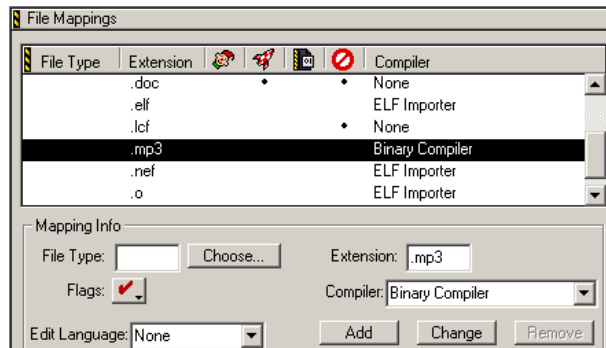
#Ext	Section	Align	Endian
jpg	.data	1	little

To specify that the binary compiler should use the settings in your ResourceSettings.txt file, enter the path to the settings file in the **Target Settings > Binary Converter** panel (see [“Binary Converter Settings Panel”](#)).

NOTE If you create a custom section (example: .jpgdata), make sure you also add this section to your program’s LCF. Otherwise you may get link errors.

The IDE invokes the binary converter when a filename extension matches a filetype that has been mapped to the **Binary Compiler** in the **Target Settings > File Mappings** panel.

Figure 8.1 The .mp3 Filetype Mapped to the Binary Compiler



Linker Issues

Link Order

To map a new filetype to the Binary Compiler in the File Mappings panel:

1. Open the **Target Settings > File Mappings** panel.
2. Type your filename extension into the **Extension** text field.
3. Clear any text in the **File Type** text field.
4. Select **Binary Compiler** from the **Compiler** listbox.
5. Click the **Add** button.

You can also invoke the Binary Converter from the command line:

```
bintoelf.exe filename.xxx [-symbolname name] [-aligndata n]
                        [-section sectionname] [-endian (little|big)]
                        [-output filename.o] [-defaultsettings file.xxx]
```

-symbolname

set symbol name

-aligndata

set data alignment

-section

set section name

-endian

set data endian mode

-output

set binary output filename

-defaultsettings

specify settings from file

Link Order

The link order for a NINTENDO DS project is specified in the Overlays view of the Project Window. For general information on setting link order, refer to the *CodeWarrior IDE Users Guide*.

C, C++, and assembly source files are processed at the same time as libraries (relocatable files (.o) and archive files (.a) are treated as libraries). In the case of a symbol being duplicately defined in both a source file and the library, the linker uses the first definition, and issues a warning about duplicated symbols.

TIP To specify that the duplicate definitions of a symbol should take precedence over the first instance, you can declare the first instance of the symbol as a weak link

symbol in your source code.

```
__declspec(weak) int foo() {...};
```

Deadstripping Unused Code and Data

Deadstripping is the process of excluding unreachable code when linking. In order to have the NINTENDO DS linker perform dead-stripping of unused code, data, and DWARF2 data, you must check the [Permit dead-stripping of functions](#) checkbox in the [ARM Processor Panel](#).

The linker deadstrips unused code, data, and DWARF2 data only from files that the CodeWarrior C/C++ compiler compiles. The linker never deadstrips assembler-relocatable files and C/C++ object files that other compilers produce.

NOTE The linker performs deadstripping on a per-function basis.

You can disable deadstripping for well-constructed projects:

- To disable deadstripping completely, clear the [Permit dead-stripping of functions](#) checkbox in the [ARM Processor Panel](#).
- To disable deadstripping for particular symbols, enter the names of those symbols in the [Force Active Symbols](#) text box in the [ARM Linker Panel](#).
- To disable deadstripping for individual linker-command-file sections, use the `KEEP_SECTION()` segment explained in [“Closure Blocks.”](#)

Deadstripping is particularly useful for C++ programming, or when you link to large, general-purpose libraries. 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. The linker always ignores completely unreferenced object files.

A `KEEP_SECTION()` segment commonly includes interrupt-vector tables because code does not directly reference them.

Linker Issues

Deadstripping Unused Code and Data

Linker Command File Specification

The CodeWarrior Extended Linker Format (ELF) 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 at the function level, and change the alignment.

The Linker Command File (LCF) provides access to all of these functions. The LCF has its own language complete with keywords, directives, and expressions, that are used to create the powerful specification for your output code.

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. A good information source is "Linkers and Loaders".
<http://linker.iecc.com/>

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:

- [Memory Segment](#) — allocates memory
- [Sections Segment](#) — defines memory contents
- [Closure Blocks](#) — prevents deadstripping specified functions

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

Memory Segment

In the memory segment, we divide our available memory into segments. The memory segment format looks like [Listing 9.1](#).

Listing 9.1 A sample MEMORY segment

```
MEMORY {  
    segment_1 (RWX): ORIGIN = 0x80001000, LENGTH = 0x19000  
    overlay_1 (RWXO): ORIGIN = 0x80020000, LENGTH = 0  
    overlay_2 (RWXO): ORIGIN = 0x80020000, LENGTH = 0  
    heap (RWU): ORIGIN = AFTER(overlay_1, overlay_2), LENGTH = 0  
    and so on...  
}
```

The (RWX) portion consists of ELF access permission flags:

- R — read
- W — write
- E — executable
- O — overlay segment
- U — unloadable segment

In the sample memory segment above:

ORIGIN... represents the start address of the memory segment.

LENGTH... represents the size of the memory segment.

If we cannot predict how much space a segment will occupy, we can use the function AFTER and LENGTH = 0 (unlimited length) to fill in the unknown values.

For a detailed examination of the MEMORY segment, please read [“MEMORY.”](#)

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 9.2](#). In this sample segment,

Listing 9.2 A sample SECTIONS segment

```
SECTIONS {
    .section_name : #the section name is for your reference
    {
        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”](#).

Closure Blocks

The linker is very good at deadstripping unused code and data. However, we may sometimes need to keep unreferenced symbols in the output file. For example, interrupt handlers 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:

Symbol-level

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

Listing 9.3 A sample symbol-level closure block

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

Section-level

- [KEEP_SECTION](#)
- [REF_INCLUDE](#)

KEEP_SECTION

KEEP_SECTION keeps a section even if the file containing the section is not referenced ([Listing 9.4](#)).

Listing 9.4 A KEEP_SECTION section-level closure block

```
KEEP_SECTION { .interrupt1, .interrupt2 }
```

REF_INCLUDE

REF_INCLUDE keeps a section in the link only if the file containing the section is referenced. One good use for this is to include version numbers ([Listing 9.5](#)).

Listing 9.5 A REF_INCLUDE section-level closure block

```
REF_INCLUDE { .version }
```

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. Topics discussed in this section are:

- [Alignment](#)
- [Arithmetic Operations](#)
- [Comments](#)
- [Deadstrip Prevention](#)
- [Exception Tables](#)
- [Alphabetical Keyword Listing](#)
- [File Selection](#)
- [Function Selection](#)
- [Stack and Heap](#)
- [Static Initializers](#)
- [Writing Data to Memory](#)
- [Writing Data to Memory from a File](#)

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”](#) and [“ALIGNALL”](#).

Arithmetic Operations

You may use standard C arithmetic and logical operations when you define and use symbols in the linker command file. [Table 9.1](#) shows the order of precedence for each operator. All operators are left-associative. To learn more about C operators, refer to the *C Compiler Reference*.

Table 9.1 Arithmetic Operators

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
```

There is a predefined section for comments in your LCF. It is `.comment`. It is not necessary to list the `.comment` section in the MEMORY block.

```
.comment_section :
{
    * (.comment)
} > .comment
```

You can also use the `WRITES()` directive to place a comment.

```
.comment
{
    * (.comment)
    WRITES("The goose is cooked");
} > .comment
```

Deadstrip Prevention

Linkers remove 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.”](#) [“KEEP_SECTION.”](#) and [“REF_INCLUDE.”](#)

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 9.6 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 known 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:

Linker Command File Specification

Linker Command File Syntax

```
_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;  
_fudgefactorospace = 0X123245EE;  
_hexonyou = 0xFFEE;
```

To create a negative decimal 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.”](#)

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.”](#)

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 9.7](#)).

Listing 9.7 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 */
```

Linker Command File Specification

Linker Command File Syntax

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

Listing 9.8 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 compiler generates two ELF sections that contain information about the static initializers. The ELF sections are:

- **.init** contains a list of all the static initializers
- **.ctor** contains a pointer to the static initializers.

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). Note that the runtime expects the static initializer table to be null-terminated.

```
__sinit__ = .;
*(.ctor)
WRITEW(0);
```

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 to Memory

You can write data directly to memory using the `WRITEEx` commands in the linker command file.

- `WRITEB` writes a byte
- `WRITEH` writes a two-byte halfword
- `WRITEW` writes a four-byte word.

- `WRITES` writes a string. The data is inserted at the section's current address.

The example in [Listing 9.9](#) is similar to the technique used to insert overlay headers.

Listing 9.9 Embedding data directly into the output.

```
.example_data_section :  
{  
    WRITEB 0x48; /* 'H' */  
    WRITEB 0x69; /* 'i' */  
    WRITEB 0x21; /* '!' */  
    WRITES ("Hi!")  
} > example_data_section
```

Writing Data to Memory from a File

The [INCLUDE](#) command lets you insert a complete binary file into the linker output. You may put the binary data file anywhere in memory, except for executable sections.

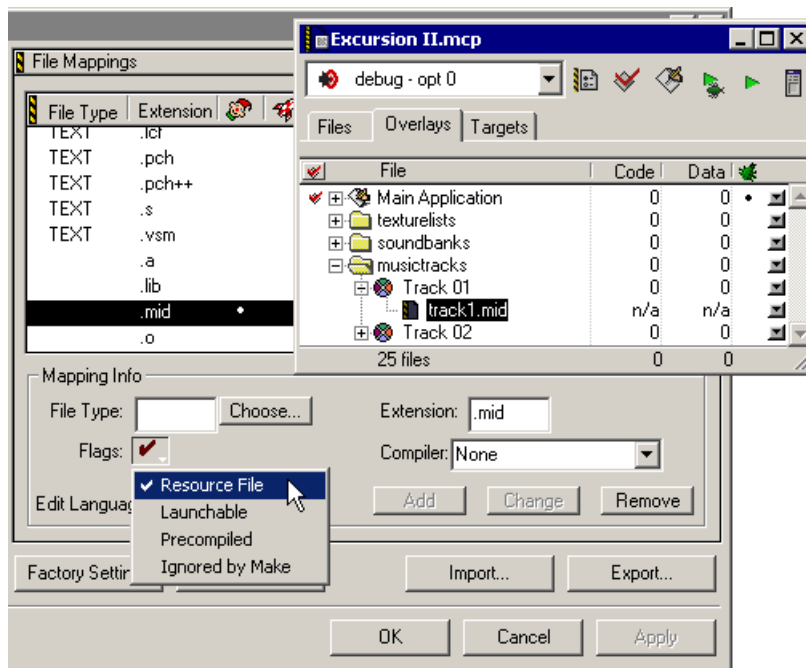
The binary file must be included in your project, and the file's extension must be typed as a resource file in the **File Mappings** target settings panel ([Figure 9.1](#)). For more help with resource files, please refer to the *IDE User Guide*.

NOTE If using the command-line tools, your binary data files must end with the extension `.bin`. This is the only filename extension that the command-line tools recognize as a resource file. The command-line tools do not let you define custom resource file types.

Linker Command File Specification

Linker Command File Syntax

Figure 9.1 Defining a Binary File Type as a Resource



To access the binary data, declare LCF symbols to mark the beginning and end of the binary data ([Listing 9.10](#)). Refer to these symbols in your code to access the data ([Listing 9.11](#)).

Listing 9.10 Embedding Binary Data into the LCF from a File

```
_musicStart = .;  
INCLUDE track01.mid  
_musicEnd = .;  
} > DATA
```

Listing 9.11 Accessing Embedded Binary Data from Your Code

```
void my_function()
{
    extern unsigned char * _musicStart[];
    extern unsigned char * _musicEnd[];
    unsigned char *ptr;

    for (ptr = _musicStart; ptr <= _musicEnd; ptr++)
    {
        do_something_with(ptr);
    }
}
```

Alphabetical Keyword Listing

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

Table 9.2 Linker Command File Keywords

.(location counter)	ADDR
ALIGN	ALIGNALL
EXCEPTION	FORCE_ACTIVE
GROUP	INCLUDE
KEEP_SECTION	LITERAL
MEMORY	OBJECT
OVERLAYID	REF_INCLUDE
SECTIONS	SIZEOF
SIZEOF_ROM	WRITEB
WRITEH	WRITES
WRITEW	

Linker Command File Specification

Alphabetical Keyword Listing

. (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 cannot 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 9.12 Moving the location counter

```
..data :
{
    *. (data)
    *. (bss)
    *. (COMMON)
    __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 9.13](#)).

Listing 9.13 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, as shown in [Listing 9.14](#).

Linker Command File Specification

Alphabetical Keyword Listing

Listing 9.14 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 following two symbols are known to the runtime system:

```
__exception_table_start__
__exception_table_end__
```

Listing 9.15 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. You may place the binary file anywhere in memory except for executable sections.

Prototype

```
INCLUDE filename
```

LCF Example

```
_binStart = .;  
INCLUDE myfile.bin  
_binEnd = .;
```

Code Example

```
void my_function()  
{  
    extern unsigned char * _binStart;  
    extern unsigned char * _binEnd;  
    unsigned char *ptr;  
  
    for (ptr = _binStart; ptr <= _binEnd; ptr++)  
    {  
        do_something_with(ptr);  
    }  
}
```

Linker Command File Specification

Alphabetical Keyword Listing

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] }
```

LITERAL

The `LITERAL` command is used for float literal merging. When you put this command in the linker command file, the linker places the `.lit4` and `.lit8` sections into the output file. These sections contain 4 byte and 8 byte floats, and the linker merges them so that you do not have duplicate entries.

The `LITERAL` sections are part of the `.sdata` segment and must be merged with an existing `* (.sdata)` statement. Do this by placing the `LITERAL` command either immediately before or after the `* (.sdata)` in your linker command file.

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 data.

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 (if supported)

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.

Linker Command File Specification

Alphabetical Keyword Listing

> **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.

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.”](#) [“Arithmetic Operations.”](#) and [“.\(location counter\).”](#)
- Describe the placement of an output section, including which input sections are placed into it. See [“File Selection.”](#) [“Function Selection.”](#) and [“Alignment.”](#)

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:

Linker Command File Specification

Alphabetical Keyword Listing

Listing 9.16 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)`

SIZEOF_ROM

The `SIZEOF_ROM` function returns the size that a segment occupies in ROM.

Prototype

`SIZEOF_ROM (segmentName)`

Until the ROM has actually been built, `SIZEOF_ROM` will always return a value of 0. For this reason, `SIZEOF_ROM` should only be called within an expression that appears inside `WRITEB`, `WRITEH`, `WRITEW`, or `AT`.

Moreover, `SIZEOF_ROM` is only needed when the `COMPRESS` option is used on the memory segment. If the memory segment has not been compressed, there is no difference between the return values of `SIZEOF_ROM` and `SIZEOF`.

For additional information, read the following topics:

- [“SECTIONS”](#)

- [“SIZEOF”](#)
 - [“WRITEB”](#)
 - [“WRITEH”](#)
 - [“WRITES”](#)
-

WRITEB

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

Prototype

`WRITEB (expression) ;`

Where *expression* returns a value 0x00 to 0xFF.

WRITEH

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

Prototype

`WRITEH (expression) ;`

Where *expression* returns a value 0x0000 to 0xFFFF.

WRITES

WRITES inserts a string at the current address of a section.

Prototype

`WRITES ("string")`

Where the length of *string* does not exceed 255 characters.

Note

There are two variables that can be used within a WRITES command:

- DATE returns the current date. With the month in alphabetic, it follows the format MMM DD YYYY.
 - TIME returns the current time. It follows the 24 hour format as HH:MM:SS.
-

Linker Command File Specification

Alphabetical Keyword Listing

```
.comment_section :  
{  
WRITES("This is a .comment section")  
WRITES("Today is " DATE " and ")  
WRITES("the time is " TIME ".")  
} > .comment
```

WRITEW

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

Prototype

WRITEW (*expression*) ;

Where *expression* returns a value 0x00000000 to 0xFFFFFFFF.

Inline Assembly and Intrinsic Functions

This chapter describes the syntax of the inline assembly language and intrinsic functions supported by the CodeWarrior compiler.

This chapter contains these sections:

- [Inline Assembly](#)
- [Intrinsic Functions](#)
- [Optimization Control Directives](#)

Inline Assembly

This section explains how to use the built-in support for assembly language programming included in the CodeWarrior compiler.

This section contains these topics:

- [Inline Assembly Syntax](#)
- [Labels](#)
- [Comments](#)
- [Preprocessor Comments and Macros](#)
- [Stack Frame](#)
- [Specifying Operands](#)
- [Using Local Variables and Arguments](#)

Inline Assembly Syntax

The inline assembler supports all the standard ARM assembly instructions and macros.

Keep these points in mind as you write assembly functions:

- All statements must either be a label, such as
 [LocalLabel:]
or an instruction, such as
 (*(instruction | directive) [operands]*)
- Each statement must end with a newline or a semi-colon.
- Hex constants must be in C style. For example:
 mov r0, 0xABCDEF // ok
- Assembler directives, instructions, and registers are not case-sensitive
- The default behavior of the compiler is to not optimize any assembly language functions. However, you may override this by using inline assembly directives. See [“Optimization Control Directives”](#).
- Every inline assembly function must end with a `bx lr` or `mov pc, lr` instruction or similar that loads the `pc` with the return address. The compiler will not do this for you. For example:

```
asm void f(void)
{
    add r2,r3,r4
    bx lr
}
```

[Listing 10.1](#) shows an example of an assembly language function.

Listing 10.1 An example assembly language function

```
asm int mystrcpy(char *to, char *from)
{
    mov r3,#0
cont:
    add r3,r3,#1      // count num chars copied
    ldrb r2,[r1],#1   // load from
    strb r2,[r0],#1   // store to
    cmp r2,#0         // check for null terminator
    bne cont          // continue if not null
    mov r0,r3         // return num chars copied
    mov pc, lr
}
```

Function-Level Inline Assembly Syntax

You can write code to specify that all the statements in the function are in assembly language using the `asm` or `__asm` qualifiers (the compiler always recognizes the `__asm` keyword, even if you check the **ANSI Keywords Only** checkbox). You may also specify local variables within the function. [Listing 10.2](#) shows an example:

Listing 10.2 Function-level Inline Assembly Example

```
asm long MyFunc(void)    // an assembly function
{
    /* Local variable definitions */
    /* Assembly language instructions */
    mov r0, #0x1
    mov r2, #0x3
    mov pc, lr
}
```

Statement-level Inline Assembly Syntax

You may also place assembly statement blocks within a function. This method uses the `asm` qualifier as a statement to combine assembly language statements and regular C/C++ statements within one function definition.

The compiler accepts three valid syntax formats for writing statement-level assembly code:

- Bracket-delimited statements ([Listing 10.3](#)) ([Listing 10.4](#))
- Parenthesis-delimited statements ([Listing 10.5](#)) ([Listing 10.6](#))

Multiple instructions must be separated by semi-colons. You do not need to terminate parenthesis-delimited inline assembly statements with semi-colons.

- Parenthesis and quote-delimited statements ([Listing 10.7](#)) ([Listing 10.8](#))

Multiple instructions must be separated by semi-colons. Multi-line statements must be continued with a backslash.

Inline Assembly and Intrinsic Functions

Inline Assembly

Listing 10.3 Bracket-delimited single-line assembly instruction example

```
long MyFunc(void)
{
    /* Local variable definitions and C/C++ statements */
    /* Statement-level Assembly language instructions */
    asm {mov r0, #0x1; mov r2, #0x3;}
    /* More C/C++ and asm {} statements */
}
```

Listing 10.4 Bracket-delimited multiple inline assembly instructions example

```
long MyFunc(void)
{
    /* Local variable definitions and C/C++ statements */
    /* Statement-level Assembly language instructions */
    asm {
        mov r0, #0x1;
        mov r2, #0x3;
    }
    /* More C/C++ and asm {} statements */
}
```

Listing 10.5 Parenthesis-delimited single-line assembly instruction example

```
long MyFunc(void)
{
    asm (mov r0, #0x1; mov r2, #0x3)
    asm (mov r0, #0x1; mov r2, #0x3);           // semi-colon optional
}
```

Listing 10.6 Parenthesis-delimited multiple inline assembly instructions example

```
long MyFunc(void)
{
    asm (
        mov r0, #0x1;
        //c++ style comment
        /* c style comment */
        mov r2, #0x3;
    )
}
```

Listing 10.7 Parenthesis and quote-delimited single-line assembly instruction example

```
long MyFunc(void)
{
    asm ("mov r0, #0x1; mov r2, #0x3;")
}
```

Listing 10.8 Parenthesis and quote-delimited multiple inline assembly instructions example

```
long MyFunc(void)
{
    asm ("mov r0, #0x1; \
        mov r2, #0x3") // backslash required to split quoted statements
}
```

Labels

A label can be any identifier that you have not already declared as a local variable. The label may start with @, so these are valid examples: red, @red, and @1. Only those labels that do not start with @ need to end in a colon ([Listing 10.9](#)).

Listing 10.9 Creating Labels

```
asm void red(void)

{
x1: add r3,r4,r5
@x2 add r6,r7,r8
}
```

Comments

You can use C and C++ comments within inline assembly. However, you cannot use comments with a pound sign (#) or semicolon (;) because the preprocessor uses those symbols.

Preprocessor Comments and Macros

You can use all preprocessor features, such as comments and macros, in the assembler. In a multi-statement, multi-line macro, you must end each assembly statement with a

Inline Assembly and Intrinsic Functions

Inline Assembly

semicolon (;) because the (\) operator removes the newlines that would usually separate each assembly instruction. [Listing 10.10](#) shows some macros and comments in use.

Listing 10.10 Using the Preprocessor

```
#define BITSET(reg, bit)\
    asm (orr reg,reg,#1 << bit)
#define BITCLR(reg, bit)\
    asm (bic reg,reg,#1 << bit)

#define MULTILINE\
    asm {                      /*multi-line macro*/\
        mov r0, #0x1;\
        mov r2, #0x3;\
    }

void NitroMain ()
{
    BITSET (r0,3)
    BITCLR (r0,3)
    MULTILINE
}
```

Stack Frame

You must create a stack frame for a function if the function

- calls other functions
- declares non-register arguments or local variables.

To create a stack frame, use the following code to generate the necessary prologues and epilogues to preserve the function's register status. ([Listing 10.11](#)).

Listing 10.11 Creating a Stack Frame

```
asm void red()
{
    // save return address on stack
    #ifndef __thumb
        str lr,[sp,#-4]!
    #else
        push {lr}
    #endif

    // Your code here

    // restore return address from stack and return
    #ifndef __thumb
        ldr pc,[sp],#4;
    #else
        pop {pc}
    #endif
} // Your code here
}
```

Specifying Operands

This section explains how to specify operands for assembly language instructions:

- [Using Register Variables and Memory Variables](#)
- [Using Registers](#)
- [Using Labels](#)
- [Using Variable Names as Memory Locations](#)
- [Using Immediate Operands](#)

Using Register Variables and Memory Variables

Variables in the ARM architecture must be accessed through register operations. C/C++ variables and arguments that live in registers may be directly accessed by instructions that operate on register operands ([Listing 10.12](#)).

Inline Assembly and Intrinsic Functions

Inline Assembly

Listing 10.12 Directly accessing register variables

```
asm int addint (register int a, register int b)
{
    add r2,a,b;
    mov r0,r2;
    mov pc,lr;
}
```

Variables that reside in system memory (global variables, local variables, and arguments on the stack) may be accessed indirectly, by loading the addresses of the variables in a register using the `lda` pseudo-instruction ([Listing 10.13](#)). Aggregate objects such as structures also reside in memory, where they cannot be fully represented in a register.

The address of a local struct can be loaded to an address register and initialized via a store multiple instruction. Since the structure resides in memory, it is necessary to load the address of the structure and to store the values indirectly to memory. For purposes of this example, we explicitly loaded the first element `x` of the struct. In practice, the bare struct variable would suffice.

Listing 10.13 Accessing a structure element

```
struct xy
{
    int x; int y;
};

int main ()
{
    struct xy lxy;
    __asm {
        lda r2,lxy.x      // Base address of struct (lxy would also work)
        stmia r2,{r0,r1}
    }
}
```

If a pointer variable resides in a register, the variable can be used to reference memory via the `[]` assembly syntax. Normally, the compiler would try to place as many variables into memory as possible. If this is not possible, you can use the `register` keyword to tell the compiler to favor this variable for a register location ([Listing 10.14](#)).

For the entire duration that a C/C++ variable is in a register, it is treated as a symbolic name for the compiler assigned register number. Use the `lda` pseudo-instruction to load variables that live in memory into a register.

Listing 10.14 Using pointer variables

```
int arr[10];

int main()
{
    register int *pint = arr;
    __asm {
        ldr r0,[pint] // Load element 0 of array
    }
}
```

[Listing 10.15](#) demonstrates additional addressing modes using symbolic register names. It copies the elements of the from array to the to array, beginning at element start_index and ending at stop_index.

Listing 10.15 Using C variables in inline assembly

```
asm void copy_array_slice(register int *from,
                          register int *to,
                          register int start_index,
                          register int stop_index)
{
    add stop_index,stop_index,#1

loop:
    ldr r5,[from, start_index, LSL #2]
    str r5,[to, start_index, LSL #2]
    add start_index,start_index,#1
    cmp start_index,stop_index
    bne loop
    mov pc,lr
}

int main()
{
    int from_arr[] = {
        1,2,3,4,5,6,7,8,9
    };
    int to_arr[4];
    copy_array_slice(from_arr,to_arr,4,7);
}
```

Using Registers

Several registers are accessible using inline assembly. Register names are not case-sensitive.

User mode registers:

- R0 – R14 (the visible general purpose registers)
- SP (Stack Pointer, normally R13)
- LR (Link Register, also accessible as R14)
- PC (Program Counter, also accessible as R15)

You can access the CPSR register (Current Program Status Register) by using the MSR instruction:

```
MSR CPSR_<fields>, #imm
```

```
MSR CPSR_<fields>, <Rm>
```

Using Labels

For a label operand, you can use the name of a label as the target of a branch instruction. You can also use function names as branch targets.

- `b @3` — correct syntax for branching to a local label
- `b red` — correct syntax for branching to external function `red`
- `bl red` — correct syntax for calling external function `red`
- `bne red` — incorrect syntax; short branch outside function `red`

NOTE Local labels (labels defined within a function) may be redefined in different functions.

Using Variable Names as Memory Locations

If an instruction requires a memory location, you can use a local or global variable name. You can modify local variable names with struct member references, class member references, array subscripts, or constant displacements.

You can also use a register variable that is a pointer to a struct or class to access a member of the struct.

Using Immediate Operands

For an immediate operand, you can use an integer or enum constant, `sizeof` expression, and any constant expression using any of the C dyadic and monadic arithmetic operators.

These expressions follow the same precedence and associativity rules as normal C expressions. The inline assembler carries out all arithmetic with 32-bit signed integers.

An immediate operand can also be a reference to a member of a struct or class type. You can use any struct or class name from a `typedef` statement, followed by any number of member references. This evaluates to the offset of the member from the start of the struct. For example:

```
add    r2,r2,#txy.y
```

You also can use the top or bottom half-word of an immediate word value as an immediate operand by using one of the @ modifiers.

Using Local Variables and Arguments

The inline assembler attempts to place as many variables as it can into registers. Non-scalar variables such as structures are stored in static memory if they are global, or on the stack if they are local. If a variable that cannot be put in a register is used in a register context, the inline assembler will generate an error.

For example, compiling the code in [Listing 10.16](#) results in the error: *'e' could not be assigned to a register*. This is because the function received more than four register arguments and the variable 'e' is being passed on the stack.

Listing 10.16 Invalid variable used in a register context generates an error

```
asm void manyargs(int a, int b, int c, int d, int e)
{
    struct xy {
        int x;
        int y;
    } lxy;

    lda r0,lxy
    str e,[r0] //store e to lxy.x
}
```

To correct the error, the address of 'e' must be loaded to a register ([Listing 10.17](#)).

Inline Assembly and Intrinsic Functions

Inline Assembly

Listing 10.17 Correct the error by loading the address of 'e' to a register

```
asm void manyargs(int a, int b, int c, int d, int e)
{
    struct xy {
        int x;
        int y;
    } lxy;

    lda r0,lxy
    lda r1,e
    ldr r2,[r1] //load e
    str r2,[r0] //store e to lxy.x
}
```

Intrinsic Functions

Intrinsic functions are functions defined within the compiler. Intrinsic functions are not part of the ANSI C or C++ standards. They are an extension that the CodeWarrior compilers provide. Using intrinsic functions, you do not need to choose the actual registers that an instruction needs. The intrinsic functions internally allocate values to registers and return the result. The result type that the function returns depends on the instruction. You can assign the result to a variable with a matching data type.

When the compiler encounters the intrinsic function call in your source code, it does not actually make a function call. The compiler substitutes the assembly instruction that matches your function call. As a result, no function call occurs in the final object code. The final code has the assembly language instructions that correspond to the intrinsic functions.

NOTE You can use intrinsic functions or the `asm` keyword to add a few lines of assembly code within a function. If you want to write an entire function in assembly, use inline assembly.

Additional intrinsic functions are available for:

- [Buffer Manipulation](#)
- [Pseudo-Instructions](#)

Buffer Manipulation

Some intrinsic functions allow control over areas of memory, so you can manipulate memory blocks

- `__alloca` implements `alloca()` in the compiler.
`void *__alloca(unsigned int);`
`__alloca` may not be used in functions that also implement C99 variable length arrays (VLAs). VLAs use a form of `__alloca` that is incompatible with the user form of `__alloca`.
- `__memcpy()` provides access to the block move in the code generator to do the block move inline.
`void *__memcpy(void *, void *, unsigned long);`

Pseudo-Instructions

Pseudo-Instructions are intrinsic functions that expand into multiple assembly instructions. They provide shortcuts for common operations that would normally require you to enter multiple instructions to implement them.

Inline Assembly and Intrinsic Functions

Intrinsic Functions

- [DCD](#)
 - [LDA](#)
 - [LDCONST](#)
-

DCD

Defines constant data

Prototype

`DCD x;`

Remarks

Provides a way to define 32-bit values. Emits data 'x' into the stream. 'x' may be any legal constant

`dcd 0xa; emit 10 in hexadecimal`

`dcd 10; emit 10 in decimal`

`dcd 0b1010; emit 10 in binary`

`dcd 010; emit 10 in octal`

LDA

Load 32-bit address into register

Prototype

`LDA r, <address constant>`

Remarks

Expands roughly to the code:

`ldr r, [pc, <offset>]`

`dcd <32bit value>`

LDCONST

Load constant immediate value

Prototype

LDCONST r, #<value>

Remarks

Similar to LDA, but can load a large immediate value.

Optimization Control Directives

The compiler does not optimize inline assembly code unless you specifically use these directives to control optimization.

- `.nonvolatile` - indicates assembly code that may be optimized.
- `.volatile` - indicates assembly code that must not be optimized

These directives may be used within statement-level `asm{ }` ([Listing 10.18](#)) and `asm()` ([Listing 10.19](#)) blocks. They may not be used within function-level inline assembly.

Listing 10.18 Usage of optimization directives within an `asm{ }` block.

```
int func(void) {
asm {
    .volatile
    mov r0,#1
    bl label1
    mov r0,#2    // this instruction remains
    b label2     // this instruction remains
label1:
    mov r0,#3
    mov r0,#4
label2:
    mov r0,#5
    .nonvolatile
label_A:
    mov r0,#1
    bl label3
    mov r0,#2    // will be removed
    b label4     // will be removed
label3:
    mov r0,#3
    mov r0,#4
label4:
    mov r0,#5
    }
}
```

Inline Assembly and Intrinsic Functions

Optimization Control Directives

Listing 10.19 Usage of optimization directives within `asm()` blocks.

```
asm(.nonvolatile; move R2,#1; bl lr; move R1,#2)
```

The compiler initializes the optimization flag once per inline assembly block, so the effect of an optimization directive does not carry over to subsequent inline assembly blocks. In other words, each block of inline assembly code, by default, is not optimized, until you use a `.nonvolatile` directive.

Overlays

Overlays are segments of code or data that share the same memory space. By loading and unloading different segments, you can create programs with code and graphics that would not normally fit into memory, without using a virtual memory manager.

- [Defining Overlay Structures](#)
- [Overlay Management](#)
- [Determining Valid Intersegment Calls](#)

Defining Overlay Structures

The structure of the overlays in your program are defined in the linker command file. You can use the Overlay tab in the project window to arrange the overlay structure, and the LCF Prelinker to automatically create the appropriate linker command file.

A working overlay sample project is located in the CodeWarrior\Examples folder.

- [Creating Overlay Groups in the Project Window](#)
- [Creating the Linker Command File with the Prelinker](#)
- [Creating Overlay Structures Using the Command Line Linker](#)

Creating Overlay Groups in the Project Window

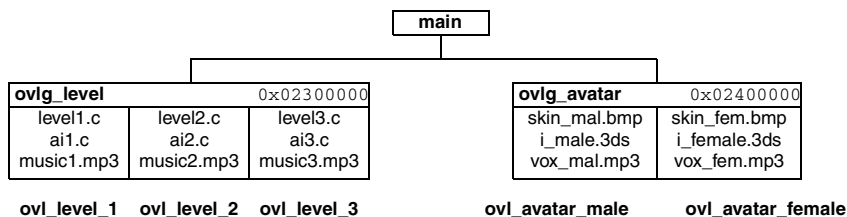
The CodeWarrior IDE lets you create an overlay tree only one-level deep ([Figure 11.1](#)). (This restriction means that you cannot create overlays within other overlays.)

You create this organization of overlay containers in the Overlays tab of the project window ([Figure 11.2](#)).

Overlays

Defining Overlay Structures

Figure 11.1 Sample OverlayTree



The IDE organizes overlays in a hierarchical fashion using two containers:

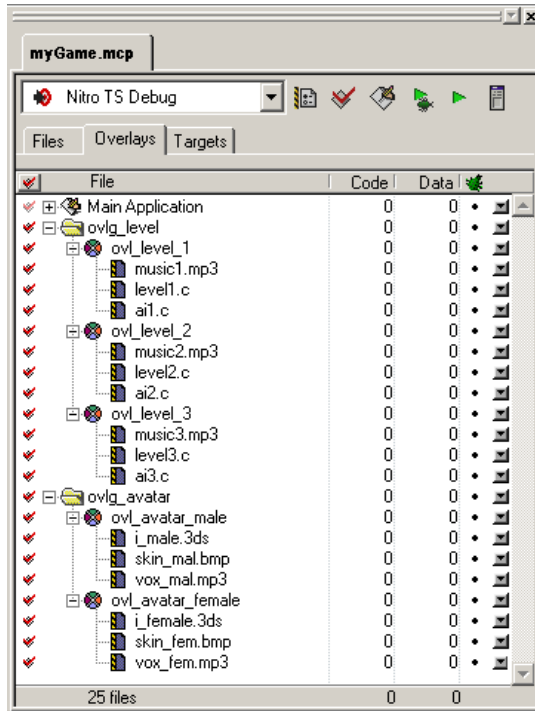
- an *overlay group* - a block of memory shared by two or more overlays. The **Project > Create Overlay Group** command creates an overlay group container.

To specify the memory (base code) address for the overlay group, double-click the overlay group in the project window. This opens the **Overlay Group Info** dialog box. Enter the base code address here. To specify that an overlay group should start at the end of the previous segment instead of a hardcoded address, enter FF.

- an *overlay* - a block of code/ data consisting of one or more source files. The **Project > Create New Overlay** command creates an overlay container within the selected overlay group. Double-click the overlay to rename it.

You add source and data files to the overlays by dragging them out of the **Main Application** overlay and into the overlay you want. New files can be added as usual, with the **Project > Add Files** command. Note that resource filetypes such as .mp3 and .jpg must first be registered with the **Binary Converter** in the **Target Settings > File Mappings** panel. See [“Binary Converter/Compiler”](#).

Figure 11.2 Overlays Tab of the Project Window



Creating the Linker Command File with the Prelinker

After you have set up the overlays in the project window, you **must** activate the Nitro LCF Generator Prelinker in the **Target** settings panel to create the appropriate linker command file. For details about this panel, see [“NITRO LCF Prelinker Panel”](#).

For the example given in [Figure 11.1](#) and [Figure 11.2](#), where we have hardcoded the memory addresses of the overlays, the prelinker would create the following sections in the LCF:

```
ovlg_level_1      (RWXO): ORIGIN = 0x02300000, LENGTH = 0x0 > ovl_level_1.sbin
ovlg_level_2      (RWXO): ORIGIN = 0x02300000, LENGTH = 0x0 > ovl_level_2.sbin
ovlg_level_3      (RWXO): ORIGIN = 0x02300000, LENGTH = 0x0 > ovl_level_3.sbin
ovlg_avatar_male  (RWXO): ORIGIN = 0x02400000, LENGTH = 0x0 > ovl_avatar_male.sbin
ovlg_avatar_female (RWXO): ORIGIN = 0x02400000, LENGTH = 0x0 > ovl_avatar_female.sbin
```

Overlays

Overlay Management

If we instead had used FF as the base code address and did not care about the exact position of the overlays in memory, the prelinker would have inserted the following into the LCF:

```
ovl_level_1      (RWXO): ORIGIN = AFTER(main), LENGTH = 0x0 > ovl_level_1.sbin
ovl_level_2      (RWXO): ORIGIN = AFTER(main), LENGTH = 0x0 > ovl_level_2.sbin
ovl_level_3      (RWXO): ORIGIN = AFTER(main), LENGTH = 0x0 > ovl_level_3.sbin
ovl_avatar_male  (RWXO): ORIGIN = AFTER(ovl_level_3,ovl_level_2,ovl_level_1),
                      LENGTH = 0x0 > ovl_avatar_male.sbin
ovl_avatar_female (RWXO): ORIGIN = AFTER(ovl_level_3,ovl_level_2,ovl_level_1),
                      LENGTH = 0x0 > ovl_avatar_female.sbin
```

Creating Overlay Structures Using the Command Line Linker

If using the command line tools, you can use command-line options to define the overlay structures.

Unless you specify otherwise, all files are assigned to the Main Application. You can define new overlay groups with the `-overlaygroup` option. Define individual overlays with the `-overlay` option; files that follow the overlay definition are placed inside that overlay.

Table 11.1 Command-line Options for Overlays

Option	Parameter	Explanation
<code>-og</code> <code>-overlaygroup</code>	name, lo-addr, [hi-addr]	Specifies new overlay group Maximum name length is 255 characters. Start address Optional end address. Default is 0 (unlimited length)
<code>-ol</code> <code>-overlay</code>	name	Specify new overlay within overlay group Maximum name length is 255 characters.

Overlay Management

To load and unload overlays, use the overlay loading functions `FS_LoadOverlay()` and `FS_UnloadOverlay()`. These functions are provided by the Nintendo DS SDK, and are described in the SDK documentation.

Determining Valid Intersegment Calls

There are restrictions regarding how you can call symbols between overlay segments and between overlay segments and the main application. The linker displays error messages if the linker detects ambiguous symbol references in your code.

- [Symbol References](#)
- [Duplicated Symbol Definitions](#)
- [Extern Calls to Duplicated Symbol Definitions](#)

Symbol References

Symbols in one overlay segment may be referenced by another segment only if there is a valid call path to the symbol.

Figure 11.3 Example 1: Valid Symbol References

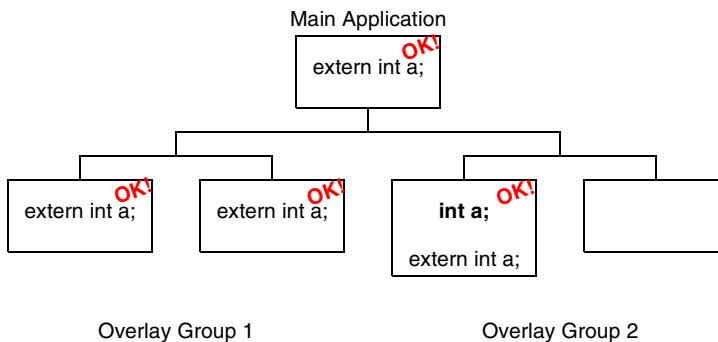
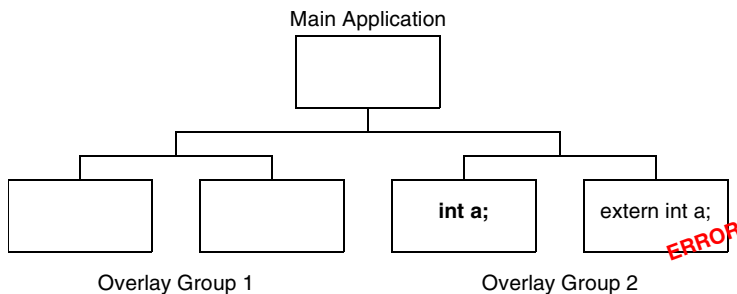


Figure 11.4 Example 2: An invalid Symbol Reference



Overlays

Determining Valid Intersegment Calls

Duplicated Symbol Definitions

A symbol may be defined multiple times if only a single copy can exist in memory at any time.

Figure 11.5 Example 3: Valid and Invalid Duplicate Symbols

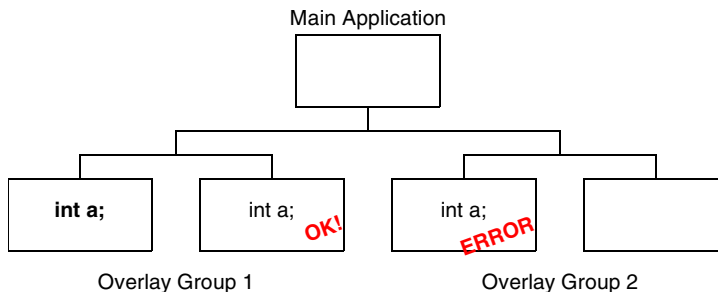
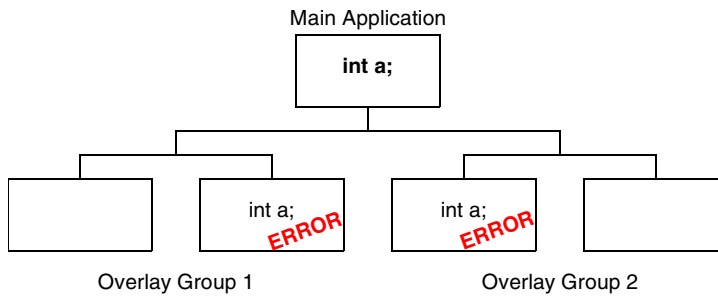


Figure 11.6 Example 4: Invalid Duplicate Symbols



Extern Calls to Duplicated Symbol Definitions

A segment may not define an extern symbol if the symbol has duplicate definitions.

Figure 11.7 Example 5: Invalid Extern

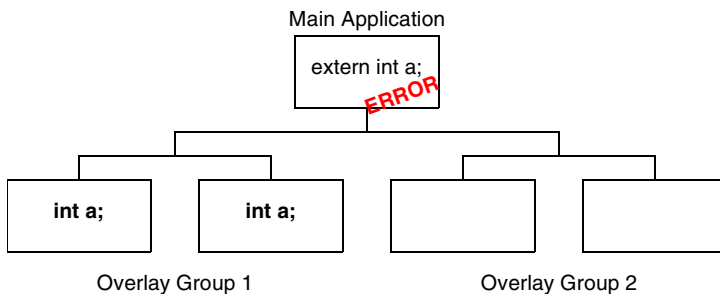
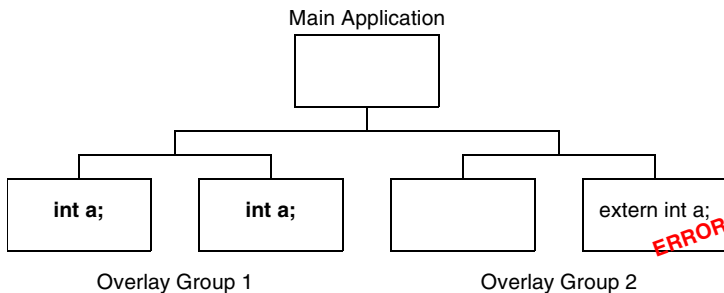


Figure 11.8 Example 6: Invalid Extern



Overlays

Determining Valid Intersegment Calls

Libraries

We provide several libraries for you to use within the Nintendo DS programming environment:

- [Runtime Libraries](#)
- [Mandatory Floating Point Libraries](#)
- [MSL C/C++](#)

Runtime Libraries

The Nintendo DS runtime libraries are located in the folder:

```
{CodeWarrior}\ARM_EABI_Support\Runtime\Lib
```

The runtime libraries conform to the naming convention:

```
NITRO_Runtime_Mode_ByteOrder(_STRB).a
```

where:

- *Mode* identifies the ARM mode
 - A — ARM mode
 - Ai — ARM mode with internetworking
 - T — thumb mode
- *ByteOrder*
 - LE — little endian
- *STRB* — libraries built without STRB instructions

Mandatory Floating Point Libraries

You must include a floating point library in each of your projects. These libraries contain more than floating-point operations. The libraries also contain integer division and 64-bit integer support code. If your application does not use any floating point operations, the linker deadstrips the floating-point support code from the output file and keeps only the integer support code.

Libraries

Mandatory Floating Point Libraries

The floating point libraries are located in the folder:

```
{CodeeWarrior}\ARM_EABI_Support\Mathlib\lib
```

The floating-point library names conform to the naming convention:

```
FP_Compliance_Architecture_ByteOrder.a
```

where:

- *Compliance* identifies the IEEE floating-point standard:
 - `flush0`—flush to zero (the default library). This mode is not compatible with the IEEE standard. The library behaves like the VFP coprocessor in RunFast mode. Finite values are accurate, with rounding-to-nearest and no exceptions. Subnormals flush to zero on input and output. The library does not produce NaNs and infinities in all circumstances that the IEEE model defines. Also, the library does not guarantee the sign of zero that the IEEE model defines.
 - `fastI`—fast IEEE (the default mode that the IEEE standard specifies). Results conform to the IEEE specification, with rounding-to-nearest and no exceptions. This mode is as fast as `flush0` on finite inputs, and the library size is up to 20% larger than `flush0`.
 - `fixedI`—fixed-rounding IEEE. Overflow, underflow, division-by-zero and invalid-operation exception generation conforms to the IEEE standard. This mode supports only rounding-to-nearest. This mode is up to 20% slower than `flush0`, and the library size is up to 65% larger than `flush0`.
 - `fullI`—full IEEE. All facilities, operations, and representations guaranteed by the IEEE standard are available in single and double precision. You can select the rounding mode dynamically at run time. This mode is up to 100% slower than `flush0`, and the library size is up to 100% larger than `flush0`.
- *Architecture* identifies the processor architecture:
 - `xM`—ARM v4 xM with no Thumb interworking. These libraries are slow, but compatible with all ARM7 cores or higher.
 - `v4t`—ARM v4 with Thumb interworking (the default architecture). These libraries require the ARM7TDMI or higher cores.
 - `v5t`—ARM v5 with Thumb interworking. These optimized libraries require ARM9E, ARM10, XScale, or higher cores.
- *ByteOrder* identifies the byte order: big endian (BE) or little endian (LE).

MSL C/C++

The Metrowerks Standard Libraries are ANSI C Standard Libraries.

The MSL C libraries are located in the folder:

```
{CodeWarrior}\ARM_EABI_Support\msl\MSL_C\MSL_ARM\Lib
```

The MSL C++ libraries are located in the folder:

```
{CodeWarrior}\ARM_EABI_Support\msl\MSL_C++\MSL_ARM\Lib
```

The library names conform to the naming convention:

```
MSL_C (PP) _NITRO__Mode_ByteOrder(_STRB).a
```

where:

- *Mode* identifies the ARM mode
 - A — ARM mode
 - Ai — ARM mode with internetworking
 - T — thumb mode
- *ByteOrder*
 - LE — little endian
- *STRB* — libraries built without STRB instructions

MSL Extras

The MSL Extras libraries contain functions, macros, and types that are not included in the ANSI/ISO C standard, and POSIX functions. The libraries are located in the folder:

```
{CodeWarrior}\ARM_EABI_Support\msl\MSL_Extras\MSL_ARM\Lib
```

The library names conform to the naming convention:

```
MSL_Extras_NITRO__Mode_ByteOrder(_STRB).a
```

where:

- *Mode* identifies the ARM mode
 - A — ARM mode
 - Ai — ARM mode with internetworking
 - T — thumb mode
- *ByteOrder*
 - LE — little endian
- *STRB* — libraries built without STRB instructions

Command-Line Tools

This appendix chapter describes the command-line tools

- [bintoelf.exe](#)
- [elftobin.exe](#)
- [mwasmmarm.exe](#)
- [mwccarm.exe](#)
- [mwldarm.exe](#)

bintoelf.exe

Converts binary files to ELF files that can be linked directly into your project. See [“Binary Converter/Compiler”](#) for more information.

```
BinToELF filename.xxx [-symbolname name] [-aligndata n]
    [-section sectionname] [-endian (little|big)]
    [-output filename.o]
```

`-symbolname name`

Assign symbol name. The symbolname may contain special formatting characters:

`%f` - maps to the passed filename passed.

For example: `BinToElf SomeFile.jpg -symbolname Prefix%f` generates a symbol `PrefixSomeFile.jpg`. Note that the periods are replaced with underscores.

`-aligndata n`

Set data alignment.

`-section sectionname`

Set section name.

`-endian (little|big)`

Set data endian mode.

`-output filename.o`

Set output filename.

Command-Line Tools

elftobin.exe

```
-defaultsettings file.xxx
```

Use default settings from file.

elftobin.exe

The elftobin utility converts an ELF file into a ROM image binary, using the `v_addr` field from the ELF file program header.

It can also combine binary images from two ELF files using a specified header binary file. The header information of the combined file is created as follows :

0x20 - 0x23	:	main ELF start address
0x24 - 0x27	:	main ELF end address
0x28 - 0x2b	:	main ELF load address
0x2c - 0x2f	:	main ELF load size
0x30 - 0x33	:	sub ELF start address
0x34 - 0x37	:	sub ELF end address
0x38 - 0x3b	:	sub ELF load address
0x3c - 0x3f	:	sub ELF load size

The binary image and header file are aligned by 512 bytes.

TIP The `elftobin.exe` utility is provided for compatibility with early versions of the Nitro SDK. The functions in `elftobin.exe` are now provided by the Nitro SDK utility, `makerom.exe`

```
elftobin <elf file1> [<elf file2> <header file>]  
[-o <file>] [-v]
```

`elf file1`

Main ELF file name.

`elf file2`

sub ELF file name. If not specified, convert ELF to Binary without header info.

`header file`

NITRO header file (binary). Specify only if a second elf file `<elf file2>` is also specified.

`-o file`

output file name. If not specified, output file name is generated from `elf file1`.

`-v`

show progress message of each sections.

mwasmarm.exe

Mwasmarm.exe is the command line assembler.

Table A.1 Assemblers-specific command-line options

Option	Parameter	Explanation
-[no]case		Make identifiers case-sensitive
-[no]colons		Require labels to be followed by colons
-[no]gnu		enable GNU compatibility
-[no]ads		enable ARM ADS compatibility
-list		create a listing file
-[no]period		global; require directives to be preceeded with a period “.”
-proc[essor] keyword	arm7 arm9	specify processor
-16		assembler 16bit thumb instructions
-32		assemble 32bit ARM instructions
-fpu keyword	none fpa vfpv1 vfpv2	specify target fpu architecture
-[no]space		allow spaces in the operand field

mwccarm.exe

Mwccarm.exe is the command line compiler.

Compiler CodeGen Options

[Table A.2](#) shows the compiler code-generation command-line options.

NOTE Enter this command line for additional help regarding compiler command-line options: `mwccarm -help`

Table A.2 Compiler code-generation command-line options

Option	Parameter	Explanation
-[no]ashla		Enable ARM Shared Library Architecture support THIS OPTION IS NOT USED FOR NINTENDO DS DEVELOPMENT
-[no]big		Generate code and link for a big-endian target
-[no]little		Generate code and link for a little-endian target; default setting
-[no]constpool		Pool constants and disable dead-stripping
-fp <i>keyword</i>		Specify floating-point options
	soft[ware]	Software floating-point emulation; default setting
-[no]interworking		Generate ARM/Thumb interworking sequences
-[no]pic		Generate position-independent code references THIS OPTION IS NOT USED FOR NINTENDO DS DEVELOPMENT

Table A.2 Compiler code-generation command-line options (*continued*)

Option	Parameter	Explanation
-[no]pid		Generate position-independent data references THIS OPTION IS NOT USED FOR NINTENDO DS DEVELOPMENT
-proc[essor] <i>keyword</i>		Specify the target processor
	arm7	ARM7 hardware; default setting
	arm9	ARM9 hardware
-rostr -readonlystrings		~str readonly
-sdatathreshold long		Set maximum size in bytes for data objects before going into .sdata section (default is 8)
-[no]thumb		Generate Thumb instructions
-[no]profile		Generate code for profiling

mwldarm.exe

Mwldarm.exe is the command line linker. The linker options are broken into these groups:

- [Project Options](#)
- [Linker Options](#)
- [Linker CodeGen Options](#)
- [ELF Disassembler Options](#)

Project Options

[Table A.3](#) shows the project command-line options.

Table A.3 Project command-line options

Option	Parameter	Explanation
-application		Global; generate an application; default setting
-library		Global; generate a static library
-partial		Global; generate a partial link

Linker Options

[Table A.4](#) shows the linker command-line options.

NOTE Enter this command line for additional help regarding linker command-line options: `mwldarm -help`

Table A.4 Linker command-line options

Option	Parameter	Explanation
-dis[assemble]		Global; disassemble object code and do not link; implies <code>-nostdlib</code>
-L+ -l <i>path</i>		Global; cased; add library search path (default setting is to search the current working directory and then system directories (see <code>-defaults</code>); search paths have global scope over the command line and are searched in the order given
-lr <i>path</i>		Global; like <code>-l</code> , but add recursive library search path
-l+ <i>file</i>		Cased; add a library by searching access paths for file named <code>libfile.ext</code> where <i>ext</i> is a typical library extension; if that fails, try to add <i>file</i> directly; library added in link order before system libraries (see <code>-defaults</code>)

Table A.4 Linker command-line options (*continued*)

Option	Parameter	Explanation
-[no]defaults		Global; same as -[no]stdlib; default setting
-nofail		Continue importing or disassembling after errors in earlier files
-[no]stdlib		Global; use system library access paths (specified by %MWLibraries%) and add system libraries (specified by %MWLibraryFiles%) at end of link order; default setting
-S		Global; cased; disassemble and send output to a file; do not link; implies -nostdlib
-proc[essor] <i>keyword</i>		Specify the target processor
	arm7	ARM7 hardware; default setting
	arm9	ARM9 hardware
	strongarm	StrongARM hardware
	xscale	XScale™ technology
-rostr -readonlystrings		~str readonly
-sdatathreshold long		set maximum size in bytes for data objects before going into .sdata section (default is 8)
-[no]thumb		Generate Thumb instructions
-[no]profile		Generate code for profiling

Linker CodeGen Options

[Table A.5](#) shows the linker code-generation command-line options.

NOTE Enter this command line for additional help regarding linker command-line options: mwldarm -help

Table A.5 Linker code-generation command-line options

Option	Parameter	Explanation
-[no]ashla		Enable ARM Shared Library Architecture support THIS OPTION IS NOT USED FOR NINTENDO DS DEVELOPMENT
-[no]big		Generate code and link for a big-endian target
-[no]little		Generate code and link for a little-endian target; default setting
-[no]constpool		Pool constants and disable dead-stripping
-[no]interworking		Generate ARM/Thumb interworking sequences
-[no]pic		Generate position-independent code references THIS OPTION IS NOT USED FOR NINTENDO DS DEVELOPMENT
-[no]pid		Generate position-independent data references THIS OPTION IS NOT USED FOR NINTENDO DS DEVELOPMENT
-proc[essor] <i>keyword</i>		Specify the target processor
	arm7	ARM7 hardware; default setting
	arm9	ARM9 hardware
	strongarm	StrongARM hardware
	xscale	XScale™ technology

Table A.5 Linker code-generation command-line options (*continued*)

Option	Parameter	Explanation
<code>-sdatathreshold long</code>		Set maximum size in bytes for data objects before going into <code>.sdata</code> section (default is 8)
<code>-[no]thumb</code>		Generate Thumb instructions
<code>-[no]profile</code>		Generate code for profiling

ELF Disassembler Options

[Table A.6](#) shows ELF disassembler command-line options.

Table A.6 ELF disassembler command-line options

Option	Parameter	Explanation
<code>-show keyword[,...]</code>		Specify disassembly options
	<code>only none</code>	For example: <code>-show only</code> <code>-show none</code>
	<code>all</code>	Show everything; default setting

Command-Line Tools

mwldarm.exe

Table A.6 ELF disassembler command-line options (*continued*)

Option	Parameter	Explanation
	[no]code [no]text	Show disassembly of code sections; default setting
	[no]comments	Show comment field in code; implies -show code; default setting
	[no]extended	Show extended mnemonics; implies -show code; default setting
	[no]data	Show data; with -show verbose, show hex dumps of sections; default setting
	[no]debug [no]sym	Show symbolics information; default setting
	[no]exceptions	Show exception tables; implies -show data; default setting
	[no]headers	Show ELF headers; default setting
	[no]hex	Show addresses and opcodes in code disassembly; implies -show code; default setting
	[no]names	Show symbol table; default setting
	[no]relocs	Show resolved relocations in code and relocation tables; default setting

Table A.6 ELF disassembler command-line options (*continued*)

Option	Parameter	Explanation
	[no]source	show source in disassembly; implies -show code; with -show verbose, displays entire source file in output, otherwise shows only four lines around each function; default setting
	[no]xtables	Show exception tables; default setting
	[no]verbose	Show verbose information, including hex dump of program segments in applications; default setting

Command-Line Tools

mwldarm.exe

Index

Symbols

- * 126
- . (location counter) 132
- .ctor 128
- .init 128
- .mem 64
- __alloca 155
- __attribute__ 109
- __call_via_rX 107
- __exception_table_end__ 134
- __exception_table_start__ 134
- __memcpy 155
- __sinit__ 128

A

- access permission flags 120, 136
- addr 132
- after 137
- align 133
- alignall 133
- aligned 109
- alignment 109, 123
- allow_byte 104
- arguments
 - inline assembly 153
- ARM Assembler panel 33
- ARM Debugger Settings panel 62
- ARM instruction mode
 - disassembler 73
- ARM Linker panel 53
- ARM Processor panel 36
- ARM Project panel 32
- ARM Thumb Procedure Call Standard 107
- ARM7 67
- ARM7 debugging 80
- ASHLA 104
- asm
- assembler 91
 - debugging 92
 - GNU extensions 93
 - labels 91

See also inline assembly

- assignment, in LCF 125
- ATPCS 107
- __attribute__ ((aligned(?)))
 - struct definition examples 109
 - struct member examples 110
 - variable declaration examples 109
- avoid_byte 105

B

- back-end compiler *See* compiler
- backward chaining, defined 15
- BatchRunner PostLinker panel 47
- BatchRunner Preprocessor panel 48
- binary compiler plug-in 115
- binary converter 115
- Binary Converter Settings panel 42
- binary files, linking 115
- bl 107
- blx 107
- bool size 100
- build target
 - defined 14
- bx 107
- bx rX 107
- byte_access 102

C

- cache viewer 81
- __call_via_rX 107
- cartridge slot 86
- char size 100
- checking syntax 17
- CHM documentation 10
- code that compiler generates for subroutine calls 107
- CodeWarrior IDE
 - checking syntax in 17
 - compared to command line 15
 - compiling code in 16
 - debugger, defined 14

- definition of 14
- development process 15–17
- editing source code in 16
- installing 12
- introduction to tools 9
- linking in 16
- preprocessing in 17
- project manager, defined 14
- project window 16
- projects compared to Makefiles 15
- tools list 14
- uninstalling 13
- command-line and CodeWarrior IDE,
 - compared 15
- command-line tool options
 - compiler codegen options 174
 - project options 175
 - specific ELF disassembler options 179
 - specific linker codegen options 177
 - specific linker options 176
- comments
 - in inline assembly 147
- compiler
 - back-end 91, 99
 - generated code for subroutine calls 107
 - inline assembly support 143
 - other documentation 99
 - See also C Compilers Reference*
- compiling 16
- compstatic 60
- configuring
 - debugger 70
 - remote debugging 72
- context-sensitive help 10
- Create New Overlay 160
- Create Overlay Group 160
- creating new projects 19

D

- data 102
- data cache 81
- DCD pseudo-instruction 156
- dead_stripping 105
- dead-code elimination. *See* deadstripping
- deadstripping 117
 - prevention 121, 124
- debug target 27

- debugger
 - configuring 70
 - settings panel 70
- debugging 69
 - multi-core 67, 80
- decl_spec 101
- declaration specifiers 101
- define_section 105
- definition
 - of build target 14
 - of CodeWarrior IDE 14
 - of interworking 107
 - of veneer 107
- disassembler 17
 - ARM/thumb mode 73
- documentation
 - in CHM format 10
 - in PDF format 9
 - overview 9
- double size 101
- DS Flash Card 86
- DWARF, disassembly 17

E

- editing source code 16
- ELF Disassembler panel 49
- ELF files, disassembling 16
- exception 124, 134
- exception tables 124
- expressions, in LCF 125

F

- file mappings 129
- file, assembler directive 93
- flash card 86
- float size 101
- floating point libraries 167
- floating-point formats 101
- force_active 121, 124, 134
- forward chaining, defined 15
- function, assembler directive 92

G

- garbage collection. *See* deadstripping
- GBA cartridge 86
- generated code for subroutine calls 107
- generic_symbol_names 106

getTime 85
GNU assembler extensions 93
group 126, 135

H

heap size 127
help
 context-sensitive 10

I

IDE
 uninstalling 13
include 129, 135
inline assembly 143
 arguments, local 153
 comments 147
 labels 147, 152
 macros 147
 operands 149, 152
 optimizations 157
 preprocessor 147
 registers 152
 stack frame 148
 syntax 145
 variables, local 153
installing CodeWarrior tools 12
instruction cache 81
int size 100
integer formats 100
integral types, in LCF 125
Integrated Development Environment (IDE) 15
interrupt 102
interrupt 111
interworking 107
interworking, defined 107
intrinsic functions 155
 __alloca 155
 __memcpy 155
 See also pseudo-instructions.
introduction 9
IS NITRO EMULATOR
 DS Flash Card 86
 GBA cartridge 86

K

keep_section 122, 124, 136
Kill All 81

L

label 91
labels
 inline assembly 147, 152
LCF. *See* linker command files
LDA pseudo-instruction, pseudo-instructions
 LDA 156
LDCONST psuedo-instruction 156
libraries
 floating point 167
 MSL C/C++ 169
 MSL Extras 169
 runtime 167
line, assembler directive 92
linker
 adding binary files 115
 NITRO-specific 113
 other documentation 99
 See also linker command files
 symbol precedence 116
linker command files 119–142
 * 126
 access permission flags 120, 136
 addr 132
 after 137
 align 133
 alignall 133
 alignment 123
 arithmetic operations 123
 assignment 125
 comments 124
 deadstripping prevention 124
 exception 134
 exception tables 124
 expressions 125
 file selection 126
 force_active 134
 function selection 127
 group 126, 135
 heap size 127
 include 129, 135
 integral types 125
 keep_section 136
 literal 136
 memory 120, 136
 object 127, 138
 overlayid 138

- overlays 129
- ref_include 138
- sections 120, 139–140
- sizeof 140
- sizeof_rom 140
- stack size 127
- static initializers 128
- symbols 125
- variables 125
- writeb 141
- writew 141
- writew 142
- writing data 128, 129
- linking 16
- literal 136
- little_endian 108
- long double size 101
- long long size 101
- long size 100

M

- macros
 - inline assembly 147
- makerom 60
- .mem 64
- memory 136
- memory locations, using as variable names 152
- memory variables 149
- MSL C/C++ libraries 169
- MSL Extras libraries 169
- multi-core debugging 67, 80, 81

N

- new projects, creating 19
- Nintendo CodeGen panel 43
- Nitro Debugger Setting Panel 67
- NITRO LCF Prelinker panel 58
- NITRO MakeRom Postlinker panel 60
- no_byte_access 102
- nonvolatile 157
- number formats 100, 101
 - floating-point 101
 - integers 100

O

- OBJECT 127
- object 127, 138

- One Time PROM 68, 89
- operands
 - inline assembler 149
 - inline assembly 152
- optimizations
 - inline assembly 157
- overlayid. See linker command files 138
- overlays
 - command-line tools 162
 - creating in IDE 159
 - headers 129
 - linker command file 161
 - restrictions 163
 - valid call trees 163
- overview of documentation 9

P

- PDF documentation 9
- postlinkers
 - batchrunner 47
 - nitro makerom 60
- #pragma directives
 - allow_byte 104
 - ASHLA 104
 - avoid_byte 105
 - dead_stripping 105
 - define_section 105
 - generic_symbol_names 106
 - interworking 107
 - little_endian 108
 - profile 108
 - section 108
 - thumb 108
 - warn_byte 108
- preprocessing 17
- preprocessor
 - inline assembly 147
- processor browser 83
- profile 108
- ProfilerClear 84
- ProfilerDump 84
- ProfilerInit 84
- ProfilerSetStatus 84
- ProfilerTerm 84
- project stationery 27
- project window 16
- pseudo-instructions 155

DCD 156
LDCONST 156

R

ref_include 122, 124, 138
register variables 102, 149
registers
 inline assembly 152
regservers.bat 14
release target 27
remote debugging
 settings panel 72
resource file 129
ROM target 27
Run All 81
runtime libraries 167

S

sdata 102
section 108
sections 120, 139–140
settings
 debugger 70
 remote debugging 72
settings panels
 ARM Assembler 33
 ARM Debugger Settings 62
 ARM Linker 53
 ARM Processor 36
 ARM Project 32
 BatchRunner PostLinker 47
 BatchRunner Preprocessor 48
 Binary Converter Settings 42
 ELF Disassembler 49
 Nintendo CodeGen 43
 Nitro Debugger Setting 67
 NITRO LCF Prelinker 58
 NITRO MakeRom Postlinker 60
short double size 101
short size 100
signed char size 100
sizeof 140
sizeof_rom 140
small data 39
source view 73
stack frame
 inline assembly 148

stack size 127
startup code 128
static initializers 128
stationery 30
stationery projects 27
Stop All 81
Stop on application launch 71
subroutine calls, and compiler-generated
 code 107
symbols, in LCF 125
syntax
 checking 17
system browser 83

T

Target Settings
 accessing 29
 configuring 29
 overview 29
target settings panels
 ARM Linker 53
 ARM Processor 36
thread browser 83
thumb 108
thumb instruction mode
 disassembler 73
timer.c 85

U

uninstalling the IDE 13
unsigned char size 100
unsigned int size 100
unsigned long long size 101
unsigned long size 100
unsigned short size 100
using
 variable names as memory locations 152

V

variable
 using names as memory locations 152
variables
 inline assembly 153
 memory 149
 register 149
variables, in LCF 125
vener, defined 107

volatile 157

W

warn_byte 108

watchpoint 68

watchpoints 85

weak 102

writeb 128, 141

writelh 128, 141

writew 128, 142