MULTI: Building Applications for Embedded V850 and RH850

Green Hills Software
30 West Sola Street
Santa Barbara, California 93101
USA
Tel: 805-965-6044
Fax: 805-965-6343
www.ghs.com

DISCLAIMER

GREEN HILLS SOFTWARE MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. Further, Green Hills Software reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Green Hills Software to notify any person of such revision or changes.

Copyright © 1983-2013 by Green Hills Software. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission from Green Hills Software.

Green Hills, the Green Hills logo, CodeBalance, GMART, GSTART, INTEGRITY, MULTI, and Slingshot are registered trademarks of Green Hills Software. AdaMULTI, Built with INTEGRITY, EventAnalyzer, G-Cover, GHnet, GHnetLite, Green Hills Probe, Integrate, ISIM, u-velOSity, PathAnalyzer, Quick Start, ResourceAnalyzer, Safety Critical Products, SuperTrace Probe, TimeMachine, TotalDeveloper, DoubleCheck, and velOSity are trademarks of Green Hills Software.

All other company, product, or service names mentioned in this book may be trademarks or service marks of their respective owners.

For a partial listing of Green Hills Software and periodically updated patent marking information, please visit http://www.ghs.com/copyright_patent.html.

PubID: build_v800-496213
Branch: http://toolsvc/branches/release-branch-2013-5-bto
Date: October 4, 2013

Contents

Preface

Contents

This preface discusses the purpose of the manual, the MULTI documentation set, and typographical conventions used.

About This Book

This book provides comprehensive documentation of the components that make up the MULTI toolchain. It is divided into the following parts:

- *Part I: Using the MULTI Compiler* — Documents how to use the compiler driver to control the toolchain. Includes an introduction to the compiler driver, discussions of commonly-used features of the compiler for your target environment, and descriptions of all the Builder and driver options.
- *Part II: Using Advanced Tools* — Documents the other elements of the toolchain: the assembler, linker, librarian, and utility programs.
- *Part III: Language Reference* — Documents the Green Hills implementation of high level languages, including macros, `#pragma` directives, libraries, headers, and optimizations.
- *Part IV: Appendices* — Documents the use of GNU compiler options and advanced options for customizing various file types.

Note

New or updated information may have become available while this book was in production. For additional material that was not available at press time, or for revisions that may have become necessary since this book was printed, please check your installation directory for release notes, **README** files, and other supplementary documentation.

The MULTI Document Set

The primary documentation for using MULTI is provided in the following books:

- *MULTI: Getting Started* — Describes how to create, build, and debug an example project in MULTI.
- *MULTI: Licensing* — Describes how to obtain, install, and administer Green Hills licenses.
- *MULTI: Building Applications* — Describes how to use the compiler driver and the tools that compile, assemble, and link your code. Also describes the Green Hills implementation of supported high-level languages.
- *MULTI: Configuring Connections* — Describes how to set up your target debugging interface for use with MULTI.
- *MULTI: Debugging* — Describes how to use the MULTI Debugger and its associated tools.
- *MULTI: Debugging Command Reference* — Describes how to use Debugger commands and provides a comprehensive reference of Debugger commands.
- *MULTI: Managing Projects and Configuring the IDE* — Describes how to create and manage projects and how to configure the IDE.
- *MULTI: Scripting* — Describes how to create MULTI scripts and contains information about the MULTI-Python integration.

For a comprehensive list of the books provided with your MULTI installation, see the **toc_*target*.pdf** file located in the **manuals** subdirectory of your installation.

Most books are available in the following formats:

- A printed book (select books are not available in print).
- Online help, accessible from most MULTI windows via the **Help → Manuals** menu.
- An electronic PDF, available in the **manuals** subdirectory of your installation.

Conventions Used in the MULTI Document Set

All Green Hills documentation assumes that you have a working knowledge of your host operating system and its conventions, including its command line and graphical user interface (GUI) modes.

Green Hills documentation uses a variety of notational conventions to present information and describe procedures. These conventions are described below.

Convention	Indication	Example
bold type	Filename or pathname	**C:\MyProjects**
	Command	**setup** command
	Option	**-G** option
	Window title	The **Breakpoints** window
	Menu name or menu choice	The **File** menu
	Field name	**Working Directory:**
	Button name	The **Browse** button
italic type	Replaceable text	**-o *filename***
	A new term	A task may be called a *process* or a *thread*
	A book title	*MULTI: Debugging*
`monospace` type	Text you should enter as presented	Type `help` *`command_name`*
	A word or words used in a command or example	The **wait** [-global] command blocks command processing, where `-global` blocks command processing for all MULTI processes.
	Source code	`int a = 3;`
	Input/output	`> print Test` `Test`
	A function	`GHS_System()`
ellipsis (...) (in command line instructions)	The preceding argument or option can be repeated zero or more times.	**debugbutton** [*name*]...

Convention	Indication	Example
greater than sign (>)	Represents a prompt. Your actual prompt may be a different symbol or string. The > prompt helps to distinguish input from output in examples of screen displays.	`> print Test` `Test`
pipe (|) (in command line instructions)	One (and only one) of the parameters or options separated by the pipe or pipes should be specified.	**call** *proc* | *expr*
square brackets ([ ]) (in command line instructions)	Optional argument, command, option, and so on. You can either include or omit the enclosed elements. The square brackets should not appear in your actual command.	`.macro` *`name`* `[`*`list`*`]`

The following command description demonstrates the use of some of these typographical conventions.

gxyz [-*option*]*...filename*

The formatting of this command indicates that:

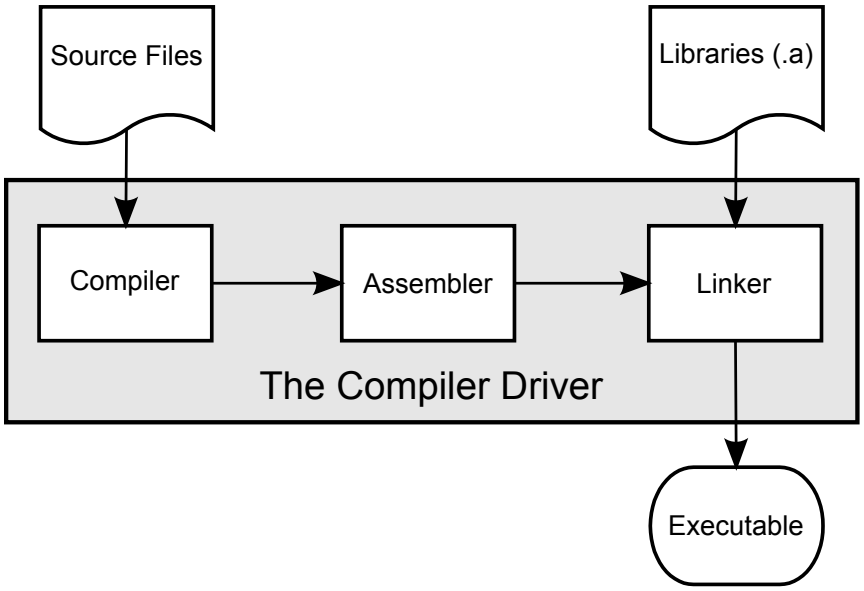
- The command **gxyz** should be entered as shown.
- The option *`-option`* should either be replaced with one or more appropriate options or be omitted.
- The word *`filename`* should be replaced with the actual filename of an appropriate file.

The square brackets and the ellipsis should not appear in the actual command you enter.

The MULTI Toolchain

The C and C++ Compiler Driver

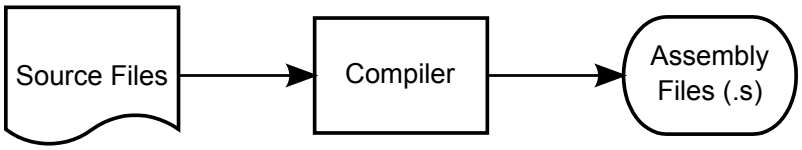
The compiler driver is the command line interface for invoking the components of the toolchain. It takes high-level source and other files as input, and invokes a compiler, the assembler, and the linker, to generate an executable.



To get started using the driver, see Chapter 1, “The Compiler Driver” on page 3.

The Optimizing Compilers

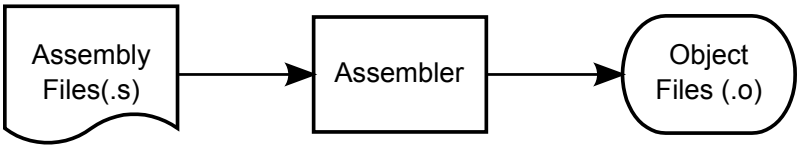
The Green Hills compilers are an integrated family of optimizing compilers that translate high-level language files into assembly language or object files.



There is a compiler for each of C and C++, which is composed of a language-specific front end, a global optimizer, and a target-specific code generator. Compatible subroutine calling conventions allow modules written in different languages to be combined. The compilers are invoked by the Builder or the appropriate compiler driver.

The ease850 Assembler

The Green Hills assembler, **ease850**, translates V850 and RH850 assembly language statements and directives into V850 and RH850 machine code and data formats. The resulting file produced is an object file.



The assembler is ordinarily invoked by the Builder or compiler driver, when required, as part of the compilation process (see "Controlling the Assembler" on page 73). For information about invoking the assembler directly, and for a detailed description of its operations, see Chapter 6, "The ease850 Assembler" on page 361.

Use the **as850** assembler for the V850 target.

Note

The V850 and RH850 compiler defaults to *binary code generation*, which allows the compiler to produce object files directly, without invoking the assembler, and results in reduced compilation times.

The ax Librarian

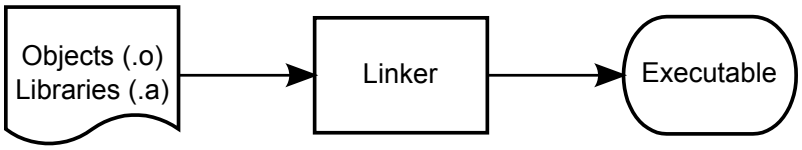
The Green Hills librarian, **ax**, combines object files into a library file.

The linker can search libraries for object files to resolve external references. The linker pulls in an object file from a library only if it is referenced in the program.

The librarian is ordinarily invoked from the Builder or compiler driver (see "Creating Libraries" on page 12). For information about invoking the librarian directly, and for a detailed description of its operations, see Chapter 10, "The ax Librarian" on page 477.

The elxr Linker

The Green Hills linker, **elxr**, combines object files into an executable.



The linker resolves symbol references among all input files, and assigns symbols to memory locations.

The linker is ordinarily invoked by the Builder or compiler driver as part of the compilation process (see "Controlling the Linker" on page 74). For information about invoking the linker directly, and for a detailed description of its operations, see Chapter 9, "The elxr Linker" on page 435.

Optimized Libraries and Header Files

Green Hills provides optimized libraries and header files that support the standard libraries specified by each language. The compiler driver selects the appropriate libraries when linking, and header files when compiling. For more information, see Chapter 16, "Libraries and Header Files" on page 767.

Utility Programs

The Green Hills Utility Programs allow you to analyze and perform various operations on object files, libraries, and executables produced with the Green Hills toolchain. For more information, see Chapter 11, "Utility Programs" on page 487.

V850 and RH850 Recommended Reading

Hardware architecture guides can be downloaded from the Renesas Electronics Web site. It is strongly recommended that users download and familiarize themselves with the Architecture Specifications and Application Binary Interface documentation there.

Part I

Using the MULTI Compiler

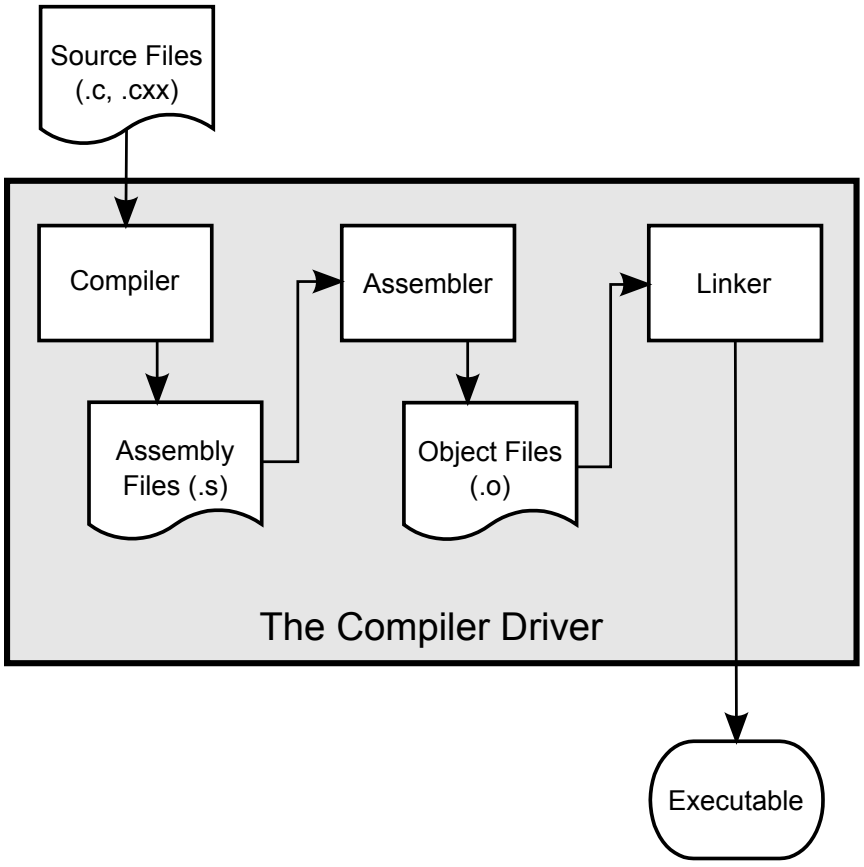
Chapter 1

The Compiler Driver

Contents

The compiler driver is the command line interface for various toolchain components. The driver enables you to specify through various options how the toolchain should process one or more input files into one or more output files. This chapter explains how to use the driver and introduces some of the most important tasks it can perform. For example, it can control all the components of the Green Hills toolchain necessary to produce an executable from high-level source files:

- The optimizing compilers for C and C++, which compile high-level source files into V850 and RH850 assembly language files.
- The **ease850** assembler, which translates V850 and RH850 assembly files into object files.
- The **elxr** linker, which links object files (including object files in libraries) together to generate an executable.



In addition, the driver accepts many other forms of input files (see “Working with Input Files” on page 8) and can produce many different forms of output, (see “Generating Other Output File Types” on page 11).

The driver is often invoked through a higher level interface, such as the MULTI

Builder, makefiles, shell scripts, or batch files. For more information about the MULTI Builder, see *MULTI: Managing Projects and Configuring the IDE*.

Compiler Driver Syntax

The syntax for using the compiler driver is:

driver [*file* | *-option*]...

where:

- *driver* is one of the following:
  - **ccv850 or ccrh850**— for programs that only contain C files
  - **cxv850 or cxrh850**— for programs that contain C and C++ files

  Throughout this manual **ccv850** and **cxv850** are used in examples. However, they can be interchanged with the corresponding equivalent **ccrh850** or **cxrh850** driver as appropriate. The **ccrh850** and **cxrh850** commands differ from **ccv850** and **cxv850** in that they default to **-cpu=rh850g3k**.
- *file* is one or more of the following:
  - C or C++ source file
  - Assembly source file
  - Object file or library of object files
  - Linker directives file (see “Passing Linker Directives Files to the Driver” on page 10)
- ***-option*** is one or more compiler driver options. All of the options are case-sensitive (for example **-l** specifies a library, while **-L** specifies a library directory), and most are host-independent. Some commonly used options are discussed in this chapter. All of the driver options are listed and explained in Chapter 3, “Builder and Driver Options” on page 117.

Files and options are listed on the command line, separated by spaces.

The driver reads all options before processing any files. When two options represent different choices for the same feature, the later choice overrides the earlier one. If

it encounters an unrecognized or invalid option, the driver will ignore it and issue a warning.

After reading the options, the driver processes files in the order they appear on the command line. If an error occurs in one file, processing will continue with the next file. If no errors occur, all object files and libraries will be linked together according to the order specified on the command line.

Building an Executable from C or C++ Source Files

When the driver receives source language files as input, it invokes the appropriate compiler, assembler, and linker to produce an executable.

To build an executable from a C source file called **hello.c**, use the C driver, **ccv850**, and enter the following:

```
ccv850 hello.c
```

To build an executable from a C++ source file called **hello.cxx**, use the C++ driver, **cxv850**, and enter the following:

```
cxv850 hello.cxx
```

In each case, an executable file called **a.out** is generated, together with the following additional files: **hello.o**, **a.map**, **a.dnm** and **a.dla**

The **hello.o** file is the object file which results from passing **hello.c** through the compiler and assembler (the driver always retains the intermediate object files for future re-use). The **a.map** file is a map file produced by the linker. The **a.dnm** and **a.dla** files contain basic debugging information for use with the MULTI Debugger. For instructions on generating full debugging information for MULTI, see "Generating Debugging Information" on page 49.

To build an executable from multiple source files, list the files on the command line, separated by spaces, as follows:

```
ccv850 hello.c foo.c bar.c
```

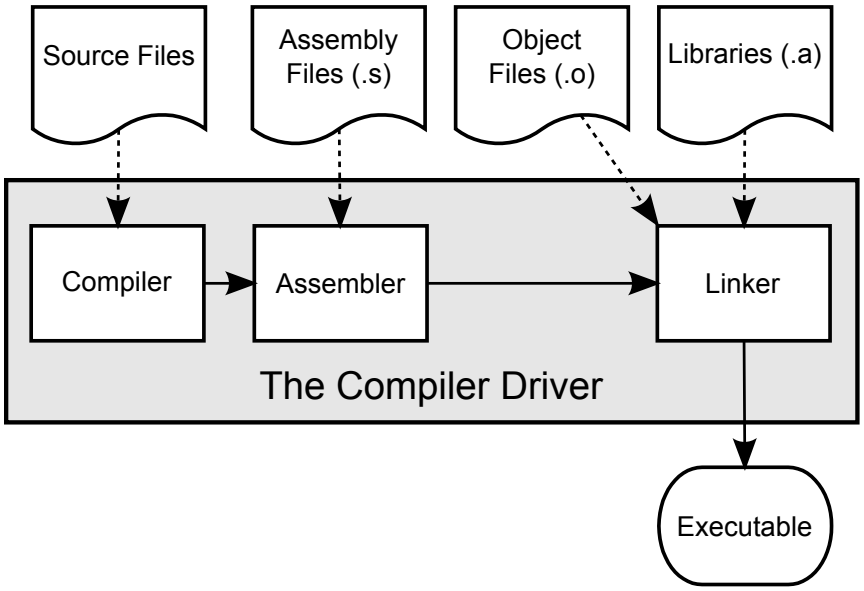
This command generates an executable called **a.out**, the debugging files **a.dnm** and **a.dla**, a map file **a.map**, and three object files, **hello.o**, **foo.o**, and **bar.o**.

Working with Input Files

Note

The example command lines in this section, and in the remainder of this chapter, are shown using the C driver, **ccv850**. They will also work with the C++ driver, **cxv850**.

In addition to high-level language source files, the driver can also process assembly language files, object files, and libraries. The diagram below shows how these files are passed to the various tools in the toolchain.



Recognized Input File Types

The compiler driver determines the type of a file (and hence the compilation steps to perform on it) from its extension. For example, a file with a **.c** extension is a C source file and needs to be compiled with the C compiler, assembled, and then linked.

The following is a partial list of file extensions which the driver recognizes:

- C source files: **.c**, **.i**,
- C++ source files: **.C**, **.cc**, **.cpp**, **.cxx**
- Assembly language files: **.s**, **.asm**, **.800**

- V850 and RH850 assembly language files with C preprocessor directives: **.850**
- Object files: **.o**
- Library files: **.a**
- Linker directives files: **.ld**

Manually Setting Input File Types

On the command line, use **-filetype.*suffix*** to indicate that the next file on the command line has the type appropriate for *suffix*. For example:

```
ccv850 -filetype.cc hello.i
```

compiles **hello.i** as if it were a C++ file.

To process all C files as C++ files, use the following syntax:

```
cxv850 -dotciscxx files
```

Passing Multiple Input File Types to the Driver

To pass assembly files, object files, and libraries to the driver, add them to the command line, separated by spaces. For example:

```
ccv850 hello.c foo.cxx bar.s baz.o libfoo.a
```

C and C++ source files can be included in the same executable as long as one of the following is done so that the program is linked properly:

- Use the C++ driver, **cxv850**.
- Pass **-language=cxx**
- Put at least one C++ source file on the driver command line that is used to link the program.

For more information about mixing C and C++ in the same executable, see “Input Languages” on page 253.

Passing Linker Directives Files to the Driver

Linker Directives (**.ld**) files define the program sections of the executable and assign them to specific regions of memory.

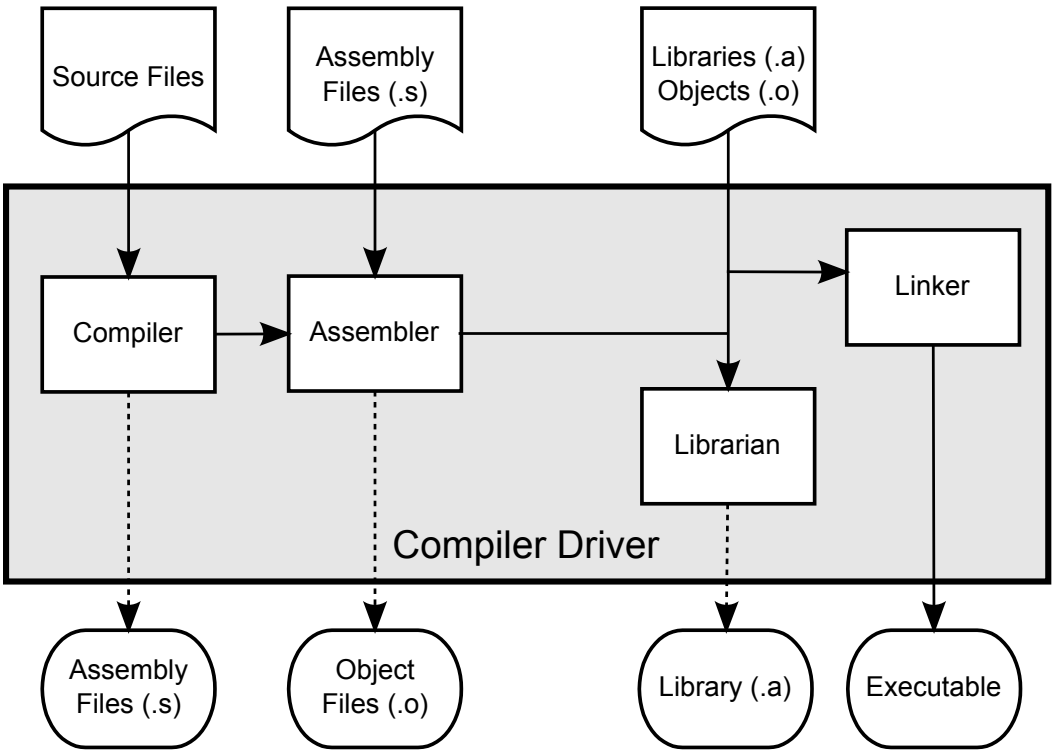
To pass a linker directives file to the driver, add it to the command line as follows (no option is required; the file is recognized by its extension):

```
ccv850 hello.c mylinkfile.ld
```

This command overrides the default linker directive files normally selected by the driver based on the current options. Several default **.ld** files are provided (see "Working with Linker Directives Files" on page 74), and the most appropriate one is used automatically. You can edit the default files to specify your own layout. For full documentation of the syntax, see "Configuring the Linker with Linker Directives Files" on page 441).

Generating Other Output File Types

The driver can be instructed to halt the compilation process at various stages in order to produce assembly files, object files, libraries, or other forms of output.



To instruct the compiler to halt the build process after the compiler has generated assembly files, use the **-S** option as follows:

```
ccv850 hello.c -S
```

This command produces an assembly language file called **hello.s**.

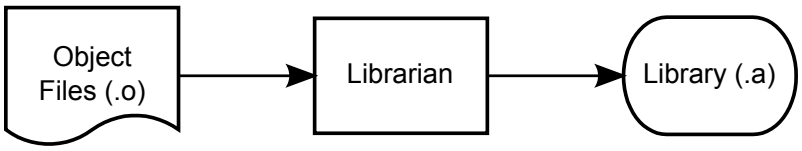
To instruct the compiler to halt the build process after the compiler has generated assembly files and the assembler has translated them to object files, use the **-c** option as follows:

```
ccv850 hello.c -c
```

This produces an object file called **hello.o**.

Creating Libraries

You can pass high-level source files, assembly language files, and object files to the driver, and instruct it to collect them into a library. The driver invokes the compiler and/or the assembler (if required) to produce object files, and then invokes the librarian to combine these object files into a library.



To instruct the driver to create a library from the C source file **hello.c** and the object file **foo.o**, use the **-archive** option as follows:

```
ccv850 hello.c foo.o -archive -o libfoo.a -G
```

This command produces a library of object files called **libfoo.a**, which contains two object files, **hello.o** and **foo.o**.

Note

When using the **-archive** option to create a library, you must use the **-o** option to specify a name for it. The **-G** option is optional, but should be passed if you intend to use the MULTI Debugger.

For more information, see “Creating a Library from the Compiler Driver” on page 478.

Adding and Updating Files in Libraries

To add an object file to an existing library or to update an object file already included in a library, use the **-archive** and **-o** options in the same way you would to create that library. For example, to generate an object file from the C source file **hello.c** and add this object file to library **libfoo.a**, enter the following:

```
ccv850 hello.c -archive -o libfoo.a
```

To perform other operations on libraries, you must invoke **ax** directly. For more information, see Chapter 10, “The ax Librarian” on page 477.

Driver Options for Intermediate Forms of Output

The following table lists all of the driver options that instruct the driver to stop at particular stages of the compilation process.

Note

These options only apply to the compiler driver and cannot be set from the Project Manager.

-archive Runs the **ax** librarian to generate a library instead of running the linker to generate an executable program. For more information, see "Creating a Library from the Compiler Driver" on page 478.
-c Produces an object file (called ***input-file*.o**) for each source file.
-E Runs the preprocessor and writes the preprocessed file output to `stdout` unless the **-o** option is specified. This option can be useful when debugging preprocessor macros and `#include` files. Each preprocessing directive is replaced with a blank line, unless there are several in a row, in which case they are replaced with a single `#line` marker that specifies the line number of the next line of code in the original file.
-merge_archive Runs the **ax** librarian with the **C** and **M** options to generate a merged library instead of running the linker to generate an executable program. For more information, see "Creating a Library from the Compiler Driver" on page 478.
-P Runs the preprocessor and writes the preprocessed file output to ***input-file*.i** by default. Each preprocessing directive is replaced with a blank line.
-Q Produces only an inline file (called ***input-file*.inf**) for each source file. These files contain partially compiled code for functions generated during the first pass of the inliner, together with the list of the include files that the source file depends on (which are used by automatic dependency checking).
-S Produces an assembly file (called ***input-file*.s**) for each source file.
-syntax Checks the syntax without generating code.

Note

See also the driver options associated with the **Linker→Generate Additional Output** option in “Linker Options” on page 219.

Output File Types

The following table lists the types of files generated by the toolchain, MULTI Project Manager and Debugger:

.ael A file generated by **intex** for an INTEGRITY application. It contains information that the Debugger uses to debug your application. If this file is deleted, you must relink the corresponding INTEGRITY application to regenerate it.
bmon.out **gmon.out** **mon.out** Profiling data files that are generated when the program runs and are read only by the toolchain.
.d Source code dependency files generated by the compiler.
.dep Program dependency files generated by the compiler.
.dba **.dbo** **.dla** **.dlo** **.dnm** Debugging information files that are written only by the toolchain, and read by the toolchain and debugger. See “Debugging Options” on page 156.
.graph Call graph information that is written only by the toolchain, and read by the user. See “Linker Output Analysis” on page 230.
.idep Dependency files generated by **Intex**.
.ii **.ti** C++ template information files that are read and written only by the toolchain.
.inf Database files used for intermodule inlining that are read and written only by the toolchain.

.ipf Database files used for interprocedural analysis that are read and written only by the toolchain.
.lcp INTEGRITY-only files that are generated by the driver for **Intex**. It contains the command line used when building with **-kernel** so that **Intex** is able to relink the executable itself.
.lst An assembler source listing that is written by the toolchain, and read only by the user. See “Assembler Options” on page 216
.map A map file generated by the linker that contains information about how the program is laid out in memory. **.map** files are written by the toolchain, and read only by the user. See “Linker Output Analysis” on page 230.
.mem An additional copy of the executable image that has been translated by the **gmemfile** utility. **.mem** is written by the toolchain, and read by non-GHS utility programs. See “Linker Options” on page 219.
.run An additional copy of the executable image that has been translated by the **gsrec** utility. **.run** files are written by the toolchain, and read by non-GHS utility programs. See “Linker Options” on page 219.
.time Files that are generated by the Project Manager when the file being processed does not generate an output file. This allows the Project Manager to track if and when the item was last processed for dependency tracking. **.time** is read and written only by the toolchain.

Controlling Driver Information

The following options control the transfer of information to and from the driver itself.

Note

These options only apply to the compiler driver and cannot be set from the Project Manager.

@file Instructs the driver to read command line arguments from *file*.

-# Displays the command lines generated by the driver (which are used to call the compiler, assembler, and/or linker for processing the input files) without executing them. For compatibility purposes, this option can also be entered as **-dryrun** or **--driver_debug**.
-help Displays a list of some of the more commonly used options with brief descriptions.
-v Displays the command lines generated by the driver as it calls the compiler, assembler, and/or linker for processing of input files.

Using a Driver Options File

When using multiple driver options, you may find that your command lines become overly long and hard to read. You can, instead, enter your options into a plain text file, which you pass to the driver using the ***@file*** option.

All characters in the file other than spaces, tabs, newline characters, and double quotes are literal.

Options and arguments must be separated by spaces, tabs, or newline characters. The double quote (`"`) may be used to include spaces or tabs in a single argument as follows:

- If the argument begins with a `"`, then everything on the line up to the next `"` will be part of the argument, and those quotes will be discarded.
- Otherwise, if a `"` is in the middle of an argument, the `"` will not be discarded, but will cause any whitespace on the line up to the next `"` to be included in the argument.

Note

Due to limitations in the command line parser, the **-bsp** and **-layout** options cannot be entered in a driver options file and must be placed directly on the driver command line, either before or after any ***@file*** option.

Example 1.1. How to Use a Driver Options File

The plain text file **myoptfile** contains the following information:

```
hello.c foo.c bar.c
-G
-Ospeed -OI
-o foo
```

To invoke the driver with the options file contained in **myoptfile**, enter:

```
ccv850 @myoptfile -bsp=sa1 -layout=romrun
```

This command instructs the driver to read **myoptfile** and to:

- Pass files **hello.c**, **foo.c**, and **bar.c** to the compiler, assembler, and linker.
- Generate MULTI debugging information (**-G**).
- Compile the files in such a way as to maximize the speed of the final executable, including using two-pass inlining (**-Ospeed -OI**).
- Name the output executable **foo** (**-o foo**).
- Generate code tailored to the instruction set associated with the `CEB-V850-SA1` board, and assign the program to memory regions of that board using a board-specific linker directives files (**-bsp=sa1 -layout=romrun**).

Note

You cannot include comments in a command file that you pass to the compiler driver with this option. Do not use this option to read a file containing assembler or linker options. Instead, use **-asmcmd=*file*** (see “Assembler Options” on page 216) or **-lnkcmd=*file*** (“Linker Options” on page 219) to pass the file option directly to the assembler or the linker, respectively. Note that linker directives files should be passed to the linker by using a known suffix, such as **.ld** or **.lnk**, rather than using this option.

Using Makefiles

The Green Hills compiler drivers are completely compatible with makefiles. This section discusses the basic issues involved in porting existing makefiles for use with Green Hills compilers.

In each of the following examples, we port a makefile from the fictional compiler **qcc** to the Green Hills compiler **ccv850**. We assume that **qcc** uses fairly standard command line syntax.

Example 1.2. A Very Simple Build

The following is an example of a very simple **qcc** makefile:

```
scanprog: scanprog.c
 qcc -O2 scanprog.c -o scanprog
```

To convert this makefile for use with **ccv850**, you would need to make the following changes:

- Change the invocation of **qcc** to **ccv850**.
- Change the optimization option **-O2** to a Green Hills option, such as **-Ogeneral** (which enables general optimizations).

The resulting **ccv850** makefile will look like the following:

```
scanprog: scanprog.c
 ccv850 -Ogeneral scanprog.c -o scanprog
```

Example 1.3. Makefile Macros, Incremental Build

The following is a common format for a makefile. It uses an incremental build process (building **.c** files to **.o** files, and then linking them). It also uses makefile macros (such as **CC** and **CFLAGS**) to gather commonly-modified parts of the makefile in one place:

```
CC=qcc
CFLAGS=-g -I../include -DITERATIONS=3 -O3 -fprocessor=A
LFLAGS=-L../lib -lxyzlib -fprocessor=A
OFILES=bigprog.o main.o utils.o

bigprog: $(OFILES)
 $(CC) $(LFLAGS) $(OFILES) -o $@

bigprog.o: bigprog.c bigprog.h utils.h
main.o: main.c bigprog.h
utils.o: utils.c utils.h
```

To convert this makefile for use with **ccv850**, you would need to change the macros as follows:

- Change:

  ```
  CC=qcc
  ```

  to:

  ```
  CC=ccv850
  ```

- Change:

  ```
  CFLAGS=-g -I../include -DITERATIONS=3 -O3
  ```

  to:

  ```
  CFLAGS=-G -I../include -DITERATIONS=3 -Ospeed -OI
  ```

Example 1.4. Mixing Languages

In this example, two C files, one V850 and RH850 assembly file, and one C++ file are linked together into a single executable:

```
OFILES=myprog.o utils1.o utils2.o interface.o

myprog: $(OFILES)
 qcc $(OFILES) -o myprog

myprog.o: myprog.c utils.h
 qcc -c myprog.c

utils1.o: utils1.c utils.h
 qcc -c utils1.c

utils2.o: utils1.s
 qasm -c utils1.s

interface.o: interface.cpp
 qc++ -c interface.cpp
```

Porting this example to use **cxv850** is easy because the Green Hills drivers recognize filename extensions and invoke the correct compiler or assembler automatically. Consequently, you would simply need to change all references to **qcc**, **qasm**, and **qc++** to **cxv850**, so that the makefile would read as follows:

```
OFILES=myprog.o utils1.o utils2.o interface.o

myprog: $(OFILES)
 cxv850 $(OFILES) -o myprog

myprog.o: myprog.c utils.h
 cxv850 -c myprog.c

utils1.o: utils1.c utils.h
 cxv850 -c utils1.c

utils2.o: utils1.s
 cxv850 -c utils1.s

interface.o: interface.cpp
 cxv850 -c interface.cpp
```

Green Hills Equivalents to GNU Tools

The following table provides the Green Hills equivalents to common GNU tools:

GNU	Green Hills
`CC=gcc`	`CC=ccv850`
`CXX=g++`	`CXX=cxv850`
`LD=ld`	`LD=cxv850`
`AR=ar`	`AR=ax` `AR=cxv850 -archive`

Use `AR=ax` for libraries written entirely in C, and for which you do not need debug information. For C++ libraries involving instantiable entities such as templates, either compile using the **--link_once_templates** option, or use `AR=cxv850 -archive`, which requires additional modifications to the rules in your makefile that create libraries.

Generating Dependency and Header File Information

These options provide information about your makefile structure:

-H

Prints to `stderr` a list of files opened by `#include` directives during normal compilation. Files are compiled and linked normally. This option can be used together with **-syntax** to print the names of header files without producing any other output.

-MD

-MMD

--no_make_depends

By default, no makefile dependency files are generated. To specify this behavior explicitly, use **--no_make_depends**.

-MD produces the following makefile dependency files:

- One ***input-file*.d** for each object file.
- One ***input-file*.dep** for each program.

Duplicate dependencies are eliminated and only one rule is output for a single object file. The source and program dependency files have different extensions to prevent one file from overwriting another in the case that an object file has the same name as a program file.

The backslash (\) character is used to form continuation lines.

This option has no effect if **-syntax** is passed.

-MMD behaves the same as **-MD**, but does not include system header directories (see **-sys_include_directory**) or C++ headers.

These options only apply to the compiler driver and cannot be set from the Project Manager.

Chapter 2

Developing for V850 and RH850

Contents

This chapter describes the V850 and RH850 target environment and provides information about some of the Builder and driver options that access and control features of that environment.

The Green Hills V850 and RH850 compiler, assembler, and linker comply with the *V850 and RH850 Embedded Application Binary Interface* specification (*V850 and RH850 EABI*).

V850 and RH850 Characteristics

The V850 and RH850 memory is byte-addressed with 32-bit addresses. However, some processors will fix the high bits of the program counter (PC) to zero, depending on the specific implementation's instruction address space capabilities. The V850 supports a 16 MB instruction address space (the high 8 bits of the PC are fixed to zero). Similarly, the V850E supports a 64 MB instruction address space (the high 6 bits of the PC are fixed to zero), while the V850E2 supports a 512 MB instruction address space (the high 3 bits of the PC are fixed to zero). Newer processors such as the RH850 can optionally support up to a full 4 GB of instruction address space (no bits of the PC are fixed to zero). Bits are numbered with bit zero as the least significant bit.

By default, bytes are ordered with the least significant byte of a multiple-byte value stored at the lowest address (little endian format).

However, the bytes of 32-bit instructions are ordered in a special manner: A 32-bit instruction is first divided into two 16-bit units, and then each of the two units is ordered independently in little endian format.

A	**A+1**	**A+2**	**A+3**
20	BC	34	12

Floating-point values use IEEE-754 format (32 and 64 bits) and are in the same byte order as other values.

Note

Although we follow the IEEE-754 floating-point format, floating-point operations might not generate the exact values required by the specification.

On systems where the compiler generates code for a floating-point unit, the floating-point computation is as accurate as the underlying hardware (Green Hills does not supply software libraries to handle cases that the hardware may ignore or signal an exception on). On systems where the Green Hills floating-point emulation routines are used instead of hardware, the floating-point emulation routines are designed for speed, and the results may differ from the IEEE specification, particularly in exceptional cases such as NANs and denormals.

Character encoding is ASCII.

The stack is always 4-byte aligned.

In C and C++ , bitfields are allocated starting at the lowest memory address. Bitfields begin at the least significant bit in little endian mode and at the most significant bit in big endian mode. Each structure, union, and array is aligned to the maximum alignment requirement of any of its components.

C and C++ Data Type Sizes and Alignment Requirements

The following table lists the data type alignments for C and C++:

C/C++ Data Type	Size	Alignment
`char`	8	8
`short`	16	16
`int`	32	32
`long`	32	32
`long long`	64	64
`float`	32	32
`double`	64	64
`long double`	64	64
`*`	32	32
`enum` (default)	32	32

Note

To reduce the size and alignment requirement of the `enum` type to 8 or 16 bits where possible:

Set the **C/C++ Compiler→Data Types→Use Smallest Type Possible for Enum** option to **On** (**--short_enum**). For more information, see "Use Smallest Type Possible for Enum" on page 198.

Structure Packing

This section discusses structure packing and shows you how to specify structure packing options to the compiler. In general, you use the `#pragma pack(n)` directive to instruct the compiler to pack all instances of a particular structure type. You then use the `__packed` type qualifier when declaring any pointer that might point to a field in a packed structure. This section is organized into several parts that contain more detailed information about how structure packing works:

Understanding Structure Packing

Structure packing is a feature you can use to reduce the amount of memory that instances of your structures require. Understanding how structure packing works is important, because this feature affects the alignment of your structures and the fields they contain.

A structure is *aligned to n bytes* if the compiler places it at an address offset by a multiple of *n* bytes from the beginning of the section. By default, a structure's alignment requirement is equal to the largest of any of its fields' alignment requirements. A structure is *naturally aligned* if it is aligned to its alignment requirement. Otherwise, the structure is *misaligned*. Like all data types, a structure's size is a multiple of its alignment requirement, so that multiple instances of the structure remain aligned when the compiler places them in an array.

The Green Hills compilers always allocate fields of a structure in the order specified in the declaration. A field is aligned to *n* bytes if the compiler places it at an address offset by a multiple of *n* bytes from the beginning of the containing (unpacked)

structure. A field is naturally aligned if it is aligned to its alignment requirement. Otherwise, the field is misaligned. For a list of the alignment requirements for each data type, see "C and C++ Data Type Sizes and Alignment Requirements" on page 27.

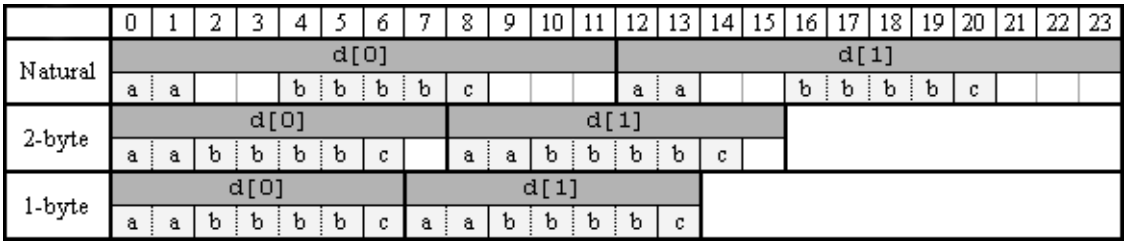
If the compiler is placing a field or structure, and placing it at the next available address would make it misaligned, the compiler will insert bytes of *padding* until it can place the field or structure so that it is naturally aligned.

Structure packing reduces the amount of padding the compiler inserts between fields in a structure. When using structure packing, you give the compiler a *packing alignment*. When the compiler places a field in a packed structure, it aligns that field to its alignment requirement, or the packing alignment, whichever is smaller. This means that the compiler will insert at most *n*-1 bytes of padding between fields, where *n* is the packing alignment. The compiler aligns a packed structure itself to the largest of any of its fields' alignment requirements, or the packing alignment, whichever is smaller.

As an example, say you define struct `s` like so:

```
struct s {
  short a;
  int b;
  char c;
} d[2];
```

The following diagram shows how the compiler will allocate `d[2]` using its natural alignment, with a 2-byte packing alignment, and with a 1-byte packing alignment:



Notice that when the compiler uses structure packing, `d[2]` takes up fewer bytes, but some of the fields and structures are misaligned.

Using #pragma pack to Pack All Instances of a Structure Type

You can instruct the compiler to pack all instances of a particular structure type by using the `#pragma pack(n)` directive, where *n* is the packing alignment in bytes, is a power of two, and is less than or equal to `__ghs_max_pack_value`. The directive must appear before the beginning of the structure definition, *not* inside it. The directive affects all subsequent structure definitions until the next `#pragma pack(n)` directive, or the end of the translation unit. If you do not specify *n*, then the compiler does not pack subsequent structures, unless you have set the **C/C++ Compiler→Alignment and Packing→Packing (Maximum Structure Alignment)** Builder option (see "Using the Packing Builder Option to Pack All Structures" on page 31 for more information).

Example 2.1. #pragma pack Scope

When you use the `#pragma pack(n)` directive around a structure declaration, the compiler packs all instances of that structure. The directive does not affect instance declarations.

```
struct s {
  short a;
  int b;
  char c;
} j;
#pragma pack(2)
struct s k;
struct s2 {
  short a;
  int b;
  char c;
} x;
#pragma pack()
struct s2 y;
```

The size of both `j` and `k` is 12. The compiler will place two bytes of padding between field `a` and field `b`. Because you already defined `struct s`, the `#pragma pack(2)` directive does not affect the declaration of `k`.

The size of both `x` and `y` is 8. The compiler will not place padding between field `a` and field `b`. Because you already defined `struct s2`, the `#pragma pack()` directive does not affect the declaration of `y`.

Using the Packing Builder Option to Pack All Structures

You can set the packing alignment for all structures in your program by using the **C/C++ Compiler→Alignment and Packing→Packing (Maximum Structure Alignment)** Builder option.

Set the **C/C++ Compiler→Alignment and Packing→Packing (Maximum Structure Alignment)** Builder option (**-pack=*n***), where *n* is the packing alignment in bytes, and has a value of 1, 2, 4, or 8.

Note

When using this Builder option on a project, you use the `__packed` qualifier on any pointer that points to a field in any structure declared in that project. See “Pointing to Packed Structures with the __packed Type Qualifier” on page 31 for more information.

Pointing to Packed Structures with the __packed Type Qualifier

The `__packed` type qualifier tells the compiler that a particular pointer might point to a misaligned structure or field. compiler uses this information to generate additional code that handles accesses to misaligned fields in software, if necessary.

Because any field in a packed structure might be misaligned, you must use the `__packed` type qualifier when declaring any pointer that points to a field in a packed structure. For instance, if you pack the following structure:

```
#pragma pack(2)
struct s {
  short a;
  int b;
  char c;
} x;
#pragma pack()
```

and then create a pointer to a field in that packed structure:

```
int *p = &(x.b);
```

Because `&(x.b)` points to a misaligned field in a packed structure, the compiler issues warning 1973, and the program might exhibit unexpected behavior. Instead, use:

```
__packed int *p = &(x.b);
```

The program now runs as expected, because the compiler generates code to handle misaligned accesses through `*p`.

If you take the address of the entire packed structure into a pointer of that structure's type, it is not necessary to use the `__packed` type qualifier, because the structure type is already declared as packed. For example:

```
/* Correct */
struct s *p1 = &x;

/* Correct, but __packed is unnecessary here*/
__packed struct s *p2 = &x;

char buf[sizeof(struct s)];
memcpy(buf, &x, sizeof(struct s));

/* Correct - p2 can point to a struct object with 1-byte alignment */
p2 = (__packed struct s *)buf;

/* Incorrect - buf might not have the 2-byte alignment required by struct s */
p1 = (struct s *)buf;
```

Note

Do not declare objects as both `__packed` and `volatile` simultaneously, or put `volatile` fields inside a packed structure. Loads and stores of such objects might exhibit unexpected behavior.

Register Usage

Thirty-two 32-bit general purpose registers are used for both integer values and single-precision floating-point values. A set of system registers is also available.

Double precision 64-bit floating-point values are contained in two adjacent 32-bit registers. The low word is stored in the lower register.

The V850 and RH850 toolchain supports three software modes of register usage, controlled with the **Target→Register Mode** option (**-registermode=22|26|32**),

which allows you to specify 22, 26, or 32 registers. For more information, see “Register Mode” on page 120.

The register modes are shown below:

Register	Alternate Name	Common Usage	Notes
r0		Zero (read-only)	
r1		Address generation in compiler and assembler	
r2		Temporary	Unused if the **Target→Register r2** option is enabled (**-reserve_r2**).
r3	sp	Stack pointer	
r4	gp	Global pointer (SDA and PID base register)	
r5	tp	ROSDA base register	
r6 – r9		Parameters	
r10		Function return value/temporary	
r11		High part of 64-bit return value/temporary	
r12		Temporary	
r13		Temporary	
r14		Temporary	
r15		Temporary	Unused in 22-register mode
r16		Temporary	Unused in 22-register mode
r17		Temporary	Unused in 22- and 26-register modes
r18		Temporary	Unused in 22- and 26-register modes
r19		Temporary	Unused in 22- and 26-register modes
r20		Permanent	Unused in 22- and 26-register modes
r21		Permanent	Unused in 22- and 26-register modes

Register	Alternate Name	Common Usage	Notes
r22		Permanent	Unused in 22- and 26-register modes
r23		Permanent	Unused in 22-register mode
r24		Permanent	Unused in 22-register mode
r25		Permanent	
r26		Permanent	
r27		Permanent	
r28	fp	Frame pointer/permanent	
r29		Local PIC base register/permanent	
r30	ep	Temporary by default. TDA base register /permanent when TDA is enabled.	Not allocated to if the **-no_allocate_ep** option is used.
r31	lp	Link pointer	

Calling Conventions

For a module to access arguments and local variables on the stack, the stack must always be located at a 4-byte boundary at the entry to a procedure, subroutine, or function. If the first procedure, subroutine, or function (called by the operating system or system library) is called correctly, then every subsequent one will be called correctly.

All call arguments following the first argument are evaluated first, from left to right, and then the remaining non-call arguments are evaluated from left to right. For example:

```
foo(a+b, bar(), c, zog(), d);
```

The arguments are evaluated in this order: `bar()`, `zog()`, `a+b`, `c`, `d`.

Another example:

```
foo(bar(), a+b, c, zog(), baz());
```

The arguments are evaluated in this order: `zog()`, `baz()`, `bar()`, `a+b`, `c`.

In C and C++, each scalar argument is extended to a 32-bit value after it is evaluated, unless the corresponding formal parameter has floating-point type and an ANSI prototype is visible. In this case, the argument is converted into either a 32-bit or 64-bit floating-point value according to the formal parameter.

In C and C++, each floating-point argument is extended to a 64-bit value after it is evaluated, unless the corresponding formal parameter is either a single-precision floating point or an integer type and an ANSI prototype is visible. If the formal parameter is single-precision, the argument is converted to a 32-bit floating-point value. If the formal parameter is scalar, the argument is converted to a 32-bit scalar value.

Any further type conversion is performed upon entry to the called procedure.

Arguments are assigned stack offsets from left to right. The first argument is always at offset zero. The size of the first argument is rounded up to a multiple of four bytes to determine the offset of the second argument. If the second argument requires 8-byte alignment and its offset would not otherwise be a multiple of eight bytes, the second argument's offset is increased by four bytes. This is repeated until offsets have been assigned to all arguments. The size of the entire argument area is then increased if necessary so that it will also be a multiple of four bytes. If this argument area is larger than 16 bytes, then a space large enough to hold this argument area less 16 bytes is present on the stack immediately before the call.

In general, the arguments are allocated to the stack according to their stack offsets, unless it is possible to place them in registers.

Arguments with offsets 0, 4, 8, and 12 are placed in parameter registers `r6` through `r9`, respectively, including single-precision and double-precision floating-point numbers. Thus in C and C++, scalar, pointer, and floating-point arguments are eligible to pass in registers, as well as some structures and unions. All remaining arguments are passed on the stack.

A varargs function in *K&R* only has `va_alist` as its arguments, and they are all considered to be passed in memory. Therefore on entry to a varargs function, all the parameter registers (`r6` through `r9`) are first saved on the stack so that all arguments can be accessed from the stack.

For a stdargs function in ANSI, all arguments are passed as they would be for a normal function call. However, on entry to a stdargs function, if `"..."` is in one

of the parameter registers (`r6` through `r9`), that register plus all parameter registers following it are first saved on the stack, so that all arguments starting from `"..."` can be accessed from the stack.

On the V850, V850E, and V850E2: A call to a procedure, subroutine, or function uses a `jarl routine, lp` instruction that saves the return address in `r31` (`lp`). The return uses a `jmp [r31]` instruction.

Return values that are scalar or pointer are returned in `r10`, sign- or zero-extended to 32 bits for types smaller than 32 bits. 32-bit floating-point values are returned in register `r10`. 64-bit floating-point values are returned in the register pair `r10`, `r11`.

To call a function that returns a structure or union in C and C++, the address of a temporary of the return type is passed by the caller in `r6`. The function returns the structure value by copying the return value to the address pointed to by `r6` and copies `r6` into `r10` before returning to the caller.

A procedure, subroutine, or function call is assumed to destroy the contents of all registers except the permanent registers (32-register mode: `r20` through `r29`; 26-register mode: `r23` through `r29`; 22-register mode: `r25` through `r29`).

Access to parameters or local stack storage are always relative to the stack pointer, `r3`, even if a frame pointer is set up.

If the **Advanced→Debugging Options→Generate Target-Walkable Stack** option is enabled (**-gtws**), a frame pointer will be set up in `r28`.

The Stack

In general, the stack for the current subroutine resembles the following:

<table>
<tr><td rowspan="4">High Address

(previous frame)</td><td>nth parameter word</td><td rowspan="3"></td></tr>
<tr><td>(n-1)th parameter word</td></tr>
<tr><td>.
.
.
6th parameter word</td></tr>
<tr><td>5th parameter word</td><td><code>sp0</code> (= previous stack pointer)</td></tr>
</table>

(*current frame*)	16-byte parameter area (only if necessary)	`sp + framesize - 4` (`= sp0 - 4`) `sp + framesize - 16`
	return address (`lp`)	`sp + framesize - parmarea - 4`
	frame pointer (`fp`)	`sp + framesize - parmarea - 8`
	permanent registers	
	local variables	
	nth argument word	
	(*n*-1)*th* argument word	
	. . .	
	6*th* argument word	
	5*th* argument word	`sp` (`= sp0 - framesize`)
(*next frame*) Low Address	. . .	

In this diagram, *parameters* are passed to the subroutine. If the subroutine has more than 16 bytes of parameters, the parameter words in excess of 16 bytes are presented on the stack as shown.

Similarly, if the subroutine passes more than 16 bytes of *arguments* to its children subroutines, the arguments in excess of 16 bytes are stored on the stack immediately before the call.

The *16-byte parameter area* is present if any of the first four parameter words of the subroutine belongs to a structure passed by value (*parmarea* = 16), otherwise it is not (*parmarea* = 0).

If the subroutine makes any calls, the link pointer `lp` (`r31`) is stored on the stack.

If the **Advanced→Debugging Options→Generate Target-Walkable Stack** option is enabled (**-gtws**), the link pointer `lp` is always stored on the stack (regardless of

whether the subroutine makes any calls) and a frame pointer is set up in `r28`. The frame pointer is , then stored on the stack immediately below the link pointer.

If the subroutine runs out of temporary registers to hold local variables, the local variables not allocated to temporary registers are stored on the stack.

Because permanent registers must be preserved across calls, any permanent registers used by the subroutine are stored on the stack. Permanent registers are stored in increasing order, contiguously, but not necessarily consecutively, at increasing offsets from the stack pointer. For example, if `r22`, `r24`, `r25`, and `r27` are used by the subroutine and the subroutine has only one call and is passing 20 bytes of arguments to that call, the following is part of the stack:

<table>
<tr><td rowspan="6">High Address

(current frame)</td><td><code>r22</code></td><td rowspan="2"></td></tr>
<tr><td><code>r24</code></td></tr>
<tr><td><code>r25</code></td><td><code>sp+16</code></td></tr>
<tr><td><code>r27</code></td><td><code>sp+12</code></td></tr>
<tr><td>local variables</td><td><code>sp+8</code>

<code>sp+4</code></td></tr>
<tr><td>5th argument word</td><td><code>sp</code></td></tr>
<tr><td>(next frame)
Low Address</td><td>.
.
.</td><td></td></tr>
</table>

Note

When **-prepare_dispose** is used, the return address register (`lp`) and frame pointer register (`fp`) are saved in the permanent register area. `lp` and `fp` are saved in the same order as other registers in that area. For more information about **-prepare_dispose**, see “Target-Specific Support” on page 38.

Target-Specific Support

Since `asm` statements are not portable, the Green Hills V850 and RH850 compiler provides special inline intrinsic functions. For a complete listing of supported intrinsic functions and associated documentation, please refer directly to the file **include/v800/v800_ghs.h** contained within the compiler installation directory.

When compiling for V850E and later processors you can prevent the compiler from using the `callt` instruction by setting the **Target→Instruction Set→Epilogues and Prologues via callt** option to **Off** (**-no_callt**).

Even when you specify `no_callt`, the compiler continues to generate interrupt routines that save the `CTPSW` and `CTPC` registers that `callt` uses. To disable this behavior, you must also set the **Target→Instruction Set→Save CTPSW and CTPC registers in Interrupt Routines** option to **Off** (**-ignore_callt_state_in_interrupts**). If library functions are used for function prologues and epilogues, these registers might still be saved.

To instruct the compiler to use the `prepare` and `dispose` instructions for function prologues and epilogues, set the **Target→Instruction Set→Epilogues and Prologues via prepare and dispose** option to **On** (**-prepare_dispose**). For most functions, this is as small as the `callt` prologue, and is always faster.

SIMD Vector Support

Intrinsic functions to support the SIMD vector instructions are available on the RH850 and later processors when SIMD support (**-rh850_simd**) is enabled. The SIMD intrinsic functions are documented in **include/v800/v800_simd.h**, which is contained within the compiler installation directory.

To facilitate use of the intrinsic functions, the following vector data types are available:

```
__ev64_u16__
__ev64_s16__
__ev64_u32__
__ev64_s32__
__ev64_u64__
__ev64_s64__
__ev64_opaque__
__ev128_opaque__
```

All `ev64` types are 8 bytes in size, and the `ev128` type is 16 bytes in size. In the list of types above, the first six types are for vectors of 16-bit, 32-bit, and 64-bit elements (both unsigned and signed). The `__ev64_opaque__` type is a special vector type that can match any of the other `ev64` types. A function prototyped to take an `__ev64_opaque__` type may be directly passed any of the other `ev64`

types without warnings or errors. The `__ev128_opaque__` type is provided solely for the few vector intrinsic functions which must return 128 bits worth of data. No other 128-bit vector types are available. The `__ev_create_vec128` and `__ev_get_vec64` intrinsic functions are provided to create the 128-bit vector type from two 64-bit vectors and to extract the component 64-bit vectors, respectively.

Example 2.2. Simple initialization of a vector

```
#include <v800_ghs.h>

__ev64_s32__ func(__ev64_s16__ v1)
{
    __ev128_opaque__ v128;
    v128 = __ev_pki16i32(v1);
    return __ev_vadd_w(__ev_get_vec64(v128, 0),
                       __ev_get_vec64(v128, 1));
}
```

Example 2.3. Basic vector manipulation

```
#include <stdio.h>
#include <v800_ghs.h>

void print_vec_s16(const char *name, __ev64_s16__ vec)
{
    printf("%s: { %d, %d, %d, %d }\n", name,
     __ev_get_s16(vec, 0),
     __ev_get_s16(vec, 1),
     __ev_get_s16(vec, 2),
     __ev_get_s16(vec, 3));
}

__ev64_opaque__ test(__ev64_opaque__ vec)
{
    print_vec_s16("before", vec);
    vec = __ev_vadd_h(vec, __ev_create_s16(1,2,3,4));
    print_vec_s16(" after", vec);
    return vec;
}

int main()
{
    __ev64_s16__ v1 = {100,200,300,400}, v2;
```

```
    v2 = test(v1);
    print_vec_s16("    v1", v1);
    print_vec_s16("    v2", v2);

    return 0;
}
```

Software Trace Logging Support

The software trace instructions on the RH850 (`DBCP` and `DBPUSH`) can be used to send log messages directly to the trace buffer.

Logging User Data and Messages

The RH850 toolchain provides specialized intrinsic functions that provide the ability to log user variables to the processor's trace buffer. Each of the functions takes a tag name as the first argument, which must follow the same naming rules as C identifiers. The tag names are used in the logging message and can, but need not, correspond to actual identifiers in the program.

The following table lists the logging functions, which are defined in the file **include/v800/rh850_eagle.h**, contained within the compiler installation directory.

Logging Function	Description
`EAGLE_LogU32(tag, value)`	Logs a user-provided 32-bit integer.
`EAGLE_LogU64(tag, value)`	Logs a user-provided 64-bit integer.
`EAGLE_LogBool(tag, value)`	Logs a user-provided Boolean value.
`EAGLE_LogPtr(tag, value)`	Logs a user-provided pointer.
`EAGLE_LogStr(tag, value)`	Logs a user-provided string.

`EAGLE_LogNdxU32(tag, index, value)`	Logs a user-provided 32-bit integer with index.
`EAGLE_LogNdxU64(tag, index, value)`	Logs a user-provided 64-bit integer with index.
`EAGLE_LogNdxBool(tag, index, value)`	Logs a user-provided Boolean value with index.
`EAGLE_LogNdxPtr(tag, index, value)`	Logs a user-provided pointer with index.
`EAGLE_LogNdxStr(tag, index, value)`	Logs a user-provided string with index.

`EAGLE_LogPrint(tag, string)`	Logs a user-provided message.
`EAGLE_LogPrintf(tag, format, ...)`	Logs a user-provided message with printf-style formatting.

There are three categories of logging functions listed above: those that log a value, those that log a value and an index, and those that log user messages. The following examples, show how they are used.

Example 2.4. Logging scalar values

```
#include <rh850_eagle.h>

long long g64 = 0x0102030405060708;
void log_values(int v32, char *str)
{
    EAGLE_LogU32(v32, v32);
    EAGLE_LogU64(g64, g64);
    EAGLE_LogBool(v32_is_even, (v32&1)==0);
    EAGLE_LogStr(str, str);
}

int main()
{
    log_values(0x1a2b3c4d, "test string");
    return 0;
}
```

This example logs four values, resulting in the following log messages in the trace list window:

```
USER U64: v32 = 0x000000001a2b3c4d
USER U64: g64 = 0x1020304050607080
USER BOOL: v32_is_even = FALSE
USER STRING: str = "test string"
```

Example 2.5. Logging array values

```
#include <rh850_eagle.h>

long long llarray[] = {
    0x0000000000000000, 0x1111111111111111,
    0x2222222222222222, 0x3333333333333333
};
```

```
void log_values(int ndx1, int ndx2, int ndx3)
{
    const char *strarray[] = {
        "element 0", "element 1",
        "element 2", "element 3"
    };

    EAGLE_LogNdxU64(llarray, ndx1, llarray[ndx1]);
    EAGLE_LogNdxU64(llarray, ndx2, llarray[ndx2]);
    EAGLE_LogNdxStr(strarray, ndx3, strarray[ndx3]);
    EAGLE_LogNdxBool(booltest, 2, 1);
    EAGLE_LogNdxBool(booltest, 7, 0);
}

int main()
{
    log_values(2, 1, 3);
    return 0;
}
```

The index logging functions provide a convenient way to generate a log message with a tag of *tagname[index]*. The trace messages for the above example are as shown below:

```
USER U64: llarray[2] = 0x2222222222222222
USER U64: llarray[1] = 0x1111111111111111
USER STRING: strarray[3] = "element 3"
USER BOOL: booltest[2] = TRUE
USER BOOL: booltest[7] = FALSE
```

Example 2.6. Logging user messages

```
#include <stdio.h>
#include <rh850_eagle.h>

void func1(int val1, int val2)
{
    EAGLE_LogPrintf(msg_func1, "In func1(): val1=%d, val2=%d",
                    val1, val2);
}

void func2(char *msg)
```

```
{
    EAGLE_LogPrint(msg_func2, msg);
}

int main()
{
    EAGLE_LogPrint(msg_main, "Start of main()");
    func1(100, 200);
    func2("Message from func2()");
    EAGLE_LogPrint(msg_main, "End of main()");
    return 0;
}
```

The logging function without formatting, `EAGLE_LogPrint` is not fundamentally different from logging a string. Note that the version with formatting requires that **stdio.h** is included before **rh850_eagle.h**. The trace message for this example are shown below:

```
USER STRING: msg_main = "Start of main()"
USER STRING: msg_func1 = "In func1(): val1=100, val2=200"
USER STRING: msg_func2 = "Message from func2()"
USER STRING: msg_main = "End of main()"
```

Run-time Overhead

When logging data that is constant and known at compile time, only the PC is logged to the trace buffer, requiring a single `DBCP` instruction. The tag and value are stored in a section of the ELF file that is not downloaded to the target. When the trace is retrieved, MULTI reconstructs the log message. When variable data is logged, the compiler inserts code to log the value. In the case of integer and Boolean values, this can be done with a single `DBPUSH` instruction. For strings, a library routine is called to log the string with one or more `DBPUSH` instructions. The indexed form of the logging functions are similar to the non-indexed ones, except that have one more instruction of overhead, to log the index.

Logging user messages without formatting has the same overhead as logging a string. When formatting is used, there is also the additional overhead to format the message. This overhead is equivalent to performing a call to `sprintf`.

Configuration

The EAGLE header file, **rh850_eagle.h**, must be included in order to use any of the EAGLE logging functions. If the `EAGLE_LogPrintf` function is used, then **stdio.h** must be include before **rh850_eagle.h**. By default all logging tags are enabled, but the behavior can be configured by defining `EAGLE_CONFIG_LOGGING` before including **rh850_eagle.h** to one of the values shown below.

Value	Code	Instrumentation
0	No	No code generated
1	Yes	Generated, but disabled
2	Yes	Generated and enabled (*Default*)
3	Yes	Generate, state determined on a per-tag basis

Logging Function Entry and Exit (FEE) Events

The RH850 toolchain can be configured to log function entry and exit events for all or selected functions. A software trace instruction is inserted at the entry of a function and at all exit points. Compiler instrumentation can be enabled with **-gen_entry_exit_log** for basic function entry and exit logging, or with **-gen_entry_exit_arg_log** for entry and exit logging with function arguments. For more information, see “Profiling - Entry/Exit Logging” on page 157 or “Profiling - Entry/Exit Logging with Arguments” on page 157.

At link-time, if a module has been compiled with FEE logging, the log points can be enabled, disabled, or stripped. For more information, see “Profiling - Entry/Exit Linking” on page 158.

This implementation allows you to compile your program with instrumentation, and easily disable or strip it away in the linker without having to recompile when you no longer need logging. To enable FEE logging on a function-by-function basis, use `#pragma ghs function_entry_exit` (see “Green Hills Extension Pragma Directives” on page 753) or `attribute((entry_exit_log))` (see `entry_exit_log` on page 678).

Specifying a V850 and RH850 Target

The Builder and compiler drivers support many of the most popular V850 and RH850 boards and the entire V850 and RH850 family of microprocessors. The compilers can produce more efficient code if you specify the target on which you intend to run the finished executable. Target options may affect the predefined preprocessor symbols, the compiler, the assembler, and the selection of header files, libraries, and linker directive files used by the linker.

Many of the most popular V850 and RH850 boards are supported through tailored linker directives files that greatly simplify the process of linking and loading an executable. To specify a particular target board:

When creating a new project with the **Project Wizard** (see the documentation about creating a project in the *MULTI: Managing Projects and Configuring the IDE* book), choose the appropriate target board.

To specify a target board with the driver, pass the **-bsp *board*** option. To obtain a list of supported boards, enter **-bsp=?** (or **-bsp=\?** in many shells).

The **-bsp=** option may imply other settings according to the requirements of your board. It also allows you to use the **-layout=** option. **-bsp=** and **-layout=** select the linker directive file used by the toolchain when building your project. For more information about the **-layout=** option, see “Assigning Program Sections to ROM and RAM” on page 81.

If your board is not supported, you should specify your target processor instead. Specifying a target processor enables the driver to determine the most efficient instruction set to use in code generation.

To specify a target processor:

When creating a project with the **Project Wizard** (see the documentation about creating a project in the *MULTI: Managing Projects and Configuring the IDE* book), expand the **Generic** option in the **Board Name** box, and choose the appropriate processor. This corresponds to the **-bsp=generic** driver option.

To specify a target processor with the driver, pass the **-cpu=*cpu*** option. To obtain a list of supported processors, enter **-cpu=?** (or **-cpu=\?** in many shells).

V850 and RH850 Processor Variants

The following table lists the supported V850 and RH850 processor variants. The default processor type is V850 if the **ccv850** driver is used and RH850G3K if the **ccrh850** driver is used.

All V850 and RH850 CPU macros are also defined without the two trailing underscores.

Processor	Driver Option -cpu=	Predefined Macro Name	Defaults	
			Library Directory	Floating Point
V850	**v850**	`__V850`	**v850**	Software
V850E	**v850e**	`__V850E__`	**v850e**	Software
V850E1F	**v850e1f**	`__V850F__`	**v850f**	Mixed*
V850E2	**v850e2**	`__V850E2__`	**v850e**	Software
V850E2R	**v850e2r**	`__V850E2R__`	**v850e2r**	Hardware
V850E2V3	**v850e2v3**	`__V850E2V3__`	**v850e2v3**	Hardware
V850E3	**v850e3**	`__V850E3__`	**rh850**	Hardware
		`__V850E3V5__`		
RH850G3K	**rh850**	`__RH850__`	**rh850**	Software
	rh850g3k**	`__RH850G3K__`		
		`__V850E3__`		
		`__V850E3V5__`		
RH850G3M	**rh850g3m**	`__RH850__`	**rh850**	Hardware
		`__RH850G3M__`		
		`__V850E3__`		
		`__V850E3V5__`		
RH850G3H	**rh850g3h**	`__RH850__`	**rh850**	Hardware
		`__RH850G3H__`		
		`__V850E3__`		
		`__V850E3V5__`		

* Hardware support for 32-bit, and software support for 64-bit floating point.

** **-cpu=rh850** and **-cpu=rh850g3k** are synonymous.

The V850E3 and RH850 are synonymous within the compiler and Builder, as they both use the V850E3V5 instruction set architecture.

Enabling Debugging Features

Generating Debugging Information

To generate debugging information for use with the MULTI Debugger:

Set the **Debugging→Debugging Level** option to **MULTI** (**-G**).

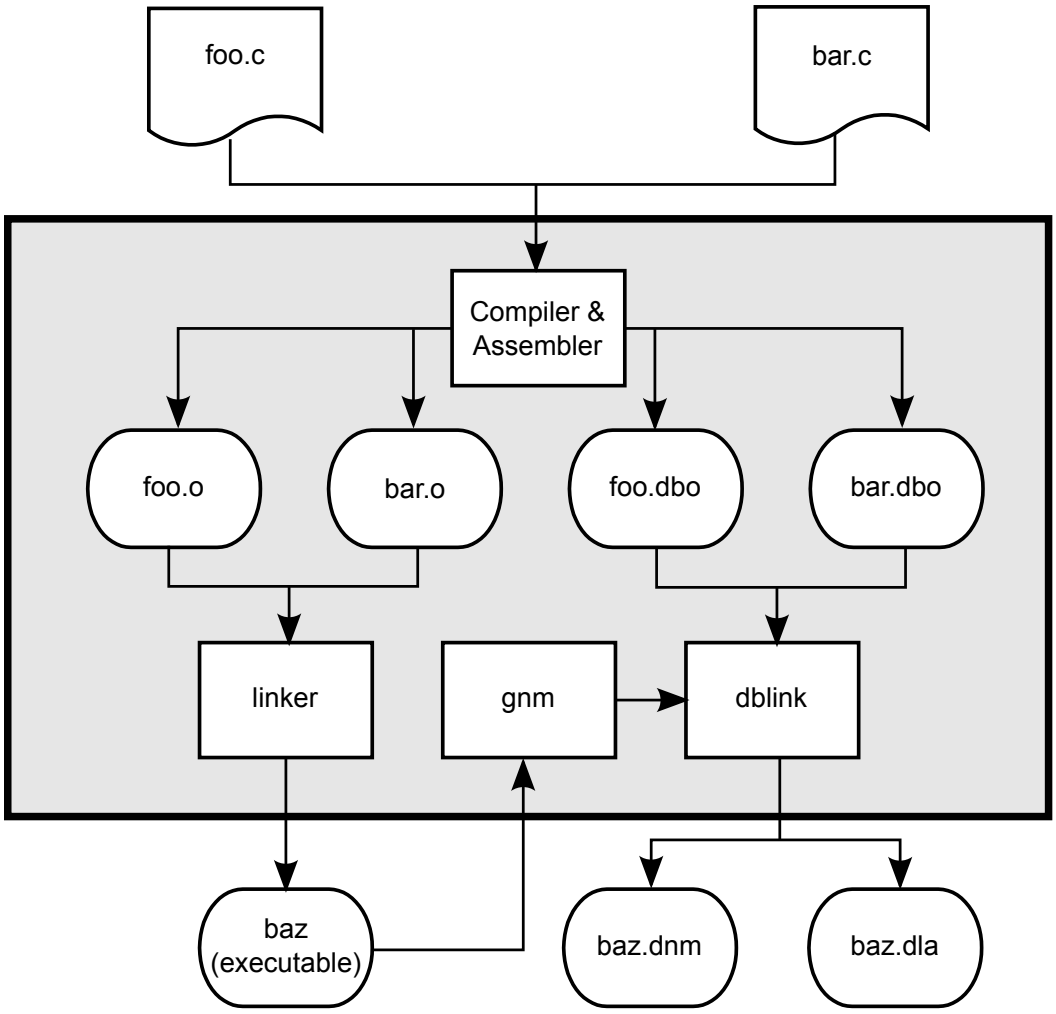
For more information about debugging options, see "Debugging Level" on page 156 and "Debugging Options" on page 156.

Debugging information is only created or updated when you create an executable or a library with the Builder or driver. It is not created when the linker or librarian are invoked separately. If you move an executable, library, or object file, you must also move the associated debugging information files. When moving these files, either preserve the original time stamps of all the files or ensure that the **.dnm** file has a more recent time stamp than the executable's time stamp. Otherwise, it may be necessary to rebuild or relink your program before you can use the MULTI Debugger. For more information, see the documentation about starting the MULTI Debugger in the *MULTI: Debugging* book.

For information about using the MULTI Debugger, see *MULTI: Debugging*.

How the Compiler Generates Debug Information

The diagram below illustrates the creation of debugging information:



As the compiler and assembler translate each C or other high-level language file into an object file, they also generate associated debugging information. For each object file created, a separate file with the same name but a **.dbo** extension is produced to hold this information. After the executable is linked together, these individual debugging information files are combined into a structure that can be quickly searched and incrementally loaded.

In the diagram, the files **foo.c** and **bar.c** are passed to the toolchain, which generates files **foo.o** and **bar.o** and links them together to generate an executable, **baz**. In addition, the debugging information files **foo.dbo** and **bar.dbo** are generated and passed to the **dblink** utility, which also extracts the executable's namelist information (via the **gnm** utility), and generates the following two files:

- **baz.dnm** — the debug symbol table for **baz**.

- **baz.dla** — an archive of the **.dbo** files associated with **baz**

When the executable is opened in the MULTI Debugger, **baz.dnm** controls the loading of debugging information from **baz.dla** as necessary.

The creation of debugging information for libraries is performed in the same way as for object files, except that an additional file with a **.dba** extension is generated to combine the debugging information for all of the members of the library.

The files **foo.dnm** and **foo.dla** are generated even if the **Debugging→Debugging Level** option is not set to **MULTI** (**-G**), so that MULTI can obtain skeletal debugging information for the executable. However, you must enable this option in order to perform source-level debugging and to view assembly language interlaced with your source.

Generating Debugging Information for Applications Compiled with Third-Party Compilers

Green Hills provides two Debug Translators that allow you to use the MULTI Debugger to debug applications compiled with third-party compilers. The MULTI Debugger provides superior debugging capabilities by using special debugging information files (***.dbo**) generated by Green Hills Software's compilers. If you do not compile with Green Hills tools, the Green Hills Debug Translators allow you to convert DWARF or Stabs debugging information generated by third-party compilers into the format required by the MULTI Debugger. If you need to use one of these translators, we recommend that you integrate them with your build process.

Note

The Debug Translator executables, **dwarf2dbo** and **stabs2dbo**, are separately licensed products of Green Hills Software. Contact your sales representative for information about obtaining and licensing these executables.

Translating DWARF and Stabs Debugging Information Automatically

The **dwarf2dbo** Debug Translator converts DWARF 1.0 and DWARF 2.0 debugging information into the special debugging information required by the MULTI

Debugger, and the **stabs2dbo** Debug Translator converts Stabs debugging information in the MULTI format.

If you have licensed either **dwarf2dbo** or **stabs2dbo**, you can configure MULTI to automatically convert DWARF or Stabs debugging information, respectively, when an executable compiled with a third-party compiler that generates the respective type of information is loaded into the Debugger. For information about enabling automatic conversion, see the documentation about using third-party tools with the MULTI Debugger in the *MULTI: Debugging* book.

Translating DWARF and Stabs Debugging Information Manually

You can also invoke the **dwarf2dbo** or **stabs2dbo** Debug Translator by passing the **-auto_translate** option to the **dblink** utility. For example, to translate an executable **a.out** that was compiled with either DWARF or Stabs information, issue the following command from a Linux/Solaris shell or a Windows command prompt:

```
dblink -auto_translate a.out
```

Note

If you compile some of the object files of your executable with a third-party compiler and others with a Green Hills Software compiler, you must pass a special driver option (**-dwarf_to_dbo** or **-stabs_to_dbo**) to the compiler driver during the final link phase. The general syntax for doing this is:

```
compiler_driver -debug_info_type_to_dbo link_options \
    object_files -o executable_name
```

where *debug_info_type* is `dwarf` or `stabs`.

For example, suppose you compile **legacy.c** and **old.c** with a third-party compiler that generates DWARF information. Suppose you then compile **ghs.c** and **new.c** with the Green Hills Software compiler, with the appropriate options set to generate MULTI debugging information for the object files. Suppose also that the executable runs on a V850 and RH850 target and that you want to generate a link map as well as the final executable **a.out**. From the command line, run the Green Hills Software compiler driver:

```
ccppc -dwarf_to_dbo -map legacy.o old.o ghs.o new.o -o a.out
```

Building and Debugging on Different Hosts

If you build programs on one host and debug them on another, and the sources exist in different locations on these different hosts, you can specify to the Debug Translator how the build host's paths correlate to the debug host's paths. Once **dwarf2dbo** knows about this remapping, MULTI is correctly able to locate your files.

To map the build host's paths to the debug host's paths, you can pass the **--remap** option to the Debug Translator. Note that you can also use the **sourceroot** command in MULTI if you want MULTI to dynamically change the file paths (see the documentation about **sourceroot** in the *MULTI: Debugging Command Reference* book). The advantages of using the **--remap** option are that the additional MULTI command need not be issued, and source scanning (see "Source Scanning" on page 53), when source file paths differ on the build host and debug host, can only be accomplished by using this option.

To enable remapping of source paths in the Debug Translator, create a **dwarf2dbo.xsw** or **stabs2dbo.xsw** file in the MULTI installation directory with the following contents.

```
--remap=build_host_path,debug_host_path
```

If you are translating debugging information manually (as described in "Translating DWARF and Stabs Debugging Information Manually" on page 52) you can also enable source scanning by passing:

```
-translator.args=--remap=build_host_path,debug_host_path
```

Only those file paths falling within `build_host_path` are modified. Multiple `--remap` options may be specified.

Source Scanning

The source scanning feature of the Green Hills Debug Translators can be used to improve the code browsing features of MULTI when using code compiled by third-party compilers. This feature causes the **dwarf2dbo** or **stabs2dbo** Debug Translator to gather information directly from source files. Because source scanning requires access to the entire source base of the program, it may be very slow if the source base resides on a remote file system and should thus only be utilized if the source files all reside on a local hard drive.

The cross-referencing and static call graph browsing functionality that results from source scanning is generally much more complete and accurate for C sources than for C++ sources, but is generally not as accurate as information generated by Green Hills compilers.

To enable source scanning, create a **dwarf2dbo.xsw** or **stabs2dbo.xsw** file in the MULTI installation directory with the following contents:

```
--scan_source
```

This causes source scanning to occur whenever debugging information is translated (either automatically or manually).

If you are translating debugging information manually (as described in "Translating DWARF and Stabs Debugging Information Manually" on page 52), you can also enable source scanning by passing:

```
--scan_source
```

to **dblink**.

Limitations

When you use MULTI to debug a program compiled by a third-party compiler, you cannot perform certain operations that are available for programs compiled by Green Hills compilers. The following limitations apply when you debug an application compiled by a non-Green-Hills compiler:

- Coverage analysis and call count profiling data is not available unless you use a trace-enabled target with a Green Hills SuperTrace Probe to collect trace data and then generate profiling data from the collected trace data. For more information, see the documentation about recording and viewing profiling data in the *MULTI: Debugging* book.
- Run-time error checking, including memory allocation checking, is not available.
- Host I/O features are generally not available. These features can be enabled for stand-alone programs by linking against certain objects in **libsys.a**, but this workaround is not fully supported by Green Hills Software.

- MULTI may not indicate when a variable has not yet been initialized or has a dead value.
- Command line procedure calls cannot be made unless the executable has been linked against **libmulti.a**. For more information, see "Enabling Command Line Procedure Calls" on page 56.
- Static call graph browsing is not available unless the program is translated using the source scanning feature (see "Source Scanning" on page 53).
- Depending upon the third-party compiler being used, advanced features may not be available in C++. Advanced features that may not be supported include using virtual command line procedure calls, viewing the most derived type of a pointer to a subclass, and viewing classes with virtual inheritance.
- Unless the source scanning feature is used (see "Source Scanning" on page 53), cross-referencing is not available for sources compiled by third-party compilers.
- Inspecting the value of preprocessor macros may not be possible. Source scanning (see "Source Scanning" on page 53) can be used to approximate the actual compile-time preprocessor macro values, but the exact values can only be known if the third-party compiler generates DWARF information with macro information. The Stabs debug information format encodes no macro information, so source scanning must be used if you are using a compiler that generates Stabs information and you want preprocessor macros debugging features.

Special Considerations for GNU Compilers

GNU compilers can generate DWARF 1.0, DWARF 2.0, or Stabs debugging information. Green Hills Software recommends generating DWARF 2.0 information rather than DWARF 1.0 or Stabs information. The standard **-g** GNU option may have been configured to generate any of these three formats, so the full option names are used here for clarification. To generate DWARF 2.0 information from the GNU compiler, enter:

```
gcc -gdwarf-2
```

If your GNU compiler cannot generate DWARF 2.0 information using the **-gdwarf-2** option, then generate Stabs information by entering:

```
gcc -gstabs+
```

If your GNU compiler cannot generate Stabs information using the **-gstabs+** option, then generate DWARF 1.0 information by entering:

```
gcc -gdwarf -g2
```

Enabling Command Line Procedure Calls

You can call procedures in your program from the Debugger command pane if you link your program with the library **libmulti.a**. **libmulti.a** must reside in RAM, not ROM, because MULTI must be able to set breakpoints in it.

When you set the Debugging Level to **MULTI** and then do a build, the Builder automatically links in **libmulti.a**. If you are not using the Project Manager, you must do one of the following to link with **libmulti.a**:

- Use the **-G** driver option.
- Use the **-lmulti** driver option.

Obtaining Profiling Information

The MULTI Debugger can analyze various forms of profiling information to help you make your program run more efficiently. This section describes how to compile your program so that profiling information is available to the Debugger and the memory requirements for each supported profiling method. For additional information, see the documentation about recording and viewing profiling data in the *MULTI: Debugging* book and "The protrans Utility" on page 614.

The following table outlines different types of profiling information, when you should collect them, and how they may impact the performance of your program. The impact of profiling varies depending on your program and the way you have configured your target:

Name	Use	Impact
Sampled	You want to find performance issues with the lowest possible overhead, and do not care about getting deterministic results. Sampled profiling requires that you run long enough to collect statistically significant results. It is not effective for code with a periodic execution pattern similar to or faster than the sampling rate. For more information, see "Enabling Sampled Profiling" on page 63.	Minimal
Instrumented Coverage	You want to find what code is executed when you run your program. See the next section.	Small
Instrumented Performance	You want to find performance issues with deterministic results. See the next section.	Medium
Instrumented Performance (64-bit)	You want to find performance issues with deterministic results, and some basic blocks execute a very large number of times. See the next section.	Large
Call Count	Deprecated. Use instrumented performance.	
Call Graph	Deprecated. Use instrumented performance.	

Profiling does not effectively measure programs that spend a lot of time doing things other than executing instructions, such as programs that spend a lot of time blocking on system calls, I/O, or other programs.

Enabling Instrumented Coverage or Performance Profiling

With *instrumented* profiling, the compiler adds instrumentation to the beginning of each basic block in a procedure to collect profiling information. There are two modes of instrumented coverage profiling:

- *Flag* profiling (**-coverage=flag**) records whether or not each block is executed. This mode is useful for determining code coverage.
- *Count* profiling (**-coverage=count**) records how many times each block is executed. This mode is useful for determining performance bottlenecks, but more expensive because it requires more RAM per block and executes more slowly.

These are compile-time options and can be selectively enabled on a subset of source files or on an entire program. For information about how to set the options, see "Profiling - Block Coverage" on page 156. You can also disable coverage profiling for specific functions by enclosing them in `#pragma ghs startnocoverage` and `#pragma ghs endnocoverage` directives.

Instrumented profiling does not use locking or an atomic increment to achieve thread safety. If you use count profiling with multi-threaded code or in interrupt handlers a block's counter may be inaccurate if two increments occur simultaneously. Flag profiling is thread-safe.

Coverage information is allocated in four sections:

`.ghcovfa`	**-coverage=flag** block addresses
`.ghcovfz`	**-coverage=flag** block flags
`.ghcovca`	**-coverage=count** block addresses
`.ghcovcz`	**-coverage=count** block counts

The block address sections are non-loading and should not appear in the section map of your linker directives (**.ld**) file. The flag and count sections must be zero-initialized at startup, like the `.bss` section. These sections can be listed in the section map, or the linker can define them. The linker defines the sections if they are not defined in the linker directives file and if coverage is enabled on the linker command line.

Instrumented Profiling Overhead

The following table describes the approximate performance overhead of instrumented profiling. The actual amount of overhead varies depending on your code and target:

Option	ROM Overhead	RAM Overhead
flag	15%	4%
count	30%	20%

If you have functions that do not need to be profiled and cause execution to take too long, you can disable profiling for those functions by surrounding their declarations with `#pragma ghs startnocoverage` and `#pragma ghs endnocoverage`. For more information, see "Green Hills Extension Pragma Directives" on page 753.

Instrumented Profiling Caveats

- Using any of the following features or code constructs may cause the compiler to generate code that is difficult or impossible to cover during execution of your program:
  - **-Ospeed**, when used with `strncpy()`
  - Run-Time error checking (**-check=**)
  - Loop unrolling (**-Ounroll**)
  - Switch statements that do not contain a default case
- Green Hills libraries are not instrumented for profiling information.

Enabling Function Entry/Exit Logging

For versions of INTEGRITY which support this feature, function entry/exit (FEE) logging keeps track of each entry into and exit from a function, storing the information into a log. Controlling function entry exit logging can be performed at compile time, link time, and run-time.

At compile time, in order to perform this logging, the compiler can generate instructions in the prologue and epilogue of each function. If FEE logging is being generated, it can be enabled, disabled, or stripped at link time. For more information, see "Profiling - Entry/Exit Logging" on page 157.

At link time, if a module has been compiled with FEE logging, the log points can be enabled, disabled, or stripped. If logging is enabled, then it is performed at run time by default. If logging is disabled, then logging is not performed at run-time by default, but can be enabled. If logging is stripped, then all logging code is removed from the module and it cannot be enabled at run-time. For more information, see “Profiling - Entry/Exit Linking” on page 158.

At run-time, if a function has been compiled with FEE logging and it has not been stripped at link time, the log points can be enabled or disabled by debuggers that support this feature. For information about how to enable and disable FEE logging at run-time, see the MULTI documentation.

This implementation allows you to compile your program with instrumentation, and easily disable or strip it away in the linker without having to recompile when you no longer need logging. To enable FEE logging on a function-by-function basis, use `#pragma ghs entry_exit_log` (see “Green Hills Extension Pragma Directives” on page 753) or `attribute((entry_exit_log))` (see `entry_exit_log` on page 678).

Enabling Call Count Profiling

Note

Call Count profiling is deprecated and will be removed in a future release.

Call Count profiling (**-p**) causes the compiler to add instrumentation code to the beginning of every procedure in your program. The program will usually have a larger `.text` section with this type of profiling enabled, but will have a smaller `.text` section than with legacy coverage profiling.

Call count profiling also dynamically allocates tables in heap memory proportional in size to the number of procedures actually executed while running your program. Call Count profiling uses an amount of heap memory equal to one pointer and one 32-bit integer counter (8 bytes on most systems) for each procedure in your program, plus some overhead for the linked lists it uses to hold the counters.

To obtain call count profiling information:

Set the **Advanced→Debugging Options→Profiling - Call Count** option to:

- **On** (**-p**), or
- **On with Call graph** (**-pg**).

If you try to use call count profiling and you get the error message "`mcount: could not allocate memory`" from the target, there is not enough heap memory available to use call count profiling. You can increase the size of the heap by changing the size of the `.heap` section in the linker directives file.

Call count profiling uses `__ghsLock()` and `__ghsUnlock()` in **libsys.a** to operate in a multithreaded environment. For more information about the profiling support routines, see “Customizing the Run-Time Environment Libraries and Object Modules” on page 813. The implementation of `__ghsLock()` and `__ghsUnlock()` in multithreaded environments such as INTEGRITY may preclude the use of this option for code that runs in certain contexts. Some call counts may be omitted in a multithreaded environment, and some counts may be omitted from interrupt service routines regardless of the environment.

Call count profiling is written to a file called **mon.out** when your program exits or when you issue the **profdump** command to the MULTI Debugger (see the documentation about manually dumping profiling data in the *MULTI: Debugging* book). You can usually find this file in the MULTI working directory, the directory of the simulator or debug server that you last used to run your program, or the same directory as your program, depending on your configuration.

Enabling Call Graph Profiling

Note

Call graph profiling is deprecated and will be removed in a future release.

Call graph profiling (**-pg**), also known as **gcount**, causes the compiler to add instrumentation code to the beginning of every procedure in your program. This type of profiling creates a larger `.text` section when enabled, but will have a smaller `.text` section than with legacy coverage profiling.

Call graph profiling requires an amount of heap memory that is, by default, between 1.5 and 2 times the size of the `.text` section. The memory is used to hold a hash

table and a pool of (`from_pc`, `to_pc`, `count`) arcs, which forms a call graph with weighted edges. This pool is large enough to store the call graphs of most programs.

If you see the following message from the target, your program has run out of available heap memory:

```
gcount: could not allocate n bytes from .heap
```

If you see the following message from the target, your program has a complicated call graph that cannot be represented with the arc pool:

```
gcount: not enough arcs to record call graph
```

You can fix this issue by customizing the low-level system library **libsys.a** (see "Customizing the Run-Time Environment Libraries and Object Modules" on page 813) and following the instructions contained in the comments of **ind_gprf.c**.

Call graph profiling is thread-tolerant, but some call counts may be omitted in a multithreaded environment, and some counts may be omitted from interrupt service routines regardless of the environment. For more information, see "Low-Level System Library libsys.a" on page 818.

Call graph profiling information is written to a file called **gmon.out** when your program exits or when you issue the **profdump** command to the MULTI Debugger (see the documentation about manually dumping profiling data in the *MULTI: Debugging* book). You can usually find this file in the MULTI working directory, the directory of the simulator or debug server that you last used to run your program, or the same directory as your program, depending on your configuration.

Enabling Sampled Profiling

This section describes how to obtain *sampled profiling* information for stand-alone and u-velOSity targets.

Sampled Profiling (**-timer_profile**) also known as **manprf**, is supported by **ind_manprf.c** in the source of **libsys.a**, and parts of the implementation might also be in the Board Initialization library **libboardinit.a** if your board contains that optional library. See "Board Initialization Library libboardinit.a" on page 823 for a description of this interface and the **board_info.txt** file in your project, if one exists. This form of profiling, which is generated by the target periodically sampling its own program counter, is faster and more accurate than host-driven profiling.

Target-based timing profiling requires a statically-allocated memory buffer, whose size is adjustable in **ind_manprf.c**, which is included in **libsys**. For information about how to include a customizable version of this file in your project, see "Creating a Project with a Customizable Run-Time Environment" on page 813. This buffer is used to hold the program counter samples before they are sent to MULTI; by default, it is large enough to hold 120 addresses. The buffer is statically allocated, so it will typically be allocated to `.bss`, not `.heap`. Rather than being written to a file by the target, target-based timing profiling information is intercepted by the debug server when the memory buffer fills.

The Target-Driven Timer Profiling library module, **ind_manprf.o**, is not linked into your program by default. To link with it and enable this form of profiling, follow these steps:

1. Right-click the program's project file and select **Configure**. In the dialog box that opens, select the **Board Initialization** check box (if present) and click **Finish**.
2. Review **ind_manprf.c** and (if present) your board's **board_info.txt** file for instructions. If your board is not explicitly supported, you must provide an implementation using the existing code as an example.
3. Rebuild the system library **libsys.a** and (if available for your board) the board initialization library **libboardinit.a** (see "Board Initialization Library libboardinit.a" on page 823 for a description of this interface).
4. To link with the **ind_manprf.o** module:

Set the **Debugging→Profiling - Target-Based Timing** option to **On** (**-timer_profile**).

Rebuild your application. If linker errors result, you may need to review the settings for your board in the Target-Driven Timing Profiling source files mentioned in the preceding.

5. Open your application in the debugger, enable profiling and run your program in the usual way. For more information about profiling, see the documentation about recording and viewing profiling data in the *MULTI: Debugging* book.

Limitations and Suggestions

Target-based timing profiling gathers stochastic samples of the target program's location at the time of periodic timer interrupts. For best results, let your program run for a long time to gather a large number of profiling data samples.

This method of gathering samples can generate misleading results if the sampling timer also causes other program actions, or if the program is dealing with external events at or near a multiple of the sampling timer's frequency.

Enabling Run-Time Error Checking

Run-time error checking instruments your code to help you find problems during execution. To enable run-time error checking:

Set the **Debugging→Run-Time Error Checks** option (**-check=**). For information about available run-time error checks, see “Run-Time Error Checks” on page 159.

In order to use run-time error checking, you must be able to single step your target without triggering an interrupt. If you are using a Green Hills Probe, see the documentation for the **step_ints** setting in the Green Hills Debug Probes User's Guide.

To perform run-time error checking without the MULTI Debugger, rebuild your application with run-time error checking enabled and run it normally. The program

reports errors and warnings to `stderr`. After each error, the program calls `exit` with a value greater than `0` , terminating itself. After 101 warnings are produced, it prints a message indicating too many errors and calls `exit` with a value greater than 0 , terminating itself.

To redirect error and warning output to your own handler:

1. Add the custom run-time error checking demo project to your Top Project (see the documentation about adding new items to your project in *MULTI: Managing Projects and Configuring the IDE*).
2. Scroll down the project tree and open **user_defined_handler.c** to see example code that sets up custom run-time error checking.

Enabling Run-Time Memory Checking

It can be very difficult to trace bugs involving the misuse of the standard library functions `malloc()` and `free()`, because the effects of these bugs may not be noticed immediately.

For example, if you mistakenly free memory that was not allocated with `malloc()`, or try to use memory that has already been freed, your process may behave in unpredictable ways.

MULTI supports two types of run-time memory checking to catch these kinds of bugs: *general memory allocation checking* and *intensive memory allocation checking*.

Enabling General Memory Allocation Checking

This form of run-time memory checking will increase your heap size considerably and reduce your process speed slightly. It requires less overhead than intensive memory allocation checking (see "Enabling Intensive Memory Allocation Checking" on page 66). It generates run-time errors when:

- Memory that has not been allocated with `malloc()` is subsequently freed
- The same area of memory is freed twice
- Recently freed memory is written to

In addition, general memory allocation checking allows you to trace *memory leaks*, which can occur when memory that is allocated with `malloc()` is not subsequently freed. In this situation, your process may unexpectedly run out of memory at some indeterminate future time. For more information about tracing memory leaks, see the documentation about viewing memory allocation information in the *MULTI: Debugging* book.

To enable general memory allocation checking, set the **Debugging→Run-Time Memory Checks** option to **General (Allocations Only)** (**-check=alloc**) before compiling your program.

Enabling Intensive Memory Allocation Checking

This form of run-time memory checking provides the most comprehensive `malloc()` checking. However, it will appreciably increase the size of your program code, and slow your execution speed more than general memory allocation checking (see "Enabling General Memory Allocation Checking" on page 65). It generates run-time errors when:

- Memory that has not been allocated with `malloc()` is subsequently freed
- The same area of memory is freed twice
- Recently freed memory is written to
- Attempts are made to dereference `NULL`
- Accesses are attempted past the end of a heap-allocated data structure, causing array overflow errors.

To enable intensive memory allocation checking, set the **Debugging→Run-Time Memory Checks** option to **Intensive** (**-check=memory**) before compiling your program.

Note

Intensive memory allocation checking may be impractical for users with memory constraints. You may want to enable general memory allocation checking for your entire project, and use intensive checking only for particular modules where `malloc()`-related errors are most likely to occur. Alternatively, you can manually change the intensity level of your memory checking with library functions, as described in the next section.

Using Memory Allocation Library Functions

The MULTI Compiler provides library functions that can control the amount and frequency of consistency checking performed by the `malloc()` library.

These functions are only available if you build your code with general memory allocation checking or intensive memory allocation checking enabled (see "Enabling General Memory Allocation Checking" on page 65 or "Enabling Intensive Memory Allocation Checking" on page 66). To call them, you must also include the **ghs_malloc.h** header file, located in ***install_dir*/ansi**, which declares the following two functions:

- `int malloc_set_debug_options(int newOptions)` — Sets the current intensity level and returns the old intensity level.
- `void malloc_check_lib(int useTheseOptions)` — Performs an immediate check of the `malloc()` library using the indicated intensity level.

Note

If you call `void malloc_check_lib()` with `0` or `M_CHECK_LEVEL_NONE`, the function uses the intensity level set when you last called `malloc_set_debug_options()`.

Both of the preceding functions take a constant that corresponds to an intensity level. The constants are defined in **ghs_malloc.h**. For a list of the available intensity levels, see "Performing Selective Run-Time Memory Checking" on page 67.

For a demonstration of the use of these memory checking functions, see Example 2.7. Enabling and Disabling All Checks on page 68.

Performing Selective Run-Time Memory Checking

As an alternative to enabling constant memory allocation checking, which may be impractical for users with memory constraints, you can enable certain memory checks by using the library functions described in "Using Memory Allocation Library Functions" on page 67.

Note

This requires that you build your code with general memory allocation checking or intensive memory allocation checking enabled. See "Using Memory Allocation Library Functions" on page 67.

The library functions take in an intensity level, which controls the amount of self-checking, and the frequency of that checking, that the memory library performs on calls to `malloc()`, `calloc()`, `realloc()`, and `free()`. The following list describes the intensity levels. For information about corresponding menu items, see the documentation about viewing memory allocation information in the *MULTI: Debugging* book.

- `M_CHECK_LEVEL_NONE` — Does not perform additional checking.
- `M_CHECK_LEVEL_MINIMAL` — Performs minimal checking on each API call.
- `M_CHECK_LEVEL_OCCASIONALLY_MAXIMAL` — Performs minimal checking on each API call, and occasionally performs full checking. This is the default setting.
- `M_CHECK_LEVEL_FREQUENTLY_MAXIMAL` — Performs minimal checking on each API call, and frequently performs full checking. Checks more frequently (and with more associated overhead) than `M_CHECK_LEVEL_OCCASIONALLY_MAXIMAL`.
- `M_CHECK_LEVEL_MAXIMAL` — Performs full checking on each API call. This may severely degrade the performance of your application.

Increasing the intensity level increases the amount and frequency of checking while also increasing the associated overhead and decreasing the speed of the application.

The default intensity level (`M_CHECK_LEVEL_OCCASIONALLY_MAXIMAL`) should be suitable for most applications. However, after you have narrowed down the area in which a memory error (corruption, for example) is occurring, it may be useful to increase the setting to `M_CHECK_LEVEL_MAXIMAL` shortly before the error is expected to occur.

Example 2.7. Enabling and Disabling All Checks

This example requires that you set the **Debugging→Run-Time Memory Checks** option to **General (Allocations Only)** (**-check=alloc**) before compiling your program.

Note that the following code is only meant as an example. Because of the errors included for demonstration purposes, it may not function exactly as documented if you copy it into your executable. Line numbers are added for clarity.

```
   #include <ghs_malloc.h>

   int main()
   {
5      int *arr;

       malloc_set_debug_options(M_CHECK_LEVEL_MAXIMAL); /* turn on all checks */
       arr = (int *)malloc(100*sizeof(int));
       free(arr);
10     arr[5] = 0; /* error - writing to a freed memory location. */

       /* it will check and find the last error now */
       arr = (int *)malloc(100*sizeof(int));
       free(arr);
15
       malloc_set_debug_options(M_CHECK_LEVEL_NONE); /* turn off all checks */


       arr = (int *)malloc(100*sizeof(int));
20     free(arr);
       arr[5] = 0; /* error - writing to a freed memory location. */

       malloc_check_lib(M_CHECK_LEVEL_MAXIMAL); /* check malloc data structure now;
                               detect error from line 21 */
25     return 0;
   }
```

This example does the following:

- `malloc_set_debug_options(M_CHECK_LEVEL_MAXIMAL)` on line 7 enables all the `malloc()` checks, so that the error on line 10 is detected when `malloc()` is called on line 13.
- `malloc_check_lib(M_CHECK_LEVEL_MAXIMAL)` on line 23 performs an immediate check on the `malloc()` data structure, and detects the error on line 21.

Using Your Own Header Files and Libraries

Note

For information about the headers and libraries provided with your distribution, see Chapter 16, “Libraries and Header Files” on page 767.

You might want to create your own libraries of frequently used functions and instruct the compiler to link against them. While the compiler knows where to find the standard libraries and header files provided with your distribution and links against them, you must tell it where it should search for your own headers and libraries. At link-time, the linker will attempt to resolve external references in the source files you pass to it by linking against your libraries before searching the standard libraries.

Instructing the Compiler to Search for Your Headers

To instruct the compiler to look for header files in a new directory:

Enter the directory in the **Project→Include Directories** option (**-I*directory***).

You can specify multiple directories. The compiler searches the directories in the order in which they are specified.

Using Two Forms of the Include Directive

The compiler recognizes both standard forms of the `#include` directive (that is, `#include <`*`header`*`>` or `#include "`*`header`*`"`). These forms can specify either a full path or only a filename. The full range of possible types of `#include` directives are identified in the following:

	Linux/Solaris host	Windows host
A	`#include "/src/h/`*`header`*`.h"`	`#include "f:\src\h\`*`header`*`.h"`
B	`#include </src/h/`*`header`*`.h>`	`#include <f:\src\h\`*`header`*`.h>`
C	`#include "`*`header`*`.h"`	`#include "`*`header`*`.h"`
D	`#include <`*`header`*`.h>`	`#include <`*`header`*`.h>`

The compiler searches for header files included with the various types of `#include` directive in the following ways:

- If the `#include` directive specifies a full path to the header file (cases A and B above), the compiler searches in that location only. It does not search for another version of the file in any other directory.
- If the `#include` directive specifies only a filename or a partial path for the header file, and uses quotes (case C above), the compiler searches in the following directories, in order:
  1. The directory of the source file that contains the `#include` directive.
  2. Directories specified with the **Project→Include Directories** option (**-I*directory***), in the order in which they appear on the command line.
  3. The default directories (see "C and C++ Header File Directories" on page 769).
- If the `#include` directive specifies only a filename or a partial path for the header file, and uses angle brackets (case D above), the compiler searches in the following directories, in order:
  1. Directories specified with the **Project→Include Directories** option (**-I*directory***), in the order in which they appear on the command line, and then
  2. The default directories (see "C and C++ Header File Directories" on page 769).

Modifying Header File Searches with the -I- Option

Specifying a dash (-) in the **Project→Include Directories** list (**-I-**) changes the rules for searching for header files where only a filename or partial path is given (cases C and D):

- For the directive `#include "header.h"` (case C in the previous section), the compiler searches in the following directories, in order:
  1. Directories specified with `-I`*`directory`*, in the order in which they appear on the command line.
  2. The default directories.

  The compiler does not search in the directory of the source file.

- For the directive `#include <header.h>` (case D in the previous section), the compiler searches in the following directories, in order:
  1. Directories specified with `-Idirectory` that appear on the command line after the **-I-** option.
  2. The default directories.

For example, if the command line includes:

```
-Iquote -I- -Iangle
```

Then `#include "foo.h"` would refer to the first matching file of **quote/foo.h** or **angle/foo.h**, but `#include <foo.h>` could refer only to **angle/foo.h**.

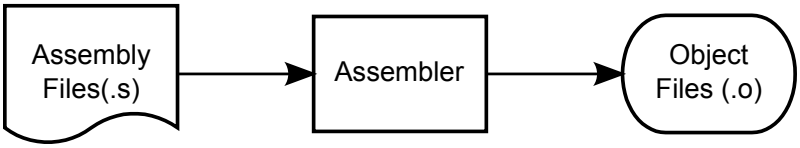
Instructing the Compiler to Link Against Your Libraries

To instruct the compiler to link against your libraries:

- Specify the directories to search in the **Project→Library Directories** option (**-L*library_directory***).
- Specify the library names to link against in the **Project→Libraries** option (**-l*library***). Note that the names of libraries must be specified in their abbreviated form. For example, a library called **libfoo.a** would be specified as **foo**.

Controlling the Assembler

Unless you instruct otherwise, the Builder or driver invokes the assembler, **ease850**, to process assembly output of the compiler when necessary. In most cases, however, the compiler generates binary directly. For more information, see "Binary Code Generation" on page 251.



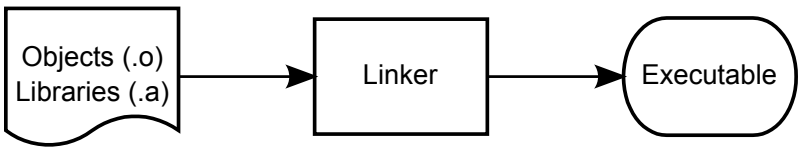
You can control the most commonly used features of the assembler with Builder and driver options. For example, to instruct the assembler to produce a source listing file:

Specify a filename in the **Assembler→Source Listing Generation** option (**-list[=*file*]**).

For a list of all the assembler-specific Builder and driver options, see "Assembler Options" on page 216. For detailed documentation of the assembler, see Chapter 6, "The ease850 Assembler" on page 361.

Controlling the Linker

Unless you instruct otherwise, the Builder or driver automatically invokes the linker, **elxr**, to combine the object files.



You can control the most commonly used features of the linker with Builder and driver options. For example, to delete unused functions:

Set the **Linker→Linker Optimizations→Deletion of Unused Functions** option to **On** (**-delete**).

For a list of all the linker-specific Builder and driver options, see "Linker Options" on page 219. For detailed documentation of the linker, see Chapter 9, "The elxr Linker" on page 435.

Working with Linker Directives Files

A linker directives file has a **.ld** extension and controls how the linker links your program and prepares program sections for allocation into specific regions of memory on your target. A typical linker directives file contains:

- A *memory map* — Lists the starting address and size of each region of memory that is used to contain program code.
- A *section map* — Lists the program sections and specifies the memory regions that they will be loaded into.

Linker directives files can also define constants and special symbols that the linker uses when linking and loading your executable. For more information about the file format, see "Configuring the Linker with Linker Directives Files" on page 441.

One or more default linker directives files are generated automatically when you create a new project for the **Stand-alone** operating system and are placed within the Target Resources project. Depending on the **Program Layout** setting you choose

in the **Project Wizard** (see "Assigning Program Sections to ROM and RAM" on page 81), one of these files is added to your program project file for use at link-time.

Note

These default files are automatically generated by the **Project Wizard** by combining the generic section maps in the ***install_dir*/target/v800** directory with a board-specific memory map taken from ***install_dir*/target/v800/*target_dir***.

You may need to edit these default files to add new memory regions or sections, or to change their attributes. To edit a file, double-click it (or right-click it and select **Edit**) to open it in a text editor.

Text and Data Placement

The compiler assigns all text and data to *program sections*, which are named collections of objects. During the link step, the linker collects all data for each program section and allocates it in memory, guided by a section map defined in your linker directives (**.ld**) file. The section map specifies the location of each program section in the final output file. Normally, the compiler and linker use a default set of program sections, but you can also define custom program sections, assign specific data to them, and assign them to specific memory blocks on your target.

The following documentation explains how the compiler assigns data to default program sections and how to create and use custom sections. It goes on to cover additional topics that explain how to manipulate the placement of text and data (such as in registers or special data areas), and how to access it.

Default Program Sections

The default program sections for text and data (except local automatic variables) are:

- `.text` — which holds program code.
- `.data` — which holds variables with explicitly initialized values (e.g. `int i=1;`).
- `.bss` — which holds variables that are not explicitly initialized and that are not COMMON variables (for example, `static int j;`). The Green Hills startup code initializes this section to all zeros, as required by the C and C++ languages.
- `.rodata` — which holds some read-only data, such as `const` variables. This data varies depending on your target.
- `COMMON` — which holds global variables in C that are not explicitly initialized and whose modules were not compiled with the **--no_commons** option (see “Allocation of Uninitialized Global Variables” on page 201). When linking an executable, the linker reassigns each variable in this section based on whether or not it is explicitly initialized in any of the object files. By default, the linker reassigns uninitialized global variables in this section to the `.bss` section.

- If the *Small Data Area* (SDA) optimization is enabled (see "Special Data Area Optimizations" on page 87), the compiler re-allocates some data from the primary sections to the equivalent SDA sections:
  - `.sdata`
  - `.sbss`
  - `.rosdata`
  - `SMALLCOMMON`
- If the *Zero Data Area* (ZDA) optimization is enabled (see "Special Data Area Optimizations" on page 87), the compiler re-allocates some data items to the equivalent ZDA sections:
  - `.zdata`
  - `.zbss`
  - `.rozdata`
  - `ZEROCOMMON`

Note

The placement of data depends on compile-time options.

Custom Program Sections

In addition to the default program sections, you can define your own custom program sections. Custom program sections allow you to group certain variables or functions in specific memory blocks on your target, using the following technique:

1. In your source code, assign text or data into a custom program section.
2. In your linker directives file, assign the custom section to a memory block on your target. If you want to initialize the data in a section to zero (as with a `.bss` section), use the `CLEAR` section attribute. For more information, see "Defining a Section Map with the SECTIONS Directive" on page 447.

For example, you might assign some variables that are accessed very often to a custom program section, and in your linker directives file, assign that section to a small block of very fast RAM. The following section explains how to assign text and data into a custom program section.

Assigning Data to Custom Program Sections in C

To assign data to a custom program section in C code, use the `#pragma ghs section` directive. This directive takes variables and functions that the linker would otherwise assign to the specified default section (such as `.data` or `.bss`) and assigns them to a custom section instead. In other words, it maps a type of data to a custom section. Each time you use the directive, it leaves mappings from earlier occurrences in place, except for those that it explicitly overrides. This directive must be placed before the definitions of the objects you want to assign to the custom section; placing it before a declaration has no effect. It should not be used inside of any function.

The syntax for the `#pragma ghs section` directive is:

`#pragma ghs section [`*`secttype`*`="`*`sectname`*`"]`

- *secttype* — A text string that represents the default section. Valid values for *secttype* are listed in the following table.
- *sectname* — The name of the custom section to which you want to re-map text or data, including the initial period character (`.`). If you specify the word `default` without quotes, it removes the mapping and restores the assignment rule to its initial state.
- If you do not specify *secttype*="*sectname*", all mappings are removed and section assignment rules are restored to their initial state.

***secttype* Value**	**Default Program Section**
`bss`	`.bss`
`data`	`.data`
`text`	`.text`
`rodata`	`.rodata`
`sbss`	`.sbss`
`sdata`	`.sdata`
`rosdata`	`.rosdata`
`zbss`	`.zbss`
`zdata`	`.zdata`
`rozdata`	`.rozdata`

Note

For information about assigning objects to special data areas such as SDA, see “Special Data Area Optimizations” on page 87.

To change the compiler's default section mappings, see “Rename Section” on page 131.

Example 2.8. Assigning Variables to Different Sections

This example assigns three different variables to three different sections:

```
/* Assign x1 to section .data1 */
#pragma ghs section data=".data1"
int x1 = 1;

/* Assign x2 to section .data2 */
#pragma ghs section data=".data2"
int x2 = 2;

/* Assign x3 to section .data3 */
#pragma ghs section data=".data3"
int x3 = 3;

/* Now go back to default rules */
#pragma ghs section data=default
```

Assigning Data to Custom Program Sections in Assembly

To assign data to a custom program section in assembly, use the `.section` directive. When defining a custom section, you must also specify the appropriate attributes:

- `a` — allocated to memory
- `w` — writable

For example, to define a section `.mydata` that will be laid out in RAM and assign the variable `foo` to that section, use the following assembly code:

```
    .section ".mydata","aw"
foo:
```

For a complete list of attributes for the `.section` directive, see "Section Control Directives" on page 389.

Reserved Section Names

The Green Hills run-time environment uses reserved program sections that are defined in default linker directives files. The run-time object files **libsys.a**, **libstartup.a**, and **crt0.o** use them to set up the run-time environment. If you are creating custom sections, do not duplicate the names of these reserved sections. For more information about them, see "Customizing the Run-Time Environment Program Sections" on page 464.

Assigning Program Sections to ROM and RAM

In embedded programming, it is generally the case that all program sections must be initially located in ROM to prevent their loss upon reset, and that any program sections containing data that may change at run-time must be copied into RAM at startup. You must decide how much of your program code will be copied into RAM at startup. Two factors that influence this decision are the amount of available RAM and the emphasis on speed of execution. In general:

- Programs that copy all of their sections from ROM to RAM at startup execute more quickly, but require more RAM.
- Programs that copy as few sections as possible from ROM to RAM at startup require less RAM, but execute more slowly.

Green Hills provides default linker directives files to manage the location of program sections. To specify one of them when creating a project with the **Project Wizard** (see the documentation about creating a project in the *MULTI: Managing Projects and Configuring the IDE* book), set the **Program Layout** option to one of the following settings:

- **Link to and execute out of RAM** (**-layout=ram**) — [default] The program is linked so that it is loaded into, and executes out of RAM. Such programs are usually downloaded and started by running the Debugger and a debug server. This layout is generally used in the early stages of development. It appears in the Target Resources project as **standalone_ram.ld**.
- **Link to and execute out of ROM** (**-layout=romrun**) — The program is linked so that it can be loaded into, and executes out of ROM/FLASH. Such programs must be burned into flash (for example, by using MULTI Fast Flash Programmer). This layout may be used for producing a final executable. It requires the least amount of RAM, but is typically slower, and cannot be debugged using software breakpoints. Some features that rely on software breakpoints may not be available in this mode, including system calls, command line procedure calls, memory leak detection, and run-time error checking. It appears in the Target Resources project as **standaline_romrun.ld**. For more information about the MULTI Fast Flash Programmer, see the documentation about programming flash memory in the *MULTI: Debugging* book.
- **Link to ROM and execute out of RAM** (**-layout=romcopy**) — The program is linked so that it can be loaded into ROM/FLASH, but copies itself to RAM

and executes out of RAM. Such programs must be burned into flash. This layout can be used for producing a final executable. It is faster than the `romrun` layout and offers software breakpoint debugging, but takes longer to initialize and requires more RAM. It appears in the Target Resources project as **standalone_romcopy.ld**.

- **Link for Position Independent Code** — A special linker directives file for projects built with *Position Independent Code* (see "Position Independent Code (PIC)" on page 106). It appears in the Target Resources project as **standalone_pic.ld**.
- **Link for Position Independent Data** — A special linker directives file for projects built with *Position Independent Data* (see "Position Independent Data (PID)" on page 108). It appears in the Target Resources project as **standalone_pid.ld**.
- **Link for Position Independent Code and Data** — A special linker directives file for projects with *Position Independent Code* and *Data*. It appears in the Target Resources project as **standalone_picpid.ld**.

Whichever layout you choose when creating your project, the **Project Wizard** puts all linker directives files in the Target Resources project. To change your program layout, right-click your program's project and select **Configure**.

These linker directives files provide a basic section map. If you want to use custom sections and control precisely where text and data are located, edit your linker directives file using the information in "Configuring the Linker with Linker Directives Files" on page 441. For further discussion about optimizing your program by editing your linker directives file, see "Modifying your Section Map for Speed or Size" on page 463.

If you are using the driver with the **-layout=** option, you must also specify the **-bsp=** option (see "Specifying a V850 and RH850 Target" on page 46). If no appropriate BSP is available, specify **-bsp=generic**. The **-layout=** option is ignored if you specify your own linker directives files.

Storing Global Variables in Registers

Machine registers are primarily used by the compiler to store local variables or working values. This generally produces smaller and faster code than if most variables were on the stack. In certain circumstances, however, you can also allocate critical global data to registers (though this may decrease performance). Unless you are building INTEGRITY code, you can reserve up to 5 global registers (`r20` to `r24` in that order). To do so:

Specify the number of registers you want to reserve with the **Target→Text and Data Placement→Global Registers** (**-globalreg=*n***) option.

Note

This option may not work correctly if you customize your startup code. If you are building INTEGRITY code, you cannot modify the default value of this option.

Note

The registers reserved will change if certain Builder or driver options are passed:

The following Builder or driver options restrict which registers may be reserved:

- **-r20has255**: Only `r21-r24` are available.
- **-r21has65535**: Only `r22-r24` are available.
- **-registermode=22**: No registers are available. The **register** keyword is accepted but ignored.
- **-registermode=26**: Only `r23` and `r24` are available.

The Green Hills library startup code (“Startup Module crt0.o” on page 815) initializes all permanent registers as if the register mode were set to 32. Additionally, the Green Hills library `setjmp` and `longjmp` implementations save and restores all permanent registers as if the register mode were set to 32.

Setting this option allows the (otherwise illegal) C/C++ `register` keyword to place global variables in registers using the following syntax:

```
register int i; /* file scope implies global */
```

Note

The syntax `register static int i` is still illegal, because it can cause a possible register numbering problem when compiling multiple files.

The data type of the global variable can be any integral or pointer type where a local variable of that type would be eligible for allocation to a single register. It must not be explicitly initialized, and its address should never be taken. The program must be compiled with the Green Hills startup code.

Global registers are permanent registers, so functions compiled with this option can call (but not be called by) functions compiled without it, including third-party object files and libraries. A function cannot access variables allocated to global registers unless all functions in the call chain before it were compiled with the global registers reserved. The easiest way to guarantee this is to compile all files with this option. The Green Hills libraries are compiled with appropriate options to allow linking with programs that use global registers. It is also generally not safe to use variables allocated to global registers in interrupt functions because such interrupts might be triggered asynchronously.

If you want to share the same global registers across compilations, consider placing them into a single `#include` file, which is always included first in all compilations to ensure consistency.

Near and Far Function Calls

A function call is normally performed with an instruction that contains a 22-bit value that is added to the address of the current instruction to yield the address of the called function. If the offset to the called function is greater than 22 bits, a direct call instruction will be out of range.

To avoid this problem, the compiler provides support for *far function calls*. The compiler generates a series of instructions to place the address of the called function into a register before the function call. This method allows you to reach a function

anywhere in the address space of the processor, but it requires additional code size and processor time.

Call Patching

By default, the linker handles far function calls using *call patching*. Call patching causes the linker to automatically detect function calls that are out-of-range and insert code between modules to resolve the far function call. The linker does this by creating a small patch of code, called a call patch, to perform a jump to the out-of-range function. It then changes the destination of the original function call to point to the call patch. This method impacts code size and run time less than using far function calls because it affects only those calls that are out-of-range. The one limitation of call patching is that the module containing the offending function call must be 2 MB or smaller; otherwise, the linker will not be able to place the call patch close enough to the function call.

Sometimes, you may want to rearrange code manually instead of using call patching. To disable call patching:

Set the **Target→Text and Data Placement→Linker-Based Far Call Patching** option to **Off** (**-nofarcallpatch**).

Far Function Call Build Option

To enable far function calls for every function:

Set the **Target→Text and Data Placement→Far Calls** option to **Make All Calls Far Calls** (**-farcalls**).

This option instructs the compiler to generate a far function call for every call (with no need for modifications to the source code), unless the option is overridden by one of the other methods specified below. This method should only be used when far call patching is not an effective solution.

Function Call Pragma Directives

The following `#pragma` directives govern the declaration of functions. When you use one of these directives, all function declarations after it will be modified until another of these `#pragma` directives occurs.

`#pragma ghs near`
`#pragma ghs callmode=near`
Any function declared after this directive will be called using a near function call.

`#pragma ghs callmode=callt` Any function declared after this directive will be called using the `callt` instruction and will return using the `ctret` instruction. Users should take caution to ensure that both the declaration (prototype) and definition (where the routine is written) of the routine are declared with this pragma, to ensure calls to the routine are done correctly, and to ensure the function return from the routine is done correctly.

`#pragma ghs far`
`#pragma ghs callmode=far`
Any function declared after this directive will be called using a far function call.

`#pragma ghs callmode=default` Any function declared after this directive will be called as directed by the **Target→Text and Data Placement→Far Calls** builder option (see "Text and Data Placement" on page 125).

Note that in certain cases, such as when both the caller and callee are located in the same file, the compiler may be able to determine that a far function call is unnecessary, and so perform a near call regardless of the `#pragma` directive.

Function Declaration

The keywords `__nearcall` and `__farcall` and `__callt` may be placed in the declaration of a function to specify the type of function call to be used for that function. These keywords override the setting implied by either the `#pragma` directive or the command line option. For example:

```
__nearcall int nearfunc();
__farcall int farfunc();
__callt int tablefunc();
```

The syntax requires that `__nearcall` and `__farcall` and `__callt` be placed before the function return type.

The `__nearcall` and `__farcall` and `__callt` keywords cannot be used for function pointers, but only for functions themselves. They cannot be used in `typedef` statements, casts, or in the type of a function parameter. Furthermore, functions declared with the `__callt` keyword cannot have their addresses taken.

Note

Users should take caution to ensure that both the declaration (prototype) and definition (where the routine is written) of the routine are declared with this `__callt` keyword, to ensure calls to the routine are done correctly, and to ensure the function return from the routine is done correctly.

Special Data Area Optimizations

Special data area optimizations place certain data in a special memory block that can be accessed by using an offset from a base address. In most cases, data in this block can be accessed more efficiently than other data. This optimization does not change the basic function of your program.

V850 and RH850 supports *Small Data Area* (SDA) optimization and a specialized version of SDA called *Zero Data Area* (ZDA) optimization. This section describes these optimizations and explains how to allocate data to special data areas.

Small Data Area (SDA) Optimization

Small Data Area (SDA) optimization uses two special data areas, RAM SDA and ROM SDA, each containing 64 KB of data.

In general, the combined size of data segments that are assigned to SDA cannot exceed 128 KB. By convention, the Green Hills compiler uses register `r4` as the base register for the RAM SDA, and register `r5` as the base register for the ROM SDA. Because offset-based addressing on the V850 and RH850 processor is signed, an SDA base register points 32 KB past the start of the SDA to provide a full 64 KB of addressing.

The V850E2V3 and later processors can have the size of the RAM SDA and ROM SDA extended to 8 MB using the **-large_sda** option; see “8 MB Special Data Areas on the V850E2V3” on page 95 for more details.

In order to implement SDA optimization, the compiler, assembler, and linker define the following special program sections:

- `.sdata` — initialized read/write data
- `.sbss` — uninitialized read/write data
- `.rosdata` — constant data

Zero Data Area (ZDA) Optimization

Zero Data Area (ZDA) optimization is a special case of SDA that uses register `r0`, which is hardwired to the value zero, to access memory around address zero. Certain variables and data structures can then be accessed using offsets from `r0` rather than full 32-bit addresses. ZDA is not supported with Position Independent Data (PID).

Due to the limitations on the offset-based addressing used to access ZDA variables, the size of the ZDA is limited to a total of 64 KB.

The V850E2V3 and later processors can have the size of the ZDA extended to 8 MB using the **-large_zda** option; see "8 MB Special Data Areas on the V850E2V3" on page 95 for more details.

To implement ZDA optimization, the compiler, assembler, and linker define the following special program sections:

- `.zdata` — initialized data
- `.zbss` — uninitialized data
- `.rozdata` — constant data

Assigning Data to Special Data Areas Using a Threshold

You can assign a threshold value (as an integer in bytes) so that the compiler assigns variables of that size or smaller to a special data area.

To specify an SDA threshold value:

Set the **Target→Text and Data Placement→Special Data Area** option to **Small Data Area** (**-sda[=*threshold_value*]**)

To specify a ZDA threshold value:

Set the **Target→Text and Data Placement→Special Data Area** option to **Zero Data Area** (**-zda[=*threshold_value*]**)

Where *threshold_value* must be one of the following:

- `none` — does not assign any data to the special data area using a threshold. If specified with **-zda**, this setting disables the `#pragma ghs startzda` and `#pragma ghs endzda` directives. If specified with **-sda**, this setting does not disable any `#pragma` directives.
- `0` — does not assign any data to special data areas using thresholds, but does not disable any `#pragma` directives. Use this option when you want to assign variables to special data areas with `#pragma` directives only.
- *n* — assigns all variables up to *n* bytes in size to the special data area. *n* must be a positive integer.
- `all` — assigns all variables to the special data area. While there is no size limitation, this option may cause the special data area to overflow.
- `never` — similar to `none`, with two additional features: 1) Unless it is reserved to be the PID base by `-pid`, the SDA base register `r4` is available to the compiler for other uses. 2) The directive `#pragma ghs startsda` and `#pragma ghs endsda` are disabled in the compiler.

-sda and **-zda** are mutually exclusive options; if you enable both of them on the command line, the last option takes effect

Note

-sda=never should only be used if all source files in the program are compiled with **-sda=never** or they contain no SDA `#pragma` directives and are compiled with **-sda=none** or **-sda=0**.

Note

-sda=never is only supported on the RH850 and newer processors.

Assigning Specific Data To Special Data Areas in C

When programming in a high-level language, you can assign individual variables to a special data area by enclosing them in a set of `#pragma` directives. These directives are:

- For SDA — `#pragma ghs startsda` / `#pragma ghs endsda`
- For ZDA — `#pragma ghs startzda` / `#pragma ghs endzda`

The `startsda` and `startzda` directives cause the compiler to allocate all subsequent variables to the special data area, even if they exceed a specified threshold value (see "Assigning Data to Special Data Areas Using a Threshold" on page 88). An `endsda` or `endzda` directive ends this mode.

If a variable is declared between these `#pragma` directives and then declared again outside them, the compiler assigns the variable to the special data area.

If a variable is declared `extern` between special data area `#pragma` directives and later in the same module is defined as a global variable without them, it still goes into the special data area.

Note

If you use these directives for ZDA and set the option **-zda=none**, the compiler issues a warning.

If you want only those variables explicitly specified by these `#pragma` directives to be allocated to the special data area, set the threshold value to `0` (see "Assigning Data to Special Data Areas Using a Threshold" on page 88).

Excluding Specific Data From Special Data Areas in C

To exclude data from a special data area even if it is smaller than or equal to the specified threshold value, enclose the variable declarations in a `#pragma ghs startdata` / `#pragma ghs enddata` block. Variables defined within this block are never assigned to a special data area.

For example, if you set the SDA threshold value to `8`:

```
int foo_a = 4;          /* goes into SDA */
#pragma ghs startdata
int foo_b = 4;          /* does not go into SDA */
#pragma ghs enddata
```

Assigning Specific Data To Special Data Areas in Assembly

When programming in assembly language, assign data to a special data area using the `.section` assembler directive. You must assign the appropriate attributes to that section.

For example, to include the variable `sda_var` in the `.sdata` section, enter the following in your assembly file:

```
.section ".sdata","aw"
sda_var:
```

To include the variable `zda_var` in the `.zdata` section of ZDA, enter the following in your assembly file:

```
.section ".zdata","aw"
zda_var:
```

ROM SDA sections only require the `a` attribute, because they are not writable.

For a complete list of attributes for the `.section` directive, see "Section Control Directives" on page 389.

Editing Linker Directives Files for Use with SDA

The linker directives (**.ld**) file for a program that uses the SDA optimization must list SDA-related sections in a specific order. In particular, custom sections must be listed after the appropriate RAM SDA or ROM SDA start directive. The Green Hills tools define the base register(s) to point to the middle of the memory region enclosed by the appropriate start and end symbols for the SDA region. For example, the RAM SDA base register will be initialized to the middle of the memory region enclosed by the `.sda_start` and `.sda_end` sections, while the ROM SDA base

register will be initialized to the middle of the memory region enclosed by the `.rosda_start` and `.rosda_end` sections.

RAM SDA sections must be listed in the following order:

- Empty section used by Green Hills debug servers and startup code (`.sda_start`)
- Default SDA section for initialized data (`.sdata`)
- Custom SDA sections for initialized data, if any
- Default SDA section for zero-initialized data (`.sbss`)
- Custom SDA sections for zero-initialized data, if any
- Empty section used by Green Hills debug servers and startup code (`.sda_end`)

ROM SDA sections must be listed in the following order:

- Empty section used by Green Hills debug servers and startup code (`.rosda_start`)
- Default ROM SDA section (`.rosdata`)
- Custom ROM SDA sections, if they exist
- Empty section used by Green Hills debug servers and startup code (`.rosda_end`)

Example 2.9. Linker Directives File for an SDA-Optimized Program

The following is an example of a linker directives file for an SDA-optimized program that has a user-defined SDA section, `.mysdata`.

```
SECTIONS
{
    .text 0x10000 :
    .syscall :
    .secinfo :
    .rodata :
    .ROM.data    ROM(.data) :
    .ROM.sdata   ROM(.sdata) :
    .ROM.mysdata ROM(.mysdata) :
    .sda_start align(8) :
    .sdata :
    .mysdata :
```

```
    .sbss :
    .sda_end :
    .data :
    .bss :
    .heap align(16) pad(0x100000) :
    .stack align(16) pad(0x80000) :
}
```

Editing Linker Directives Files for Use with ZDA

You must specify the placement of the ZDA in a linker directives (**.ld**) file. The linker directives file will vary depending on the availability of ROM and RAM near location zero. For example, if RAM is available from address `0x0` through address `0x1000`, the following entries might be included in the linker directives file:

```
.zdata 0x0 :
.zbss max_endaddress(0x1000)
```

Assigning Data to Custom Special Data Area Sections

To assign variables to a custom special data area section, use `#pragma ghs section` within a block of C code. The following example assigns SDA variables to both the default SDA section `.sdata` and a custom SDA section `.mysdata`:

```
#pragma ghs startsda
/* puts variables into default SDA section */
int e = 3;
double f = 15.2;
#pragma ghs section sdata=".mysdata"
/* puts variables into .mysdata user-defined SDA section */
int g = 10;
short h = 3;
#pragma ghs endsda
```

For more information about `#pragma ghs section`, see “Assigning Data to Custom Program Sections in C” on page 78.

Linking Special Data Area Modules

Program source modules must agree about which variables are assigned to special data areas. While using just a threshold value avoids this problem, you must pay special attention to it when using `#pragma` directives to assign specific data. You may want to put all special data area `#pragma` directives and variable declarations into header files that are included by modules that use those variables.

In most cases, you can safely combine modules built with the SDA optimization with non-SDA modules. Since the ZDA base register `r0` is always available for ZDA, modules built with the ZDA optimization can also combine safely with non-ZDA modules. However, problems may arise when one of these special data area modules uses a global variable that is defined in a module that does not use the same special data area. If the module attempts to use base offset addressing to reference the variable, a link-time error can occur because the variable is not actually in a special data area. To correct this problem, recompile the modules with consistent use of special data areas.

Green Hills libraries are built with the SDA base register reserved. This allows both SDA and non-SDA modules to use the same libraries.

Note

If a variable is assigned to SDA but is out of range of the SDA base register, but is within range of the ROM SDA base register (if ROM SDA is enabled) or is within the ZDA region, references to this variable may be adjusted by the linker to use the ROM SDA or ZDA base register as appropriate. Likewise if a variable is assigned to ROM SDA but is out of range of the ROM SDA base register, its references may be adjusted by the linker to be relative to the SDA or ZDA base registers. This behavior can be disabled with the **-no_xda_modifications** linker option. However, this behavior is necessary to link some C++ programs without generating a linker error error.

Predefined Linker Symbols for Initializing the SDA Base Registers

User startup code (**crt0.o**, see “Startup Module crt0.o” on page 815) can reference the special C symbols `_tp` and `_gp` for setting the appropriate initial values of the `tp` and `gp` registers respectively. These symbols are defined in **v800_ghs.h**, and are set at link time.

Linker Errors Related to Special Data Areas

The amount of data assigned to a special data area cannot exceed the size of that data area. If data overflows, the linker issues an error. For SDA, the message begins with:

```
[elxr] (error) out of range: offset (signed) didn't fit in n bits
```

ZDA provides a similar error message.

If you want to eliminate these errors by increasing the size of the special data areas, see “8 MB Special Data Areas on the V850E2V3” on page 95. Another possible cause of these link errors is inconsistent use of the `#pragma` directives between modules. See “Assigning Specific Data To Special Data Areas in C” on page 90 for information about these directives.

In certain cases, the assembler is able to determine that the offset does not fit into 16 bits, and it prints a `value too large for word sized cast` error. This error is likely an indicator that too much data has been allocated to the ZDA section. Use the methods described in the previous sections to ensure that the end of the ZDA section is no more than 32 KB from address zero.

In the event that a module compiled with TDA disabled is linked with a module that uses TDA, the linker will give one of the following errors as appropriate:

```
Module FILENAME was compiled with TDA disabled but we are
linking in MTDA mode

Attempt to link module (FILENAME) compiled with -notda
against a module (FILENAME) that uses the TDA
```

8 MB Special Data Areas on the V850E2V3

On the V850E2V3 and later CPU variants, the Green Hills tools support larger special data areas containing 8 MB of data. To use this larger data area for SDA:

Set the **Target→Text and Data Placement→23-bit SDA (V850E2V3 and later)** to **On** (**-large_sda**). When you enable this option, the SDA can be accessed using 6-byte load and store instructions with a 23-bit offset from the SDA base register.

To use this larger data area for ZDA:

Set the **Target→Text and Data Placement→23-bit ZDA (V850E2V3 and later)** to **On** (**-large_sda**). When you enable this option, the ZDA can be accessed using 6-byte load and store instructions with a 23-bit offset from the ZDA base register.

If you enable either of these options, we recommend that you also:

Set the **Target→Text and Data Placement→Linker-Based SDA23/ZDA23 Shortening (V850E2V3 and later)** option to **On** (**-shorten_loads**). This option converts 6-byte load and store instructions to 4-byte instructions when possible. When you enable this option in conjunction with either of the previous options, you can benefit from a larger SDA without dramatically increasing code size.

V850 Tiny Data Area (TDA) Optimization

The V850 processor provides on-chip SRAM for quick data access. The processor also has special short load and short store instructions to access data in the SRAM.

The Tiny Data Area (TDA) optimization, similar to the Small Data Area (SDA) and Zero Data Area (ZDA) optimizations, reserves a register to point into the SRAM area. Certain variables and data structures can then be accessed as offsets from this TDA base register using the short load and short store instructions.

Like SDA and ZDA, TDA is strictly an optimization technique and does not change the basic function of the program. It produces code that is both smaller and faster than code that accesses variables and data structures using 32-bit addresses.

TDA Implementation and Limitations

To implement TDA, the compiler, assembler, and linker define a special program data section, `.tdata`. Both initialized data and uninitialized (zero) data, accessible via TDA, are put into `.tdata`.

Due to the limitations on the offset-based addressing for accessing TDA variables, the size of the TDA is limited to a total of 128 bytes (for byte access) or 256 bytes (for half word and word access), even though the entire on-chip SRAM area may be much larger. Because most programs have data segments that are much larger than this, only a few variables can go into the TDA. The programmer decides which variables are placed into the TDA. The Green Hills V850 compiler provides

variable-by-variable control over the use of the TDA through a mechanism based on the `#pragma` statement.

Pragma Directives

Two `#pragma` directives control the selection of variables for the TDA. The directive `#pragma ghs starttda` causes the compiler to place all following variables in the Tiny Data Area. The directive `#pragma ghs endtda` disables this mode. The result is that variables defined or declared between these two `#pragma` directives are placed into the TDA.

The compiler generates shortened addressing for variables that will be placed into the Tiny Data Area. At the same time, those variables are placed into a special TDA section, `.tdata`, rather than the regular `.data` and `.bss` sections.

The program source modules must agree about which variables are going into the TDA. If module `A` accesses a variable defined in module `B` as though it is in the TDA, but module `B` is not actually placing it into the TDA, this causes a link-time error. (For more information, see “Linker Errors” on page 98.)

The user must make sure this agreement exists. One useful approach is to put all `#pragma` directives for TDA control into header files that are included by all modules referring to those variables.

If a variable is declared `extern` between the TDA `#pragma` directives and later in the same module as a global variable without `#pragma` directives, it still goes into the TDA. Any declaration of the variable between the TDA `#pragma` directives is sufficient to cause it to be put into the TDA.

The linker includes the `.tdata` section as needed for proper use of TDA. Also, a special global symbol, `__ep` (note that there are two leading underscore characters “_”), is added to the user program by the linker. The user program's startup code uses this symbol to initialize the TDA base register.

Base Register

The Green Hills V850 compiler uses the `ep` register (`r30`) as the TDA base register. The offset-based addressing on the V850 processor is unsigned. Thus, the TDA

base register is pointed to the start of the Tiny Data Area, providing an addressable range for the following:

- 128 bytes for signed byte (8-bit) access
- 256 bytes for signed half-word (16-bit)access
- 256 bytes for all word (32-bit) access
- 32 bytes for unsigned half-word (16-bit) access
- 16 bytes for unsigned byte (8-bit) access

There are only two unsigned accesses available for the V850E architecture and newer.

User startup code (**crt0.o**, see "Startup Module crt0.o" on page 815) can reference the special C symbol `_ep` for setting the appropriate initial value of the `ep` register. This symbol is defined in **v800_ghs.h**, and is set at link time.

Linker Errors

Since the TDA size limitation applies to the program as a whole, the linker checks that the size does not exceed 256 bytes. For each reference to a TDA variable, the linker calculates the offset from the TDA base register in the addressing instruction. If this offset exceeds the limit allowed for this instruction, the linker prints the following error message:

```
tiny data area overflow: offset didn't fit in width bits
        while performing relocation in file filename
        at location location, to reference symbol symbol_name
```

The *symbol_name* is the name of the variable causing the offset to be too large; *offset* is its location in the `.tdata` section. The *filename* and *location* show where the illegal reference occurred.

This error occurs when too much data is requested to be placed into the TDA. To correct the error, decrease the number of variables that go into the TDA.

Another reason for this link error is inconsistent use of the `#pragma` directives between modules.

Combining TDA Modules with Other Modules

Because the TDA base register `ep (r30)` is always reserved for TDA, modules built with the Tiny Data Area optimization can be combined safely with modules that do not use TDA. However, modules compiled with **-notda**, cannot be safely combined with modules that make use of TDA. Because the V850 and RH850 libraries do not use TDA, they can be safely combined with any module, regardless if **-notda** is used. For more information, see "Text and Data Placement" on page 125.

Note

The TDA base register is only reserved when TDA is enabled in the compiler, which is not the default behavior. Modules compiled with the TDA disabled (**-notda**, the default) cannot be combined with modules that use the TDA. Attempting to do so will result in an error from the linker. To use the TDA, all modules must be compiled with TDA enabled. See **V850 Tiny Data Area** in "Text and Data Placement" on page 125 for more information about **-mtda** and **-single_tda**. The Green Hills Libraries are compiled with the TDA enabled. Consequently, the libraries can be safely used with both applications using the TDA and with applications that disable use of the TDA.

A problem can occur if one module defining a variable is not compiled with TDA, while another module using that variable is compiled with TDA. The second module may attempt to use TDA addressing to reference the variable, but the variable will not actually be in the Tiny Data Area, which may produce a link-time error (see "Linker Errors" on page 98). To correct this problem, recompile the modules with consistent usage of the TDA optimization.

Multiple TDA

The Multiple TDA optimization makes it possible to have multiple TDA sections, thereby increasing the usage of short load and short store instructions. This results in smaller code.

To use the Multiple TDA optimization:

Set the **Target→Text and Data Placement→V850 Tiny Data Area** to **Multiple TDAs** (**-mtda**).

Function Categories

The `#pragma` directives provide multiple TDA optimization by supporting separate Tiny Data Area sections on a function-by-function basis. Every function falls into one of three categories:

- *no-TDA* — Functions that do not use the TDA and do not call any function that use the TDA. A *no-TDA* function can call an *export-TDA* function, but not a *named-TDA* function.
- *named-TDA* — Functions that use a particular Tiny Data Area section.
- *export-TDA* — Functions that use a particular Tiny Data Area section and have prologue code that saves the TDA base register `ep` (`r30`). These functions load the base register with the virtual starting address of the particular Tiny Data Area section. Epilogue code restores the base register before the *export-TDA* functions return.

Multiple TDA Pragma Directives

The Multiple TDA `#pragma` directives support separate Tiny Data Area sections on a function-by-function basis.

For *no-TDA* functions,pragma directives are not necessary:

```
int foo(int i)
{
    return i+1;
}
```

For *named-TDA* functions, use

```
#pragma ghs function tdata="sectname"
```

where *sectname* is the name of the particular named Tiny Data Area section, which by convention begins with a period (“.”).

Example 2.10. Using Named-TDA Functions

```
#pragma ghs function tdata=".tdataA"

int named_tda_fooA(int i)
{
    return i+2;
}

int named_tda_fooB(int i)
{
    return i+3;
}
#pragma ghs function
```

After the `#pragma ghs function` directive, all functions become *no-TDA* again, until the next `#pragma ghs function tdata=` directive.

For *export-TDA* functions, use

`#pragma ghs function exporttda tdata="`*`sectname`*`"`

where *sectname* is the name of the particular named Tiny Data Area section, which by convention begins with a period (“.”).

Example 2.11. Using Export-TDA Functions

```
#pragma ghs function exporttda tdata=".tdataB"

int export_tda_fooA(int i)
{
    return i+4;
}
int export_tda_fooB(int i)

{
    return i+5;
}
#pragma ghs function
```

Multiple TDA Variables Pragma Directives

```
#pragma ghs section tdata=".tdataA"
#pragma ghs starttda
int myvar=1;
#pragma ghs endtda
#pragma ghs function exporttda tdata=".tdataA"

int foo(int i)
{
    return i+myvar;
}
```

Multiple TDA Calling Convention

Because all tiny data variable accesses occur through the same base register, `ep` (`r30`), a calling convention ensures that all function calls are legal.

Legal function calls include:

- A *named-TDA* function calling another *named-TDA* function with the same named section
- A *named-TDA* function calling any *export-TDA* function
- A *named-TDA* function calling any *no-TDA* function
- An *export-TDA* function calling a *named-TDA* function with the same named section
- An *export-TDA* function calling any *export-TDA* function
- An *export-TDA* function calling any *no-TDA* function
- A *no-TDA* function calling any *export-TDA* function
- A *no-TDA* function calling any *no-TDA* function

Illegal function calls include:

- A *named-TDA* function calling a *named-TDA* function with a different named section
- An *export-TDA* function calling a *named-TDA* function with a different named section

- A *no-TDA* function calling any *named-TDA* function

Multiple TDA Special Symbols

To enforce the Multiple TDA calling convention, the Green Hills V850 compiler defines a special symbol for each function and a special symbol reference for each function call. This information is passed to the linker, which checks the legality of the calls.

For every *named-TDA* function, the compiler defines the following symbol

```
..lxtda.functionname.tdaname
```

where *functionname* is the function and *tdaname* is the particular Tiny Data Area section the function uses. The special symbol indicates the function can only be called by a *named-TDA* function of the same Tiny Data Area section, or an *export-TDA* function of the same Tiny Data Area section.

Example 2.12. Using Multiple TDA Special Symbols

Given the following function:

```
#pragma ghs function tdata=".tdataA"
int named_tda_fooA(int i)
{
    return i+2;
}
```

the compiler will place the following special symbol definition in the assembly file:

```
..lxtda._named_tda_fooA..tdata1=:0
```

For every *export-TDA* and *no-TDA* function, the compiler defines the symbol

```
..lxtdaG.functionname
```

where *functionname* is the callee.

Example 2.13. Using Export-TDA and No-TDA Functions

Given the following functions:

```
#pragma ghs function exporttda tdata=".tdataB"

int export_tda_fooA(int i)
{
    return i+4;
}

#pragma ghs function
int blot(void)
{
    return 1;
}
```

the compiler will define the following special symbols:

Symbol	Corresponding Function
`..lxtdaG._export_tda_fooA`	`export_tda_fooA()`
`..lxtdaG._blot`	`blot()`

For each function call within every *named-TDA* and *export-TDA* function, the compiler generates a reference to the special symbol

`..lxtda.`*`functionname`*`.`*`tdaname`*

where *functionname* is the callee and *tdaname* is the TDA name of the caller.

Example 2.14. Using Named-TDA and Export-TDA Functions

Given the following function:

```
#pragma ghs function tdata=".tdataA"
int named_tda_fooA(int i)
{
    return bar(i+1);
}
```

the compiler will place the following special symbol definition in the assembly file:

```
..tmp==..lxtda._bar..tdataA
```

For each function call within every *no-TDA* function, the compiler generates a reference to the special symbol

```
..lxtdaG.functionname
```

where *functionname* is the callee.

Example 2.15. Using No-TDA Functions

Given the following function:

```
#pragma ghs function
int bel(int i)
{
    return canto(i+1);
}
```

the compiler will place the following special symbol definition in the assembly file:

```
..tmp==..lxtdaG._canto
```

Multiple TDA Linker Errors

If a *no-TDA* function calls a *named-TDA* function, the linker prints the following error message:

```
Function 'functionname' called from a no-TDA function in 'filename.o'
```

where *functionname* is the name of the *named-TDA* function called, and *filename* is the file in which this illegal call was made.

If a *named-TDA* or an *export-TDA* function calls a function that is in a different name Tiny section, the linker prints the following error message:

```
Cross-TDA call to function 'functionname' from TDA 'tdaname' in 'filename.o'
```

where *functionname* is the *named-TDA* function called, *tdaname* is the Tiny Data Area section of the caller, and *filename* is the file in which the illegal call was made.

Multiple TDA Assembly Language Programming

The Multiple TDA calling convention must be followed for hand-coded assembly functions and function calls.

Position Independent Code (PIC)

Position independent code (PIC) is code that can be run at an address that was not known at the time the code was linked. You can use PIC to create executable files that can be placed anywhere in memory.

PIC involves the calculation of the address of any object in a `.text` section, typically a function label. Addresses in C can appear as:

- Local branches
- Normal function calls
- Far function calls
- Switch statements
- Function pointers

Instructions that implement local branches and normal function calls are relative to the program counter and therefore are inherently position-independent. No additional code is needed to achieve position independent code for these instructions. To achieve position independent code for far function calls, switch statements, and function pointers, the compiler must generate additional code.

To compile a program with position independent code:

Set the **Target→Text and Data Placement→Position Independent Code** option to **On** (**-pic**).

When you enable PIC:

- The linker suppresses errors that identify overlapping sections.
- The `.rodata` section is used less often.

Running a PIC Program with the Simulator

When running a PIC program on the simulator without the MULTI Debugger, you should pass the **-text *offset*** option in order to specify the PIC offset. For example,

to run the PIC program **myprog** on the simulator and to load the program code at address `0x100000`, enter:

```
sim850 -text 0x100000 myprog
```

Debugging a PIC Program

When debugging a PIC program in the MULTI Debugger, you must specify the PIC base address so that the Debugger can properly offset the addresses of code. You can specify the PIC base address in two ways:

- Use the **-text** option when starting the MULTI Debugger from the command line. For example, to start debugging a PIC program **myprog** with the PIC base address set to `0x100000`, enter:

  ```
  multi -text 0x100000 myprog
  ```

- Set the `_TEXT` variable while the Debugger is running. For example, suppose you open a PIC program **myprog** in the Debugger and want to set the PIC base address to `0x100000`. In the Debugger command pane, enter:

  ```
  _TEXT=0x100000
  ```

Combining PIC with Other Modules

You can link modules that were compiled with PIC with modules that were not, as long as the resulting program does not run as a PIC program. Any program that is running as PIC must have all its source modules compiled with PIC.

To reduce the number of library variants, Green Hills libraries are built with PIC enabled. Therefore, the same libraries can build both PIC and non-PIC executables. PIC addressing calculations are made relative to the `PC` register.

Position Independent Data (PID)

If you enable *Position Independent Data* (PID), the executable file is built so that the program's data can be located at any position in memory. .

Because data accesses are not inherently position-independent, PID requires a base register that is initialized by the operating system, monitor, or program loader to pinpoint the location of the data segment in the memory.

To compile a program with position independent data:

Set the **Target→Text and Data Placement→Position Independent Data** option to **On** (**-pid**).

When you enable PID:

- The linker suppresses errors that identify overlapping sections.
- The `.rodata` section is not used.

Running a PID Program with the Simulator

When running a PID program on the simulator without the MULTI Debugger, you should pass the **-data *offset*** option in order to specify the PID offset. For example, to run the PID program **myprog** on the simulator and to load the program data at address `0x200000`, enter:

```
sim850 -data 0x200000 myprog
```

Debugging a PID Program

When debugging a PID program in the MULTI Debugger, you must specify the PID base address so that the Debugger can properly offset the addresses of data. You can specify the PID base address in two ways:

- Use the **-data** option when starting the Debugger from the command line. For example, to start debugging a PID program **myprog** with the PID base address set to `0x200000`, enter:

```
multi -data 0x200000 myprog
```

- Set the `_DATA` variable while the Debugger is running. For example, to set the PID base address to `0x200000`, enter the following in the Debugger command pane:

  ```
  _DATA=0x200000
  ```

Initializing the PID Base Register

The PID base register, by convention `r4`, must be initialized to a fixed offset from the starting address of the data segment (in the case of SDA projects, the small data area is used instead; see "Special Data Area Optimizations" on page 87). One way of doing this is to have the loader, monitor, or operating system initialize the base register when loading the program on the target. Alternatively, the program's startup code can initialize the base register each time it runs. If the data segment of the program is loaded into ROM and then copied from ROM to RAM at startup, be sure that the base register is initialized properly by the operating system or startup code.

Combining PID with Other Modules

You can link modules compiled with PID with modules that were not, as long as the resulting program does not run as a PID program. You cannot run a mixed PID and non-PID program in PID mode; to run a program in PID mode, you must compile all its modules with PID.

To reduce the number of library variants, Green Hills embedded system libraries are built with PID enabled, so the same libraries can be used in both PID and non-PID executables. For PID code to work correctly in a non-PID executable, the PID base register must be initialized to the start of the data area. The Green Hills run-time environment does this automatically.

Customizing the Green Hills Run-Time Environment

You can customize the Green Hills run-time environment to implement or enhance the environment for a particular hardware system by:

- Modifying the source code of the run-time object modules **libsys.a**, **libstartup.a**, and **crt0.o**, and then rebuilding these object modules (see "Customizing the Run-Time Environment Libraries and Object Modules" on page 813). Each source file, including files with only assembly language, is fully commented. These source files are located in the **src/libsys** and **src/libstartup** directories of your installation.
- Modifying the linker directives file for special program sections that **libsys.a**, **libstartup.a**, and **crt0.o** use to set up the run-time environment (see "Customizing the Run-Time Environment Program Sections" on page 464).

Other Topics

Renaming the Output Executable

To change the name of the output file:

Enter the desired name in the **Project→Output Filename** (**-o *name***) option.

Specifying an Alternate Program Start Address

The linker normally uses the address of the global symbol `_start` as the start address for the user program. To specify an alternate start address:

Specify a new symbol in the **Linker→Start Address Symbol** (**-e=*symbol_name***) option.

For example, to use the symbol `newstart` as the program start address, enter the following from the command line:

```
ccv850 -e=newstart file.s
```

If the entry symbol is copied from ROM to RAM during program startup, the entry point is in the ROM version of the entry symbol.

Interrupt Routines

Writing Interrupt Routines

In C and C++, if you want a function to handle hardware interrupts or exception conditions, declare that function as an interrupt routine. When you declare a function as an interrupt routine, the compiler generates code in the function prologue that saves all the caller-save and callee-save registers it requires. It also generates code in the function epilogue that restores those registers to the state they were in before the function was called.

These functions should be of `void` type and should take no arguments.

Note

Interrupt functions on the V850 and RH850 use different calling conventions than normal functions. Specifically:

- Interrupt functions must return using the `reti` (for the V850E2 and older architectures) or `eiret` (for the V850E2R and newer architectures) instruction rather than the `jmp` instruction.
- Normal functions are permitted to destroy the contents of all temporary registers. Interrupt functions are not permitted to destroy the contents of temporary registers.

You can use either of the following methods to declare a function as an interrupt routine:

- Place `#pragma ghs interrupt` immediately before the function.
- Prepend the `__interrupt` keyword to the function definition.

Interrupts are already disabled when an interrupt routine is called, and are enabled immediately when it returns. The compiler does not normally insert code to enable interrupts; this is left to the implementation of the interrupt routine. To instruct the compiler to insert code that enables interrupts early on in the interrupt function's prologue, pass the `enabled` parameter to `#pragma ghs interrupt` using the following syntax:

```
#pragma ghs interrupt(enabled)
```

The compiler assumes in most cases that user code in the interrupt routine enables interrupts, even if you have not passed the `enabled` parameter. This means that the compiler assumes interrupts can be interrupted (they are reentrant). If you know that an interrupt routine cannot be interrupted, pass the `nonreentrant` parameter to `#pragma ghs interrupt` using the following syntax:

```
#pragma ghs interrupt(nonreentrant)
```

This directive generally causes the compiler to create a smaller function prologue and epilogue, resulting in a faster and smaller interrupt routine. Use this option only when you use inline prologues and epilogues (see “Function Prologues” on page 249).

Establishing Interrupt Vectors with the intvect Pragma Directive

To establish an interrupt vector which is needed to call an interrupt routine in C and C++, use:

```
#pragma intvect function integer_constant
```

The compiler arranges for a jump to *function* to be placed in memory at the address specified by *integer_constant* using a `.org` directive to the assembler. The named *function* must be a valid interrupt routine that returns `void`.

Thus, the following code fragment declares an interrupt routine, `foo`, with an interrupt vector at address `0x60`, established by `#pragma intvect`:

```
__interrupt void foo(void);

#pragma intvect foo 0x60

__interrupt void foo()
{
    ...
}
```

Note

This feature is supported only when using a Green Hills assembler and is not available in binary code generation mode. You must ensure that the call destination is within range of the vector or unpredictable behavior may result.

Symbolic Memory-Mapped I/O

Embedded systems may have memory locations reserved for I/O, with the registers of a hardware device mapped in these locations. Several techniques access such memory-mapped I/O devices, but using specially named sections has some advantages.

For memory-mapped I/O, the variables that correspond to the memory locations should be kept in one or more sections that contain only memory-mapped locations. These sections should not be initialized during program startup.

This approach creates a C language structure definition that matches the allocation of the I/O registers in the memory space. Named structure fields are assigned to correspond to the I/O registers. Next, the `#pragma` directive is used to define a variable of the `struct` type within a named `.data` section that holds only this variable. A linker section map places this named section at the position in memory of the I/O device. The I/O registers are now structure fields of the variable in the named section.

Structures should be defined with the `volatile` qualifier, because the registers' values can change without being modified explicitly by the program.

With this approach, the variable definition can be placed in a source file of its own. This file is compiled to an object file that is shared among different projects that need to refer to this I/O device. Each project needs the object file, an entry in the section map to position the section, and a header file to define the `struct` type.

In addition, you can then display the named variable in a MULTI Debugger data explorer to provide ongoing displays of the state of the I/O registers, and edit the fields of the data explorer to cause writes to the I/O registers. This displays and manipulates memory-mapped I/O registers by MULTI view windows, similar to built-in registers.

For example, consider a simplified universal asynchronous receiver-transmitter ("`UART`") that has three 16-bit registers: an output register, an input register, and a control/status register, mapped at locations `0x1000`, `0x1002`, and `0x1004` respectively. With code containing the following lines:

```
#pragma ghs section bss=".uartsec"
struct {
    short Tx;    /* Transmit Register        */
    short Rx;    /* Receive Register         */
    short CSR;   /* Control/ Status Register */
} volatile uart;
#pragma ghs section bss=default
```

and a section map entry in the linker directives file such as:

```
SECTIONS {
    ...
    .uartsec 0x1000 :
    ...
}
```

Code can then refer to the registers as `uart.Tx`, `uart.Rx`, and `uart.CSR`.

Within MULTI, the command `view uart` views a window showing the contents of the UART registers. Because MULTI must perform read operations to display this data, this approach is not suitable for registers that are sensitive to read and write operations (that is, registers for which accessing a register triggers an activity or referencing a data location near the register causes problems).

Verifying Program Integrity

If you want to verify that sections in memory are identical to those created by the linker, set the **Linker→Link-Time Checking→Checksum** option to **On** (**-checksum**). This option instructs the linker to calculate a Cyclic Redundancy Check (CRC) checksum for each section that contains text or data and store it in the section. To exclude a section from checksum creation, use the `NOCHECKSUM` attribute when defining that section in the section map.

When the **Checksum** option is enabled, the symbol `__ghs_checksum` is defined by the linker to be the number of bytes at the end of each section reserved for the checksum. It is currently 4 bytes.

To verify a section upon initialization, scan the `__secinfo` table, calculate the same CRC on all but the last *`__ghs_checksum`* bytes of the section, and compare the result to the stored CRC. A match indicates that the program has the same byte values in memory that it had when the linker created the executable file. For more information about performing checksums, see ***install_dir*/src/libstartup/cksum.c**. For more information about the `__secinfo` structure, see ***install_dir*/src/libsys/indsecinfo.h**.

Chapter 3

Builder and Driver Options

Contents

This chapter lists all of the Builder and driver options.

These options control various aspects of the compilation, assembly, and linking of your project. The following information is listed for each option:

- The name of the option as it appears in the **Build Options** window.
- A description of the option.
- Permitted option settings (if applicable).
- The equivalent driver option and description (if any).

For information about setting build options in the Project Manager, see the documentation about setting Builder options in the *MULTI: Managing Projects and Configuring the IDE* book.

To pass options to the compiler driver, add them to your command line or makefile. For more information about using the driver, see Chapter 1, “The Compiler Driver” on page 3.

Target Options

This is a top-level option category.

These options allow you to control aspects of compilation relating to your target, such as available memory models and forms of floating-point support, and to control the instruction set to be used.

For additional, more specialized options, see “Target Options” on page 249.

Register Description File

Specifies a **.grd** file, which contains a description of the target board's registers, and instructions on how MULTI may access them. The equivalent driver option is:

- **--register_definition_file=*file*.grd**

Register r2

Controls the treatment of register `r2`. Permitted settings for this option are:

- **Reserve for User** (**-reserve_r2**) — Reserves this register for use by the user.
- **Permit Compiler Use** (**-noreserve_r2**) — [default] Permits the compiler to use this register as a temporary register.

Register r5

Controls the treatment of register `r5`. Permitted settings for this option are:

- **Reserve for User** (**-reserve_r5**) — Reserves this register for use by the user. This disables the use of ROM SDA; consequently, read-only data will not be placed in an SDA section.
- **Permit Compiler Use** (**-noreserve_r5**) — [default] Permits the compiler to use this register.

Register Mode

Controls the number of registers that will be used by the compiler. You may want to restrict this number in order to guarantee the availability of particular registers for hand-written assembly. For more information, see “Register Usage” on page 32. Permitted settings for this option are:

- **Restrict to 22 Registers** (**-registermode=22**)
- **Restrict to 26 Registers** (**-registermode=26**)
- **Standard 32 Registers** (**-registermode=32**) — [default]

Set Register r20 to the Value 255

Controls an optimization in which register `r20` is set to the value 255 at startup, and is subsequently treated as read-only. Permitted settings for this option are:

- **On** (**-r20has255**) — Register `r20` is set to the value 255.
- **Off** (**-nor20has255**) — [default] Register `r20` is available for general use.

When this optimization is enabled, the compiler may generate smaller code by accessing `r20`. For example, the 4-byte instruction:

```
andi 255, r3, r3
```

is replaced with the 2-byte instruction:

```
and r20, r3
```

Set Register r21 to the Value 65535

Controls an optimization in which register `r21` is set to the value 65535 at startup, and is subsequently treated as read-only. Permitted settings for this option are:

- **On** (**-r21has65535**) — Register `r21` is set to the value 65535.
- **Off** (**-nor21has65535**) — [default] Register `r21` is available for general use

When this optimization is enabled, the compiler may generate smaller code by accessing `r21`. For example, the 4-byte instruction:

```
andi 65535, r3, r3
```

is replaced with the 2-byte instruction:

```
and r21, r3
```

This option implies **Set Register r20 to the value 255** (see above).

Data Bus Width

Specifies the data-bus width of the target hardware. Any bits in a target address above the bus width are assumed to be masked off during a data access. This option is passed to `elxr` to resolve section mappings. Permitted settings for this option are:

- **24-bit data bus** (**-data_bus_width=24**)
- **26-bit data bus** (**-data_bus_width=26**)
- **29-bit data bus** (**-data_bus_width=29**)
- **32-bit data bus** (**-data_bus_width=32**) — [default]

For example:

If your target hardware has a 29-bit data bus width, a section placed at address `0xFFFF0000` in the link map is viewed by `elxr` to actually be at `0x1fff0000`. This is useful when using the ZDA capability of the V850 and RH850 compiler on targets with a small data bus because it still allows data sections to be placed within a small negative offset from zero, which is suitable for use with ZDA. Otherwise, `elxr` might complain that this section does not fit within the specified address memory region for the target.

Note: If **-data_bus_width=29** is specified, `elxr` first tries the address `0xffff0000` as usual, and then tries the 29-bit wide value (`0x1fff0000`).

Floating-Point

This option category is contained within **Target**.

Floating-Point Mode

Specifies how to perform floating-point calculations. Permitted settings for this option are:

- **No Floating-Point** (**-fnone**) — Disallows all floating-point operations (and directs the C and C++ compilers to give an error for any use of floating-point variables or constants), thus greatly reducing the size of such library functions as `printf` and `scanf`. Passes `-D__NoFloat__` to allow source code and header files to hide references to floating-point objects. If your code uses floating-point operations, then an error will be generated. For more information, see "Predefined Macro Names" on page 734.

  Because code related to floating-point operations may be linked in to your program even if it does not use floating-point variables, using this option may reduce the size of your program.

  Because this setting requires support from the system libraries and header files, it is not supported for environments using embedded operating systems with their own C or C++ headers and libraries. In addition, certain target-specific header files may require the use of floating-point types.

  This option implies **-no_float_scanf**.
- **Hardware Single, Software Double** (**-fsingle**) — Instructs the compiler to generate floating-point instructions to perform single-precision floating-point operations, and to generate library subroutine calls to emulate double-precision floating-point operations.
- **Software Emulation** (**-fsoft**) — Specifies *software floating-point* (SFP) mode, in which the compiler uses integer registers to hold floating-point data and generates library subroutine calls to emulate floating-point operations, regardless of the capabilities of the selected processor.

Treat Doubles as Singles

Controls the treatment of `double` types. Permitted settings for this option are:

- **On** (**-floatsingle**) — Treats `double` types as `float`, so that no 64-bit instructions are required.

Note

This mode does not fully comply with the C language specification. For example, the `double` data type does not have the required minimum precision or a precision greater than that of the `float` data type.

- **Off** (**-nofloatsingle**) — [default] Treats `double` types normally

Floating-Point Functions

Controls the replacement of calls to floating-point math library functions (such as `sqrt()`), with the corresponding instructions in the instruction set architecture. Permitted settings for this option are:

- **On** (**-ffunctions**) — Use instructions. These instructions may not be exactly equivalent to the function calls they are replacing. For example, they may not exhibit the same side effects or set `errno`.
- **Off** (**-fnofunctions**) — [default] Use floating-point library functions. Implied by Strict ANSI (**-ANSI**) and Strict ISO C99 (**-C99**) as the instructions do not set `errno` and so are not ANSI compliant.

Floating-Point Precise Signed Zero

The Green Hills tools generally conform to IEEE floating-point standards as much as possible. However, better code performance in both size and speed can be obtained by allowing small deviations from the standard with respect to the treatment of signed zero quantities (`-0.0` versus `+0.0`). When **-no_precise_signed_zero** is used, the compiler is permitted to replace sequences of operations that might have resulted in a negative zero (-0.0) outcome with one that may instead produce a positive zero result. Similarly, sequences that resulted in a positive zero result may be replaced with one resulting in a negative zero result. Consequently, because negative zero and positive zero are treated as equal quantities in every respect by the compiler, this should have no impact on the accuracy of user code when the distinction between signed zero quantities is of no importance. Permitted settings for this option are:

- **On** (**-precise_signed_zero**)

- **Off** (**-no_precise_signed_zero**) — [default]

Floating-Point Precise Signed Zero Compare

By default, Green Hills tools conform to the IEEE requirement that negative zero (`-0.0`) quantities must compare equal to positive zero (`0.0`) quantities. Consequently, expressions involving an equality such as `-0.0 == 0.0` or `0.0 <= -0.0` always returns true, and expressions involving an inequality such as `-0.0 != 0.0` and `-0.0 < 0.0` always returns false. On some architectures, better optimization strategies can be employed if the compiler is permitted to ignore this requirement by using **-no_precise_signed_zero_compare**. However, caution must be employed when using this option because it can affect both the accuracy and correctness of user floating point code if that code relies on equalities between the zeros. You may want to consider enabling **-precise_signed_zero** when attempting to use **-no_precise_signed_zero_compare**. Permitted settings for this option are:

- **Allow -0 to differ from +0 when it helps optimizations** (**-no_precise_signed_zero_compare**)
- **Force -0 to compare equal to +0 always** (**-precise_signed_zero_compare**) — [default]

Use Floating-Point in stdio Routines

Controls the use of floating-point in `stdio` operations. This option is deprecated and may be removed in future versions of **MULTI**. Permitted settings for this option are:

- **On** (**-floatio**) — [default] Use standard libraries.
- **Off** (**-nofloatio**) — Use **libnoflt.a**, a library containing special versions of `printf`, `scanf`, and related functions which, since they contain no floating-point operations, are therefore smaller. Floating-point formats (`%f`, `%g`, `%e`) are not supported. In environments where floating-point uses large library support routines, this option can save space for programs that use `printf`, but which do not require floating-point. This option only works with whichever **WChar Size** option is the default for your processor (**-wchar_u16** or **-wchar_s32**).

Text and Data Placement

This option category is contained within **Target**.

These options control the placement of text and data in memory.

Special Data Area

Specifies that data up to a certain size should be placed in the *Small Data Area* or *Zero Data Area*. Permitted settings for this option are:

- **Small Data Area** (**-sda**) — Enables the *Small Data Area* optimization with a threshold of 8. For more information, see "Special Data Area Optimizations" on page 87.
- **Small Data Area with Threshold** (**-sda=*size***) — Enables the *Small Data Area* optimization, where *size* specifies the threshold size for objects placed in the SDA.
- **Zero Data Area with Threshold** (**-zda=*size***) — Enables the *Zero Data Area* optimization, where *size* specifies the threshold size for objects placed in the ZDA. For more information, see "Special Data Area Optimizations" on page 87.
- **No Special Data Area** (**-nothreshold**) — [default] Disables the special data area optimization.

Valid formats for the *size* parameter threshold include: `1024`, `0x400`, and `all`. If *size* is set to `all`, then no size restriction is set on objects placed into the special data area.

V850 Tiny Data Area

Controls use of the *Tiny Data Area* Optimization. Permitted settings for this option are:

- **Multiple TDAs** (**-mtda**) — Enables multiple tiny data areas.
- **Single TDA** (**-single_tda**) — Enables a single tiny data area.
- **No TDA** (**-notda**) — [default] Disables tiny data areas. This option instructs the compiler to produce code that assumes that the TDA optimization will not be used by any module linked into the program thereby permitting the compiler

to overwrite the contents of the TDA base register `(ep)` without any attempt at preservation. Consequently, programs compiled with **–mtda** or **–single_tda** cannot be combined safely with programs compiled with **–notda**. It is recommended to use the setting **-single_tda** for modules that do not require the use of the TDA optimization, but must remain operable with those that do.

For more information, see "V850 Tiny Data Area (TDA) Optimization" on page 96.

23-bit SDA (V850E2V3 and later)

Controls whether or not the tools increase the size of the small data area (SDA) to 8 MB. Permitted settings for this option are:

- **Generate 23-bit SDA relocations for load/store instructions** (**-large_sda**) — Increases the size of the SDA to 8 MB. When accessing the SDA, the compiler generates 6-byte load and store instructions with a 23-bit offset from the SDA base register. This option is applicable only to V850E2V3 and later processor variants on which 6-byte load and store instructions are supported.
- **Generate 16-bit SDA relocations for load/store instructions** (**-no_large_sda**) — [default] Does not increase the size of the SDA. When accessing the SDA, the compiler generates 4-byte load and store instructions with a 16-bit offset from the SDA base register.

23-bit ZDA (V850E2V3 and later)

Controls whether or not the tools increase the size of the zero data area (ZDA) to 8 MB. Permitted settings for this option are:

- **Generate 23-bit ZDA relocations for load/store instructions** (**-large_zda**) — Increases the size of the ZDA to 8 MB. When accessing the ZDA, the compiler generates 6-byte load and store instructions with a 23-bit offset from the ZDA base register. This option is applicable only to V850E2V3 and later processor variants on which 6-byte load and store instructions are supported.
- **Generate 16-bit ZDA relocations for load/store instructions** (**-no_large_zda**) — [default] Does not increase the size of the ZDA. When accessing the ZDA, the compiler generates 4-byte load and store instructions with a 16-bit offset from the ZDA base register.

Linker-Based SDA23/ZDA23 Shortening (V850E2V3 and later)

Controls a linker-based optimization that shortens 6-byte load and store instructions to 4-byte instructions when possible. This option usually produces smaller code for programs compiled with **-large_sda** or **-large_zda**. Permitted settings for this option are:

- **Convert 23-bit SDA relocations to 16-bit in load/store instructions when possible** (**-shorten_loads**) — Converts 6-byte load and store instructions with a 23-bit offset from the SDA base register to an equivalent 4-byte instruction with a 16-bit offset, if the offset can fit in 16 bits. If you enable **-Olink**, this optimization is enabled implicitly.
- **Do not convert 23-bit SDA relocations to 16-bit in load/store instructions** (**-no_shorten_loads**) — [default]

Linker-Based Move Shortening

Permitted settings for this option are:

- **Convert 32 and 48-bit move relocations to 16-bit in move instructions when possible** (**-shorten_moves**) — If you enable **-Olink**, this optimization is enabled implicitly.
- **Do not attempt to shrink move relocations in move instructions** (**-no_shorten_moves**) — [default]

Allow Common Variables in ZDA

Controls whether common variables are permitted in the Zero Data Area. Permitted settings for this option are:

- **On** (**-zero_commons**) — [default] Permits common variables in ZDA.
- **Off** (**-no_zero_commons**) — Does not permit common variables in ZDA.

Position Independent Code

Controls the generation of *Position Independent Code* (PIC). Permitted settings for this option are:

- **On** (**-pic**)
- **Off** (**-nopic**) — [default]

For more information, see "Position Independent Code (PIC)" on page 106. This option cannot be used in conjunction with **-layout**.

Position Independent Data

Controls generation of *Position Independent Data* (PID). Permitted settings for this option are:

- **On** (**-pid**)
- **Off** (**-nopid**) — [default]

For more information, see "Position Independent Data (PID)" on page 108. This option cannot be used in conjunction with **-layout**.

Far Calls

Controls whether the compiler will generate a far call for every call. Permitted settings for this option are:

- **Make All Calls Far Calls** (**-farcalls**) — Enables generation of a far function call for every call. This allows for functions to be located at any distance from the caller.
- **Make No Far Calls** (**-nofarcalls**) — [default] Disables generation of far function calls. Large programs or programs with discontinuous text sections may not link if the range of the call instruction is exceeded.

For more information, see "Near and Far Function Calls" on page 84.

Linker-Based Far Call Patching

Controls a linker-based optimization which generates far calls only for calls which are out of range and would otherwise fail. This option will generally produce smaller code than generating far calls for every call (see "Far Calls" on page 128). Permitted settings for this option are:

- **Patch Far Calls When Necessary (-farcallpatch)** — [default]
- **Do Not Patch Far Calls (-nofarcallpatch)**

Placement of Zero-Initialized Data

Controls the allocation of statically-initialized variables and arrays that are explicitly initialized to zero. Allocating such objects to an uninitialized section will generally reduce the size of the executable ROM image. Permitted settings for this option are:

- **Place Zero-Initialized Data in BSS Sections (-discard_zero_initializers)** — Places statically-initialized variables that are explicitly initialized to zero as if they were defined but not initialized. For example, if `int x = 0;` would normally place `x` in the `.data` section, the compiler may place it in the `.bss` section instead, as if you had declared `x` with `int x;`. Regardless, `x` would still be initialized to zero.
- **Place Zero-Initialized Data in Data Sections (-no_discard_zero_initializers)** — [default] Places statically-initialized variables that are explicitly initialized to zero in the same way as those initialized to other values. If a global variable is not explicitly initialized in the source code, it may still be allocated to the appropriate `.bss` section.

Global Registers

Reserves up to 5 registers (`r20` to `r24` in that order) to hold global variables.

The equivalent driver option is:

- **-globalreg=*n***

Note

If you are building INTEGRITY code, you cannot modify the default value of this option. If you attempt to do so, the compiler issues the following message:

```
Warning: The number of global registers on INTEGRITY
is not configurable. option -globalreg=n ignored
```

For more information, see “Storing Global Variables in Registers” on page 83.

Stack Limit Checking

Controls stack limit checking, which is performed by making a call to a function `__stkchk()` in the prologue of each routine. Because these calls are added by the compiler, this option does not detect stack overflows that are caused by pre-compiled code, such as the Green Hills libraries or linker generated code. In stand-alone environments, `__stkchk` is in **libsys.a**. The code for it can be obtained in **src/libsys/ind_stackcheck.c** (see “Customizing the Run-Time Environment Libraries and Object Modules” on page 813). Permitted settings for this option are:

- **On** (**-stack_check**)
- **Off** (**-no_stack_check**) [default]

To enable stack checking on a per-function basis, use the `stackcheck` attribute (see `stackcheck` on page 682).

Check for Stack Smashing Attacks

Controls code instrumentation that provides protection against stack smashing attacks. Permitted settings for this option are:

- **On** (**-stack_protector**) — Enable protection against stack smashing attacks.
- **Off** (**-no_stack_protector**) — Do not enable protection against stack smashing attacks.

When enabled, functions with variables greater than 8 bytes in length are instrumented. The compiler places a *canary* — a special location on the stack — in the prologue, adjacent to the saved registers. This canary is marked with a specific value. In the epilogue, this canary value on the stack is compared against the original value. If the values do not match, an error function is called, terminating the program.

This technique offers protection against stack smashing attacks, which attempt to overwrite the return value, although a motivated attacker can overcome the protection using a multi-phased attack. It is important to understand that a program that contains no possibility of an array overrun is immune to all such attacks, even without the

protection afforded by a stack canary. **-stack_protector** is not a substitute for well designed software.

The canary value remains fixed throughout a single execution of the program. This value, which is initialized to a fixed value in **libsys/ind_stackcanary2.c**, should be randomly set based on system entropy. To customize the error handling routine, customize `__stack_chk_fail()` in **libsys/ind_stackcanary.c**. This function must never return.

Rename Section

Assigns variables and functions to specific user-named sections. This option works in the same way as the `#pragma ghs section` directive but does not modify the source file (see "Custom Program Sections" on page 77). The equivalent driver option is:

- **-section *secttype=sectname***

  secttype specifies which kind of code or data item is affected by the option and may be any of those listed in "Default Program Sections" on page 76 (with or without the leading period).

  sectname is the user-defined section name, which starts with a period (`.`) by convention.

  This option changes the default name of the specified section type throughout the file. Its effect is overridden by any `#pragma ghs section` in the source file with the same *secttype*, and its effect is restored by `#pragma ghs section secttype=default`.

Use Small Block Malloc

Specifies whether to use a faster or less-fragmenting `malloc()` implementation. This option only affects stand-alone applications, and has no effect when Run Time Memory Checking (**-check=alloc** or **-check=memory**) is enabled.

Permitted settings for this option are:

- **On** (**-fast_malloc**) — [default] A separate allocation pool is maintained for small blocks, resulting in faster allocation times at the expense of increased heap fragmentation. This version of `malloc()` has a slightly larger library footprint. INTEGRITY always uses this implementation, regardless of this option.
- **Off** (**-no_fast_malloc**) — Use the `malloc()` implementation that shipped with older versions of the compiler.

Generate a backwards-compatible (MULTI 5.1.x and earlier) call table in the linker

Older releases of MULTI (versions 5.1.x and earlier) made use of the `callt` instruction and its associated call table differently than current versions. Previous releases used a statically allocated call table where the first 44 entries were reserved for specific library functions. The tools now dynamically assign entries at link time, and entries are called by function name and resolved to a table index at link time. However, this new behavior is incompatible with objects built with previous releases, since they will reference the table by hard-coded indices that will no longer map to the expected entry. While the linker will attempt to automatically detect such code and adopt a compatible strategy, this option is provided to force that behavior. The linker will give a warning when it detects a need to be backwards compatible. If all objects are rebuilt with newer tools, that warning can be avoided by adding a `.callt` section to the program's linker directives file (it should be placed before .text). Permitted settings for this option are:

- **On** (**-v800_old_callt**) — Forces the linker to reserve the first 44 entries of the call table for the same library routines that occupied these entries in legacy releases.
- **Off** (**-no_v800_old_callt**) — [default] Suggests that the linker can dynamically create a call table. This option does not force this behavior, as the linker will still revert to the **-v800_old_callt** behavior when it detects incompatible object files built with previous releases of the tools.

Instruction Set

This option category is contained within **Target**.

These options control aspects of the instruction set.

Epilogues and Prologues via callt

Controls the use of the V850E `callt` instruction function prologues and epilogues. This option requires that you use the default Green Hills startup code. Permitted settings for this option are:

- **On** (**-callt**) — [default]
- **Off** (**-no_callt**)

Save CTPSW and CTPC registers in Interrupt Routines

Controls whether the `CTPSW` and `CTPC` registers are saved in interrupt routines generated by the compiler. To use this option, you must also enable **-no_callt**. This option has no effect if library prologue routines are used. Permitted settings for this option are:

- **On** (**-no_ignore_callt_state_in_interrupts**) — [default]
- **Off** (**-ignore_callt_state_in_interrupts**)

Epilogues and Prologues via prepare and dispose

Controls the use of the V850E `prepare` and `dispose` instructions for function prologues and epilogues. Permitted settings for this option are:

- **On** (**-prepare_dispose**) — [default when **-callt** is disabled]
- **Off** (**-no_prepare_dispose**) — [default when **-callt** is enabled]

Enabling this option will override any use of the `callt` instruction for prologues or epilogues (see “Epilogues and Prologues via callt” on page 133).

Use pushsp and popsp instructions

Controls the use of the RH850 `pushsp` and `popsp` instructions for interrupt function prologues and epilogues and for general function stack spills in some cases. Permitted settings for this option are:

- **On** (**-push_pop**) — [default]
- **Off** (**-no_push_pop**)

Fix EVA Load Hazard with NOP Instructions

Controls a software workaround for a hardware bug on the uPD703191Eva ICE core. Permitted settings for this option are:

- **On** (**-EVA_load_nops**)
- **Off** (**-no_EVA_load_nops**) — [default]

Allocate Pointers to the EP Register

Controls an optimization that attempts to allocate pointer variables to the `EP` register in order to use short load and store instructions. Permitted settings for this option are:

- **On** (**-allocate_ep**) — [default]
- **Off** (**-no_allocate_ep**)

The Tiny Data Area can interfere with this optimization, and you may obtain better results by disabling the **Target→Text and Data Placement→V850 Tiny Data Area** option (**-notda**).

Optimize Accesses to Adjacent Bitfields

Controls a set of optimizations for reading and writing adjacent bitfields. Permitted settings for this option are:

- **On** (**-smaller_bitops**) — [default]
- **Off** (**-no_smaller_bitops**)

Note that these optimizations do not affect volatile bitfields.

Allow 1-bit accesses to single-bit volatile fields with any base type

Permitted settings for this option are:

- **On** (**-allow_1bit_volatile_any_basetype**) — Allow the instructions `set1`, `clr1`, `tst1`, and `not1` to be used to access single-bit volatile bitfields with a base type larger than `char`. This results in more efficient code, but will result in single-byte accesses even though the volatile object has a type larger than `char`. This was the default behavior in MULTI 4.
- **Off** (**-no_allow_1bit_volatile_any_basetype**) — [default] Choose instructions that access volatile bitfields strictly according to the size of their base type.

Use divq Instruction for Divide (V850E2R and later)

When compiling for the V850E2R or later processors, this option specifies whether or not to use the `divq` and `divqu` instructions (divide quick) for integer division where possible. Permitted settings for this option are:

- **On** (**-divq**) — [default] Use `divq` and `divqu` for integer division where possible.
- **Off** (**-no_divq**) — Do not use `divq` or `divqu`. Because these instructions have a variable execution time depending on the values of their arguments, setting this option results in more deterministic behavior.

Use recipf.s/recipf.d for 1/x

Controls whether the compiler performs a divide or uses the `recipf.s` and `recipf.d` instructions when generating code for floating-point reciprocal operations (`1/x`). This option is only valid on hardware that implements these instructions. Permitted settings for this option are:

- **On** (**-recipf**)
- **Off** (**-no_recipf**) — [default]

Use FPU 3.0 instructions (RH850 and later)

Controls the use of the RH850 FPU 3.0 instructions in the compiler and the assembler. When enabled, the compiler will use the fused floating-point multiply-add and multiply-subtract instructions FMAF.S,FMSF.S,FNMAF.S,FNMSF.S in place of MADDF.S,MSUBF.S,NMADDF.S,NMSUBF.S. This will also enable the intrinsics __CVTF.HS and __CVTF.SH in the compiler. When disabled, the compiler will not use the FPU 3.0 instructions and disallow the intrinsics, and the assembler will report error if these instructions are used in the assembly.

This option applies only to RH850 and will be ignored when used on other CPU options. Permitted settings for this option are:

- **On** (**-fpu=fpu30**) — [default for RH850]
- **Off** (**-fpu=fpu20**) — [default for non-RH850]

Enable SIMD instructions (RH850 and later)

Specifies whether or not to enable support for the SIMD coprocessor in the toolchain. This will enable the SIMD intrinsics in the compiler and SIMD instruction support in the assembler. This option applies only to targets that support a SIMD coprocessor. Permitted settings for this option are:

- **On** (**-rh850_simd**)
- **Off** (**-no_rh850_simd**) — [default]

Force the save/restore of R4/R5 in ISR prologue/epilogue

Permitted settings for this option are:

- **On** (**-v850_isr_save_r4r5**) — Forces the `r4` and `r5` registers to be saved and restored in interrupt routine prologue and epilogue functions.
- **Off** (**-no_v850_isr_save_rr4r5**) — [default]

Force the save/restore of the EIIC register in ISR prologue/epilogue

Controls whether the compiler will save the `EIIC` register in interrupt prolog. Permitted settings for this option are:

- **On** (**-v850_isr_save_eiic**)
- **Off** (**-no_v850_isr_save_eiic**) — [default]

Require 4-byte alignment on offsets of word-sized load/store instructions

Forces the compiler and linker to ensure that offsets of 32-bit load and store instructions are 4-byte aligned. Generally the RH850 architecture requires the final computed address to be 4-byte aligned, but not the offset. Permitted settings for this option are:

- **On** (**-rh850_4bytewordoffset**)
- **Off** (**-no_rh850_4bytewordoffset**) — [default]

Use 64-bit Load/Store instructions (RH850 and later)

Permitted settings for this option are:

- **On** (**-64bit_load_store**) —
- **Off** (**-no_64bit_load_store**) —

Application Binary Interface

This option category is contained within **Target**.

Generate extra RH850 ABI flags in .note.renesas section (RH850 only)

Permitted settings for this option are:

- **On** (**-renesas_info**) —

- **Off** (**-no_renesas_info**) —

Check incompatible RH850 ABI flags (RH850 only)

Permitted settings for this option are:

- **On** (**-check_rh850_abi_flags**) —
- **Off** (**-no_check_rh850_abi_flags**) —

Operating System

This option category is contained within **Target**.

These options are only available when your project specifies an embedded operating system. Additionally, some of these options are only relevant when you are creating ThreadX or INTEGRITY projects.

Additional System Library

Specifies an alternate system library, which is able to override parts of the default system library (**libsys**). **-syslib** can be used to allow an operating system to integrate with the C library without modifying the system library. The equivalent driver option is:

- **-syslib=*library***

OS Directory

Specifies the root of the operating system distribution. If you plan to work on your project with both Windows and Linux/Solaris hosts, we recommend that you use forward slashes (`/`) in the path. The equivalent driver option is:

- **-os_dir** ***directory***

Event Logging

This option is available for ThreadX projects only.

Controls *event logging*, which collects event data for viewing and analysis in the MULTI EventAnalyzer (see the documentation about viewing trace events in the EventAnalyzer in the *MULTI: Debugging* book). Permitted settings for this option are:

- **On** (**-event_logging**) — Enables event logging by defining the `TX_ENABLE_EVENT_LOGGING` preprocessor symbol and linking with the **txe.a** ThreadX library.
- **Off** (**-no_event_logging**) — [default] Links with the standard **tx.a** ThreadX library.

Create S-Record File Image

Creates an S-Record file image. For more information, see "The gsrec Utility Program" on page 554. The equivalent driver option is:

- **-srec**

Project Options

This is a top-level option category.

These options allow you to control basic configuration settings for your project, such as source, output and include directories.

For additional, more specialized options, see “Project Options” on page 251.

Object File Output Directory

Puts object files into the specified *directory*, which is often a subdirectory of the current working directory. Along with the object files, the assembly listings, debugging information files, inliner files, and other intermediate files that have the same base name as the object file but a different suffix are also put into this directory. Note that the output of the linker and the output of the archiver are never put into the object file output directory. If this option is not specified, the object files will be put into the current working directory.

Multiple projects may not share an object directory.

The equivalent driver option is:

- **-object_dir=*directory***

Emulate Behaviors of a Specific Compiler Version

Specifies a previous version of the compiler in order to emulate specific behaviors for backwards compatibility. Not all behaviors are emulated; additional warnings may need to be suppressed or enabled for compatibility. Permitted settings for this option are:

- **5.0** (**-act_like=5.0**) — Act similarly to MULTI 5.0.
- **5.2** (**-act_like=5.2**) — Act similarly to MULTI 5.2.
- **2012** (**-act_like=2012.1**) — Act similarly to Green Hills Compiler 2012.1.
- **2012.5** (**-act_like=2012.5**) — Act similarly to Green Hills Compiler 2012.5.
- **2012.5** (**-act_like=2013.1**) — Act similarly to Green Hills Compiler 2013.1.

- **2012.5** (**-act_like=2013.5**) — Act similarly to Green Hills Compiler 2013.5.
- **Latest** (**-act_like=latest**) — [default] Do not emulate previous behavior.

Source Root

Specifies a source root for use by the MULTI Debugger **sourceroot** command (see MULTI: Debugging Command Reference). The Debugger resolves paths relative to this root. The equivalent driver option is:

- **-dbg_source_root** ***path***

Include Directories

Specifies a *directory* in which the compiler should search for header files. The equivalent driver option is:

- **-I*directory***

When the compiler processes source files that have `#include "`*`header_file`*`"` or `#include <`*`header_file`*`>` directives, it looks for *header_file* in any directories specified with this option. You can specify multiple directories, and the compiler will search them in the order in which they are specified. If the source file's `#include` directive declares a full path to the header file, then the compiler ignores any directories specified with this option.

For a detailed discussion of how the compiler searches for included files, see “Instructing the Compiler to Search for Your Headers” on page 70.

Do not use this option to specify the location of Green Hills header files. The driver selects these files automatically, based on other options.

Library Directories

Specifies a *directory* in which the linker searches for libraries specified by the **-l** option. The equivalent driver option is:

- **-L*directory***

All **-L** options on the command line are processed before the linker searches for the libraries specified by **-l**. If you use this option multiple times, the builder searches directories in the order in which they appear on the command line.

If you are not using the default startup code, note that this option changes only the search path for libraries. This means that the linker takes **crt0.o** from the default location, even if it finds **libstartup.a** and **libsys.a** in another specified location. For more information about how to configure your project to use custom startup code, see the documentation about settings for Stand-Alone programs in the *MULTI: Managing Projects and Configuring the IDE* book.

Do not use this option to specify the location of Green Hills libraries. The driver selects these libraries automatically, based on other options.

Libraries

Specifies a *library* for the builder to link against. The equivalent driver option is:

- **-l*name***

If *name* does not have an extension or directory path, it is assumed to be an abbreviation for **lib*name*.a**. If *name* does not have a directory path, the driver and linker search for **lib*name*.a** in the directories specified by the **-L*dir*** option in the order that the **-L** options were specified, followed by a list of default directories. For example, to specify a library `libfoo.a` that exists in a default directory, use:

```
-lfoo
```

If *name* has a directory path, *name* is assumed to be the complete filename of a library or object file that is passed directly to the linker without the **-l** option.

You can specify multiple libraries, and they will be linked against in the order specified. When using the driver, each library must be specified with its own **-l** option, for example:

```
-lfoo -lbar -lbaz
```

To prevent multiply defined symbols, it is recommended that you list your source files on the command line first, and then any **-l*library*** libraries.

Do not use this option to specify the location of Green Hills libraries. The driver selects these libraries automatically, based on other options.

Output Filename

Names the generated output file. The output file type (for example, a library or an assembly file) depends on the other options that have been specified. The builder enforces specific suffixes for some types of output files. This option has no effect when set on a Top Project. The equivalent driver option is:

- **-o *filename***

Unless there is a single input file on the command line, you cannot use **-o** with **-c**, **-E**, **-P**, **-Q**, or **-S**.

Source Directories Relative to This File

Specifies a *directory* in which the Builder should search for source files. The syntax for this option is:

- **:sourceDir=*directory***

The Builder searches for source files in the same directory as the project file that specifies them before searching directories specified by **:sourceDir**.

This option controls behavior in the Builder. There is no equivalent driver option.

Intermediate Output Directory Relative to Top-Level Project

Specifies the path where object files (and, in addition, any custom output file types) are written to, relative to the location of your Top Project (usually **default.gpj**). The syntax for this option is:

- **:outputDir=*directory***

This option is most useful for custom tools that do not allow you to specify a directory for intermediate output. For stand-alone projects that use only the Green Hills Compiler Driver, use **-object_dir** instead. **-object_dir** overrides this option.

This option controls behavior in the Builder. There is no equivalent driver option.

Optimization Options

This is a top-level option category.

This section covers the wide range of optimizations Green Hills provides to help you produce smaller and/or faster executables.

Optimization Strategy

Specifies the high-level optimization strategy that MULTI uses when compiling your project. Permitted settings for this option are:

- **Maximum Debugging And No Inlining** (**-Omaxdebug**) — Disables inlining and all optimizations.
  - *Debugging ability* — excellent
  - *Code size and speed* — not optimized
  - *Compilation speed* — moderately fast
  - *Intended use* — development for code that has a lot of inline functions
- **Maximum Debugging And Limited Optimizations** (**-Omoredebug**) — [default] Enables optimizations that do not affect debugging.
  - *Debugging ability* — excellent (except for inline functions)
  - *Code size and speed* — slightly optimized
  - *Compilation speed* — moderately fast
  - *Intended use* — development; production for projects that are not size restricted and that use the same builds as development
- **Optimize for Debuggability** (**-Odebug**) — Enables optimizations that do not affect debugging.
  - *Debugging ability* — good (debug information is unavailable in rare cases)
  - *Code size and speed* — moderately optimized
  - *Compilation speed* — fast
  - *Intended use* — development; production for projects that are not size restricted and that use the same builds as development
- **Optimize for General Use** (**-Ogeneral** or **-O**) — Enables optimizations that improve both size and performance.

  - *Debugging ability* — moderate (debug information is occasionally inaccurate)
  - *Code size and speed* — optimal for general performance
  - *Compilation speed* — moderate
  - *Intended use* — production, when speed and size are equally important
- **Optimize for Size** (**-Osize**) — Enables optimizations that improve both size and performance, and additional optimizations that improve code size at the expense of performance. When using this strategy, apply it as globally as possible. The **-Ospace** option is equivalent to **-Osize**.
  - *Debugging ability* — moderate (debug information is occasionally inaccurate)
  - *Code size and speed* — optimal size, moderately fast
  - *Compilation speed* — moderate
  - *Intended use* — production, for size-restricted projects
- **Optimize for Speed** (**-Ospeed**) — Enables optimizations that improve both size and performance, and additional optimizations that improve performance at the expense of size. We recommend that you apply this strategy only for files and functions that use the most program execution time. You can then set a different optimization strategy globally (on your project file) to get multiple benefits.
  - *Debugging ability* — poor
  - *Code size and speed* — moderately small, optimal speed
  - *Compilation speed* — moderate
  - *Intended use* — production, when speed is more important than size
- **No Optimizations** (**-Onone**) — Disables all optimizations and provides the most straightforward code generation. Inlining of explicitly marked functions is performed at this level.
  - *Debugging ability* — good
  - *Code size and speed* — not optimized
  - *Compilation speed* — fastest
  - *Intended use* — when compilation speed is your highest priority, or when instructed by Green Hills support

When deciding how to optimize your program, always begin by selecting the optimization strategy that is the best fit for your project. If you must fine-tune a strategy to your specifications, see Chapter 4, "Optimizing Your Programs" on page 293 and "Optimization Options" on page 266.

Intermodule Inlining

Enables *two-pass inlining*. If no optimization strategy is selected, this option enables **-Ospeed**. Permitted settings for this option are:

- **On** (**-OI**)
- **Off** (**-Onoinline**) — [default]

For more information, see "Inlining Optimizations" on page 294.

Linker Optimizations

Controls the linker optimizations listed in "Linker Optimizations" on page 225. Permitted settings for this option are:

- **On** (**-Olink**) — The linker optimizations that are enabled depend upon whether you are optimizing for speed or size. Many of these optimizations make complex changes to your code, and might slow down the link stage, be harder to debug, or have other drawbacks. To disable individual linker optimizations, set them to **Off** in combination with this option.

  -Olink implicitly enables **-delete**. When building shared objects, this could result in unresolved symbols. For more information about linker-specific options, see "Linker-Specific Options" on page 437.

  For more information about **-delete**, see "Deleting Unused Functions" on page 471.
- **Off** (**-Onolink**) — [default]

Interprocedural Optimizations

Performs optimizations based on all available functions, unlike most other optimizations that only consider one function at a time. Permitted settings for this option are:

- **Wholeprogram** (**-Owholeprogram**) — Enabling wholeprogram optimizations allows program control and data flow to be analyzed at a high level. Speed and size optimizations such as one call-site inlining, interprocedural constant propagation and dead code elimination, and interprocedural alias analysis are performed. This option can improve program speed and size, often simultaneously. If no optimization strategy is selected, this option enables **-Ospeed**.
- **Interprocedural** (**-Ointerproc**) — Enabling interprocedural optimizations allows optimizations based on knowledge of functions being called, such as interprocedural alias analysis. Unlike **-Owholeprogram**, **-Ointerproc** does not require that the entire program is available during compilation. However, a strict subset of the optimizations from **-Owholeprogram** are applied with **-Ointerproc**. If no optimization strategy is selected, this option enables **-Ospeed**.
- **Analysis without optimizations** (**-Oip_analysis_only**) — Performs interprocedural analysis but does not apply any optimizations.
- **Off** (**-Onoipa**) — [default] Disables all interprocedural optimizations.

For more information about using interprocedural optimizations, see “Interprocedural Optimizations” on page 307.

Optimization Scope

This option category is contained within **Optimization**.

These options provide general control over the major optimizations.

Pipeline and Peephole Scope

Restricts the scope of optimizations to ensure the retention of debugging information. Permitted settings for this option are:

- **Restrict peephole optimization scope to increase debuggability (-Olimit=peephole)**
- **Restrict pipeline optimization scope to increase debuggability (-Olimit=pipeline)**

Inline Larger Functions

Controls whether the compiler will consider larger functions when **-OI** is enabled. If no optimization strategy is selected, this option enables **-Ospeed**. Permitted settings for this option are:

- **On (-OB)**
- **Off (-Onobig)** — [default] You can use `-Onobig` in a MULTI Project (**.gpj**) file, but there is no equivalent driver option.

For more information, see "Varying Inlining Thresholds" on page 301.

Unroll Larger Loops

Controls the size of the loops that the compiler will consider for unrolling. Has no effect unless loop unrolling is enabled. Permitted settings for this option are:

- **On (-Ounrollbig)** — Consider larger loops for unrolling.
- **Off (-Onounrollbig)**

For more information, see "Loop Unrolling" on page 318.

Maximize Optimizations

Controls the aggressiveness with which the compiler will pursue optimizations, without regard for compile time. Permitted settings for this option are:

- **On (-Omax)**
- **Off (-Onomax)** — [default]

For example, to optimize most aggressively for size, without regard for compilation time, set **Optimization→Optimization Strategy** to **Optimize for Size** (**-Osize**) and set this option to **On**.

Symbols Referenced Externally

This option should only be used if you are optimizing with wholeprogram optimizations, and the entire program is not available during each compilation. This often occurs if you have assembly files or other code that is linked in. Because wholeprogram optimizations rely on the knowledge of all function and variable uses, if a function or variable is referenced externally (for example, from an assembly file) you must list that symbol with **-external**.

If you are using linker optimizations and want to prevent a function from being deleted, you may also need to pass the **-keep** linker option. For more information about linker-specific options, see "Linker-Specific Options" on page 437.

The equivalent driver option is:

- **-external=*function1* -external=*function2* -external=*variable1***

function1, *function2*, and *variable1* are functions and variables in the source code that can be referenced externally.

Files Listing External Symbols

Takes a file listing externally visible symbols rather than taking the symbol names explicitly. If you have many symbols that can be referenced externally, you might want to list them in a file and pass this file to the compiler rather than specifying each symbol as shown above.

The equivalent driver option is:

- **-external_file=*file1* -external_file=*file2***

file1 and *file2* are the complete paths of the files listing symbols that can be referenced externally.

These files can contain comma- or white-space separated symbol names listing those symbols that are externally visible. Additionally, these files can be composed of global symbols defined or used in object files and libraries that are not specified during compilation, but are linked into the final executable. A list of these symbols can be generated by using the **gnm** utility (see "The gnm Utility Program" on page 541 for more information).

IP One-Site Inlining

Inlines functions that only have one call-site. This option is only meaningful if you pass the **-Ointerproc** or **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oiponesiteinlining**)
- **Off** (**-Onoiponesiteinlining**)

IP Delete Functions

Deletes unreachable functions. With or without this option enabled, functions may be removed if all of their uses are inlined. This option is only meaningful if you pass the **-Ointerproc** or **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipdeletefunctions**)
- **Off** (**-Onoipdeletefunctions**)

IP Delete Globals

Deletes constant-value global variables that are not indirectly referenced. This option is only meaningful if you pass the **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipdeleteglobals**)
- **Off** (**-Onoipdeleteglobals**)

IP Constant Globals

When a global variable is initialized to a constant value and never changed, and the variable is not indirectly referenced, the initial value is propagated into the uses of the variable. This option is only meaningful if you pass the **-Ointerproc** or **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipconstglobals**)
- **Off** (**-Onoipconstglobals**)

IP Small Inlining

Functions are inlined when the inlined size might be less than the size of the call to the function. This option is only meaningful if you pass the **-Ointerproc** or **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipsmallinlining**)
- **Off** (**-Onoipsmallinlining**)

IP Constant Propagation

If a parameter of a function is always set to the same constant value by callers, this optimization treats uses of that parameter in that function as a constant instead of setting the parameter in callers. This option is only meaningful if you pass the **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipconstprop**)
- **Off** (**-Onoipconstprop**)

IP Remove Parameters

When certain parameters are not required, the parameters are removed, and function calls do not pass arguments for them. This option is only meaningful if you pass the **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipremoveparams**)
- **Off** (**-Onoipremoveparams**)

IP Limit Inlining

Limits the amount of inlining performed by interprocedural optimizations. This option is only meaningful if you pass the **-Ointerproc** or **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oiplimitinlining**)
- **Off** (**-Onoiplimitinlining**)

IP Constant Returns

When functions return constant values, those values can be propagated to the caller. This option is only meaningful if you pass the **-Ointerproc** or **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipconstantreturns**)
- **Off** (**-Onoipconstantreturns**)

IP Remove Returns

When the result of a function call is not required, the return statement is removed. This option is only meaningful if you pass the **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipremovereturns**)
- **Off** (**-Onoipremovereturns**)

IP Alias Reads

Determines what memory can be read by function calls. This option is only meaningful if you pass the **-Ointerproc** or **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipaliasreads**)
- **Off** (**-Onoipaliasreads**)

IP Alias Writes

Determines what memory can be written by function calls. This option is only meaningful if you pass the **-Ointerproc** or **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipaliaswrites**)
- **Off** (**-Onoipaliaswrites**)

IP Alias Lib Functions

Performs interprocedural alias analysis on some common library functions such as `printf`, `strlen`, and `fopen`. This option is only meaningful if you pass the **-Ointerproc** or **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipaliaslibfuncs**)
- **Off** (**-Onoipaliaslibfuncs**)

Individual Functions

This option category is contained within **Optimization**.

These options allow you to specify individual functions for optimization.

Optimize Specific Functions for Speed

Enables speed optimizations for specific functions. The equivalent driver option is:

- **--fast=*function1*[,*function2*…]**

Note that this option has no effect unless an overall **Optimization→Optimization Strategy** is specified.

Optimize Specific Functions for Size

Enables size optimizations for specific functions. The equivalent driver option is:

- **--small=*function1*[,*function2*…]**

Note that this option has no effect unless an overall **Optimization→Optimization Strategy** is specified.

Inline Specific Functions

Enables two-pass inlining for specific functions. If not optimization strategy is selected, this option enables **-Ospeed**. The equivalent driver option is:

- **-OI=*function1*[,*function2*...]**

For more information, see “Inlining Optimizations” on page 294.

Loop Optimize Specific Functions

Enables loop optimizations for individual *functions*. If no optimization strategy is selected, this option enables **-Ospeed**. The equivalent driver option is:

- **-OL=*function1*[,*function2*...]**

For more information, see “Loop Optimizations” on page 315.

Debugging Options

This is a top-level option category.

These options control the generation of debugging and profiling information. For more information, see “Generating Debugging Information” on page 49 and “Obtaining Profiling Information” on page 57.

For additional, more specialized options, see “Debugging Options” on page 271.

Debugging Level

Permitted settings for this option are:

- **MULTI** (**-G**) — Generates source-level debugging information, and allows procedure calls from the MULTI Debugger's command line.
- **Plain** (**-g**) — Generates source-level debugging information, but does not allow procedure calls.
- **Stack Trace** (**-gs**) — Generates minimal debugging information sufficient for MULTI to perform stack traces, but not sufficient for source-level debugging. The Green Hills libraries are compiled with this option.
- **None** (**--no_debug**) — [default] Generates no debugging information.

Note

The **-gs** option enables **-gtws** by default. To override this setting, use the **-nogtws** option. For more information, see “Generate Target-Walkable Stack” on page 271.

Profiling - Block Coverage

Controls *block coverage* profiling, which records a flag or a count for each basic block in your program. A basic block is a set of instructions for which if the first instruction is executed, the remaining instructions in the block must also be executed. Permitted settings for this option are:

- **Flag** (**-coverage=flag**) — Records whether or not each block is executed.

- **Count (32-bit counters)** (**-coverage=count**) — Records how many times each block is executed using a 32-bit counter.
- **Off** (**-coverage=none**) — [default]

Block coverage profiling does not use locking or an atomic increment to achieve thread safety. If you use count profiling with multi-threaded code or in interrupt handlers a block's counter may be inaccurate if two increments occur simultaneously. Flag profiling is thread-safe.

For more information, see "Enabling Instrumented Coverage or Performance Profiling" on page 58.

Profiling - Entry/Exit Logging

For targets that support function entry/exit (FEE) logging, controls whether or not the compiler instruments code for FEE logging. Permitted settings for this option are:

- **Off** (**-no_gen_entry_exit_log**) — [default] FEE logging instructions are not generated by the compiler.
- **On** (**-gen_entry_exit_log**) — FEE logging instructions are generated by the compiler in the prologue and epilogue of each logged function.

For more information about FEE logging, see "Logging Function Entry and Exit (FEE) Events" on page 45.

Profiling - Entry/Exit Logging with Arguments

For targets that support function entry/exit (FEE) logging, controls whether or not the compiler instruments code for FEE logging with function arguments. Permitted settings for this option are:

- **Off** (**-no_gen_entry_exit_arg_log**) — [default] FEE logging instructions are not generated by the compiler.
- **On** (**-gen_entry_exit_arg_log**) — FEE logging instructions are generated by the compiler in the prologue and epilogue of each logged function.

For more information about FEE logging, see "Logging Function Entry and Exit (FEE) Events" on page 45.

Profiling - Entry/Exit Linking

For targets that support function entry/exit (FEE) logging, controls whether or not FEE logging instructions are replaced, stripped, or left in place by the linker. Permitted settings for this option are:

- **Enable** (**-enable_entry_exit_log**) — FEE logging instructions are not replaced or removed by the linker.
- **Disable** (**-disable_entry_exit_log**) — [default] FEE logging is disabled by the linker, and logging instructions are replaced by `nop` instructions.
- **Strip** (**-strip_entry_exit_log**) — FEE logging is disabled by the linker, and logging instructions are stripped from the executable by the linker.

For more information about FEE logging, see "Logging Function Entry and Exit (FEE) Events" on page 45.

Profiling - Strip EAGLE Logging

For targets that support EAGLE logging, controls the inclusion of logging code at link time. Permitted settings for this option are:

- **Off** (**-enable_eagle_log**) — [default] Do not disable or remove EAGLE logging code.
- **On** (**-strip_eagle_log**) — Disables and removes the logging code from the final executable at link time.

Profiling - Target-Based Timing

Controls *timing* profiling, which records the time spent in each function. You should only use this option in stand-alone and u-velOSity projects. Permitted settings for this option are:

- **On** (**-timer_profile**)
- **Off** (**-no_timer_profile**) — [default]

Enabling target-based timing profiling requires that timer code be present and running on the target. For more information, see "Enabling Sampled Profiling" on page 63.

Note

To obtain PC samples for a Linux native application, you must link with the **libmulti.a** library.

Run-Time Error Checks

Controls run-time error checking, which can be helpful during a debugging session, but which increases the size and reduces the speed of your application. Run-time error checks catch many occurrences of the common errors described below. The syntax for this option is:

-check=*type*`[,`*type*`]`...

-check=*type* [**-check=***type*]...

Enables run-time error checks for each specified *type*.

-check=`all[,no`*type*`]`...

-check=`all` [**-check=**`no`*type*]...

Enables all run-time error checks. To enable all checks except those of a certain type, follow this option with `,no`*type* or `-check=no`*type*.

Run-time error checks are also enabled by MISRA 2004 rule 12.8 and 21.1, and MISRA 1998 rules 4, 38, and 107. The various checks generate warnings or errors as specified below. Permitted values for *type* are:

- **All** (**-check=all**) — Enables all run-time error checks. This option does not affect run-time memory checks.
- **None** (**-check=none**) — [default] Disables all run-time error checks. This option does not affect run-time memory checks.
- **Assignment Bounds** (**-check=assignbound**) — Generates a warning if a value assigned to an integral or enumerated object is out-of-bounds for the object being assigned to. No warning is issued when an assignment is made to a

sparsely populated `enum` if the assignment is within the range of the `enum`, but is not one of the defined values. For example:

```
enum E { one=1, three=3 } e;

void func(void)
{
    e = (enum E)0; /* warning */
    e = (enum E)1; /* no warning */
    e = (enum E)2; /* no warning */
    e = (enum E)3; /* no warning */
    e = (enum E)4; /* warning */
}
```

- **Array Bounds** (**-check=bounds**) — Generates an error if an array is accessed with an invalid index. Only arrays whose bounds are known at compile time are checked; incomplete and variable-length arrays (VLAs) are not checked. The compiler checks each individual array index separately. For example:

  ```
  int a[2][3];
  a[0][3] = 0; // produces a run-time error
  a[1][0] = 0; // equivalent behavior, no run-time error
  ```

  This option does not output a run-time check for the following example, because the `arr` parameter's type decays into `int*` and could legitimately point to an array of any size:

  ```
  int func(int arr[20], int i)
  {
      return arr[i];
  }
  ```

- **Case Label Bounds** (**-check=switch**) — Generates a warning if the `switch` expression does not match any of the `case` labels. This does not apply when a default `case` label is used.
- **Divide by Zero** (**-check=zerodivide**) — Generates an error indicating that a divide by zero occurred and then terminates the program. This option instruments both integer and floating-point divides.
- **Nil Pointer Dereference** (**-check=nilderef**) — Generates an error for dereferences of null pointers.

- **Write to Watchpoint** (**-check=watch**) — Instruments your code to watch writes to one arbitrary address in memory. Use this check when your target does not support hardware breakpoints. Specify the watchpoint address at run time with the **watchpoint** command. For more information about watchpoints, see the documentation about breakpoint commands in the *MULTI: Debugging Command Reference* book.

To enable or disable all the checks when using the Builder, use the **All** toggle switch at the bottom of the dialog box. When using the driver, pass the option **-check=all** or **-check=none**.

Note

-check=assignbound and **-check=switch** produce run-time errors on code that is legal in C and C++, but may be considered to have bad style.

Run-Time Memory Checks

Controls run-time memory checking. Note that this option is not available for projects built with PIC or PID. Permitted settings for this option are:

- **General (Allocations Only)** (**-check=alloc**) — Selects a special version of memory allocation library routines, including `malloc` and `free`. These library routines detect a limited number of memory allocation problems, such as freeing an object that has not been allocated. This option only changes the way the program is linked and does not change the code produced by the compiler.
- **Intensive** (**-check=memory**) — Introduces additional code in the user application at each dereference of a pointer to detect a wide range of memory allocation problems, including those detected by **-check=nilderef**. This setting implies **-check=alloc** because the special library routines selected by that option are required. To perform full memory checking on a small part of an application, specify **-check=memory** when compiling the files or files of interest, and specify **-check=alloc** when linking the entire program.
- **None** (**-check=nomemory**) — [default] Disables memory checking.

Both forms of memory checking require a frame pointer to walk the stack at run time, and imply setting **Advanced→Debugging Options→Force Frame Pointer** to **On**.

For detailed information, see “Enabling General Memory Allocation Checking” on page 65, “Enabling Intensive Memory Allocation Checking” on page 66, and the documentation about viewing memory allocation information in the *MULTI: Debugging* book.

Data Trace

This option category is contained within **Debugging**.

Preprocessor Options

This is a top-level option category.

These options control the preprocessor. For information about the Green Hills predefined macros, see "Predefined Macro Names" on page 734.

For additional, more specialized options, see "Preprocessor Options" on page 278.

Define Preprocessor Symbol

Defines a preprocessor *symbol*, and optionally sets it to a *value*. When setting this option in the **Edit List Option** window in the Project Manager, type `symbol=value` in the text field. The equivalent driver option is:

- **-D*symbol*[=*value*]**

This is equivalent to placing the following at the top of each source file:

```
#define symbol [value]
```

`-Dsymbol` is equivalent to `-Dsymbol=1`

Undefine Preprocessor Symbol

Undefines a preprocessor *symbol*. The equivalent driver option is:

- **-U*symbol***

This is equivalent to placing the following at the top of each source file:

```
#undef symbol
```

Display includes Preprocessor Directives Listing

Controls the display to standard error of a list of files opened by a `#include` directive during normal compilation. Files are compiled and linked normally.

Permitted settings for this option are:

- **On** (**-H**)
- **Off** (**--no_trace_includes**) — [default]

C/C++ Compiler Options

This is a top-level option category.

These options control various aspects of compilation that relate directly to the C and C++ languages.

For additional, more specialized options, see "C/C++ Compiler Options" on page 281.

C Language Dialect

Controls the version of C to be accepted by the compiler. Permitted settings for this option are:

- **Strict ISO C99 (-C99)** — Specifies strict ISO C99. This mode is compliant with ISO/IEC 9899:1999 and does not allow any non-standard constructs.
- **ISO C99 (-c99)** — Specifies ISO C99. This mode provides all features of the C99 language in addition to some extensions. Some non-standard constructs produce warnings rather than errors.
- **Strict ANSI C (-ANSI)** — Specifies strict ANSI. This mode is compliant with the ANSI X3.159-1989 standard and does not allow any non-standard constructs.
- **ANSI C (-ansi)** — [default] Specifies ANSI C with extensions. This mode extends the ANSI X3.159-1989 standard with certain useful and compatible constructs. For details, see "ANSI C" on page 623.

  For details about Green Hills C and the differences between the various modes, see Chapter 12, "Green Hills C" on page 621.
- **GNU C (-gcc)** — Specifies full GNU mode, supporting GNU features (including zero size arrays, multi-line string constants, and `inline` functions) which are less compatible with ANSI C. For details, see "GNU C" on page 630.
- **GNU C99 (-gnu99)** — Specifies a GNU-compatible version of C99 mode, supporting many GNU features not explicitly included in ISO C99.
- **K+R C (-k+r)** — Support for K&R C is deprecated and may be removed in future versions of MULTI. Specifies the C version documented in *Kernighan & Ritchie, First Edition*, which is somewhat compatible with the portable C compiler (PCC). In this mode, the compiler will accept ANSI-style prototypes. For details, see "K&R C" on page 638.

C Japanese Automotive Extensions

Controls support for the Japanese Automotive C extensions. Permitted settings for this option are:

- **On** (**-japanese_automotive_c**)
- **Off** (**-no_japanese_automotive_c**) — [default]

For more information, see "Japanese Automotive C Extensions" on page 643.

C++ Language Dialect

Specifies the version of C++ to be accepted by the compiler. Permitted settings for this option are:

- **Standard C++ (Violations Give Errors)** (**--STD**) — Specifies C++ Strict Standard Mode, which gives errors when non-Standard features are used and disables features that conflict with Standard C++. For more information, see "Standard C++" on page 692.
- **Standard C++ (Violations Give Warnings)** (**--std**) — [default] Specifies C++ Standard Mode, which gives warnings when non-Standard features are used and disables features that conflict with Standard C++. For more information, see "Standard C++" on page 692.
- **Standard C++ with ARM Extensions** (**--arm**) — Specifies Standard C++ with extensions, which extends the standard with many useful and compatible constructs. For more information, see "Standard C++ with ARM Extensions" on page 698.
- **GNU C++** (**--g++**) — Specifies GNU C++ mode. For more information, see "GNU C++" on page 699.
- **Embedded C++** (**--e**) — Specifies Embedded C++. For more information, see "Embedded C++" on page 695.
- **Extended Embedded C++** (**--ee**) — Specifies Extended Embedded C++, which adds templates, namespaces, mutable new-style casts, and the Standard Template Library (STL) to Embedded C++. For more information, see "Extended Embedded C++" on page 697.

C++ Libraries

Specifies the type of C++ libraries to use. Permitted settings for this option are:

- **Standard C++ Library without Exceptions** (**--stdl**)
- **Standard C++ Library with Exceptions** (**--stdle**)
- **Extended Embedded C++ Library without Exceptions** (**--eel**)
- **Extended Embedded C++ Library with Exceptions** (**--eele**)
- **Embedded C++ Library without Exceptions** (**--el**)
- **Embedded C++ Library with Exceptions** (**--ele**)

The default is to use libraries equivalent to your **C/C++ Compiler→C++ Language Dialect** without exceptions, and you may not specify a library that is more full-featured than your **C/C++ Compiler→C++ Language Dialect**. For more information, see "Specifying C++ Libraries" on page 691.

C++ Exception Handling

Controls support for exception handling. Permitted settings for this option are:

- **On** (**--exceptions**) — Enables support for exception handling. Code size and speed may be impacted even when exception handling is not directly used.
- **Off** (**--no_exceptions**) — [default] Disables support for exception handling.

Allow C++ Style Slash Comments in C

Controls treatment of C++ style `//` comments. Permitted settings for this option are:

- **On** (**--slash_comment**) — [default] C++ style `//` comments are accepted.
- **Off** (**--no_slash_comment**) — C++ style `//` comments are not accepted and generate errors.

ANSI Aliasing Rules

Controls assumptions based on ANSI aliasing rules in the compiler. Permitted settings for this option are:

- **On** (**-ansi_alias**) — [default] Enables ANSI aliasing rules.
- **Off** (**-no_ansi_alias**) — Disables ANSI aliasing.

When this option is enabled, the compiler assumes that memory may only be accessed through pointers of compatible types. This allows the compilers to generate better code. Illegal casts may cause the compiler to generate code that does not behave as expected. Consider the following example:

```
int inc_and_get(int *a, short *b)
{
  (*a)++;
  return *b;
}

int loc()
{
  int var = 12345;
  /* the dereference of *b inside of inc_and_get illegally aliases var */
  return inc_and_get(&var, (short *)&var);
}
```

This may behave unexpectedly because an `int` type may not be used to access an object of the `short` type.

Disabling this option suppresses diagnostics 1518, 1519, and 1798.

MISRA C 2004

This option category is contained within **C/C++ Compiler**.

The MISRA C 2004 Standard was created by the Motor Industry Software Reliability Association (MISRA). It is a set of guidelines for the C programming language that is designed to enforce good practices in the development of embedded automotive systems. You can find complete details in the *Guidelines For The Use Of The C Language In Critical Systems, Motor Industry Software Reliability Association, October 2004* book. For more information, see the MISRA Web site [http://www.misra.org.uk/].

There are 141 rules, classified into two categories, as follows:

- 120 *required* rules (designated **[R]** below) — mandatory requirements placed on the programmer that will, by default, generate an error if enabled and breached.
- 21 *advisory* rules (designated **[A]** below) — suggestions to the programmer that will, by default, generate a warning if enabled and breached.

The MISRA rules are organized into 21 groups, each focusing on a particular C language topic (for example: types, functions, expressions, the C preprocessor, etc.). For information about each group and individual rules, see "MISRA C 2004" on page 168.

Enforcing MISRA C 2004 Rules

A single driver option allows you to enable any or all of the MISRA rules. It may be used in any of the following forms (where *n* is a rule number):

- **--misra_2004=all** — Enables checking of all the rules.
- **--misra_2004=none** — Disables checking of all the rules.
- **--misra_2004=*n*[,*n1*…]** — Enables checking of a comma-separated list of rules.
- **--misra_2004=-*n*[,-*n1*...]** — Disables checking of a comma-separated list of rules.
- **--misra_2004=*n*-*n1*** — Enables checking of rules *n* through *n1*.
- **--misra_2004=-*n*-*n1*** — Disables checking of rules *n* through *n1*.

For a list of all supported MISRA C 2004 rules, see "MISRA C 2004" on page 168.

When the **--diag_*=** options and **--misra_2004=** options are both used in a single project, both options are passed to the compiler and both are used.

The compiler first uses the **--misra_2004=** options to determine whether certain MISRA behavior is enabled. If a MISRA rule is not enabled based on the **--misra_2004=** options, the compiler applies the **--diag_*=** options to determine the severity of the message.

If a MISRA rule is enabled based on the **--misra_2004=** options, the compiler follows MISRA-specific logic to determine if a message should be given, and then usually applies the **--diag_*=** options to determine the severity and suppression of the message. It is not always possible to change the severity of a MISRA rule using **--diag_*=**.

Coding standard profiles, such as Green Hills Standard Mode, are implemented using the same mechanism as **--diag_*=** and therefore have a similar interaction with **--misra_2004=**.

The following example demonstrates the use of these MISRA options:

Example 3.1. Using MISRA Builder and Driver Options

--misra_2004=all,-2.3 — Enables checking of all rules except rule 2.3.

--misra_2004=2.3-4.2 — Enables checking of rules 2.3 through 4.2.

--misra_2004=2.3,4.2 — Enables checking of rules 2.3 and 4.2.

--misra_2004=-2.3,-4.2 — Disables checking of rules 2.3 and 4.2.

MISRA C 2004 Implementation

The Green Hills tools provide options to control the severity of messages that alert you to violations of MISRA C 2004 rules. Most of the messages are given at compile time, although some are generated at run-time. Because run-time checking can adversely affect code performance, an option is provided to disable it.

Most of the rules that generate messages at compile time are implemented though compiler checks that are only enabled when the corresponding MISRA rule is enabled. Some of the rules specify constraints that the compiler already has checks for. In these cases, the severity of the diagnostic message will be elevated to the severity specified for the corresponding MISRA rule if it exceeds the default severity for that message.

In some cases a MISRA rule has been implemented using a number of distinct compile time checks that each generate unique diagnostic messages when they are violated. This means that one item that violates one MISRA rule might cause multiple diagnostics to be issued if it violates more than one of the checks used to implement

the rule. The advantage of this approach is that by modifying the severity of or suppressing individual diagnostic messages, the user can allow a deviation from a portion of a MISRA rule without disabling the entire rule. For example, if you have **Rule 8.1** enabled but want to allow static functions to be defined without a separate prototype declaration, you can specify **--diag_remark=1828** or **--diag_suppress=1828** to disable the portion of **Rule 8.1** that requires static function definitions to have separate prototype declarations.

In the introduction to section 6.15, "Switch statements," MISRA-C:2004 defines a restricted switch statement syntax. Some of the syntax restrictions are enforced as part of the rules contained in section 6.15. The remainder of the syntax restrictions are enforced by the compiler by default if any of the rules from section 6.15, 15.1-15.5, are enabled. Because all of the rules in section 6.15 are required rules, the diagnostic severity for the restricted switch statement syntax is set to the required rule severity level by default. The compiler diagnostics issued for these syntax restrictions include 1705, 1706, 1708, 1709, and 1821.

The compiler toolchain enforces some MISRA rules by disabling features or changing logic that is not specific to MISRA rule checking. If you enable checking for one of these rules, and your code violates that rule, you will receive an error regardless of the specified MISRA diagnostic level. Because the compiler toolchain does not issue the error due to the MISRA rule directly, the error message will not contain a reference to that rule. These MISRA rules are not affected by `#pragma ghs startnomisra` or `#pragma ghs endnomisra` (see “Green Hills Extension Pragma Directives” on page 753).

For example, you might enable checking for MISRA required rule 1.1, which enables strict ANSI C by disabling recognition of certain non-standard keywords. Because the compiler can no longer recognize those keywords, it cannot parse code that violates the rule. As a result, the compiler will issue a non-discretionary error even if you have used `--misra_req=silent` to downgrade and hide MISRA required rule errors.

Some MISRA rules are enforced through preprocessor directives in the standard headers provided by Green Hills and are only enforced when the standard headers are used.

Some MISRA rules change other command line controllable options as documented in the following section. The MISRA rules override explicit settings of the other

command line options, even if these options appear later on the command line than the MISRA option.

MISRA C 2004 Rules - 1 Environment

- **1.1 [R] ISO 9899:1990 C conformance w/o extensions** (**--misra_2004=1.1**) — Enables Strict ANSI C. This option enables **-ANSI**, which disables recognition of many non-standard keywords and extensions. Errors issued because of this check are not identified as MISRA errors, and their severities are unaffected by MISRA classification or the **--misra_req** option.
- **1.2 [R] No dependencies can be placed on undefined or unspecified behavior** (**--misra_2004=1.2**) — Not enforced.
- **1.3 [R] Code other than C must conform to the standard interface** (**--misra_2004=1.3**) — Not enforced.
- **1.4 [R] Compiler and linker verified to support 31 significant case-sensitive characters for identifiers** (**--misra_2004=1.4**) — Green Hills tools have no set limit by default.
- **1.5 [A] Floating-point implementation is standard-compliant** (**--misra_2004=1.5**) — The Green Hills compilers use the IEEE floating-point representation. This rule cannot be disabled.

MISRA C 2004 Rules - 2 Language Extensions

- **2.1 [R] Inline assembly only in functions or macros with no other code** (**--misra_2004=2.1**) — Enforced.

  Associated diagnostics include 1749 and 1839.
- **2.2 [R] Use of /*...*/ comments only** (**--misra_2004=2.2**) — Enforced.

  Associated diagnostics include 1876.
- **2.3 [R] No nested comments** (**--misra_2004=2.3**) — Enforced.

  Associated diagnostics include 9.
- **2.4 [A] No `commented out' sections of code** (**--misra_2004=2.4**) — Does not allow the following characters inside of a comment:
  - A semicolon followed by by a new-line character.

- Open or closed curly braces.

Associated diagnostics include 1771.

MISRA C 2004 Rules - 3 Documentation

- **3.1 [R] Documentation of implementation-defined behavior required** (**--misra_2004=3.1**) — Not enforced.
- **3.2 [R] Values of char types restricted to subset of ISO 10646-1** (**--misra_2004=3.2**) — All Green Hills compilers use the ASCII standard for mapping character sets to numeric values. This property cannot be disabled.
- **3.3 [A] Implementation of integer division determined and documented** (**--misra_2004=3.3**) — This behavior depends on the hardware configuration of the target. Most Green Hills compilers, however, round the division result toward 0 (including Power Architecture, MIPS, TriCore, and x86).
- **3.4 [R] Uses of #pragma documented and explained** (**--misra_2004=3.4**) — This book lists all `#pragma` constructs, their usages, and effects (see "General Pragma Directives" on page 750).
- **3.5 [R] Documentation of implementation-defined behavior and packing of bitfields required** (**--misra_2004=3.5**) — Not enforced.
- **3.6 [R] All library production code conforms to MISRA** (**--misra_2004=3.6**) — Not enforced.

MISRA C 2004 Rules - 4 Character Sets

- **4.1 [R] Only ISO C escape sequences used** (**--misra_2004=4.1**) — Enforced.

  Associated diagnostics include 192 and 1823.
- **4.2 [R] No trigraphs** (**--misra_2004=4.2**) — Enforced.

  Associated diagnostics include 1695.

MISRA C 2004 Rules - 5 Identifiers

- **5.1 [R] No more than 31 chars to determine an identifier** (**--misra_2004=5.1**) — Enforced by truncating all identifiers at 31 characters.

Associated diagnostics include 1772.

- **5.2 [R] No use of same id name in inner and outer scope** (**--misra_2004=5.2**) — Enforced.

  Associated diagnostics include 460 and 1721.

- **5.3 [R] Each `typedef' must be a unique identifier** (**--misra_2004=5.3**) — Enforced within a translation unit.

  Associated diagnostics include 1722 and 1841.

- **5.4 [R] Tag names must be unique** (**--misra_2004=5.4**) — Enforced within a translation unit.

  Associated diagnostics include 188, 469, and 1842.

- **5.5 [A] No reuse of function or static object identifiers** (**--misra_2004=5.5**) — Enforced within a translation unit.

  Associated diagnostics include 1843.

- **5.6 [A] No identifiers with the same name in different name spaces except for struct and union members** (**--misra_2004=5.6**) — Enforced within a translation unit.

  Associated diagnostics include 1723.

- **5.7 [A] No reuse of identifiers** (**--misra_2004=5.7**) — Enforced within a translation unit.

  Associated diagnostics include 1840.

MISRA C 2004 Rules - 6 Types

- **6.1 [R] `char' only used for storage and use of character values** (**--misra_2004=6.1**) — Enforced.

  Associated diagnostics include 1850 and 1852.

- **6.2 [R] `signed char' and `unsigned char' only used for storage and use of numeric values** (**--misra_2004=6.2**) — Enforced.

  Associated diagnostics include 1851.

- **6.3 [A] Basic types used only in `typedef's and bitfields** (**--misra_2004=6.3**) — Enforced.

  Associated diagnostics include 1698.
- **6.4 [R] Bitfields can only have `unsigned int' or `signed int' types** (**--misra_2004=6.4**) — Enforced.

  Associated diagnostics include 230 and 1717.
- **6.5 [R] Signed bitfields must be least 2 bits long** (**--misra_2004=6.5**) — Enforced.

  Associated diagnostics include 108.

MISRA C 2004 Rules - 7 Constants

- **7.1 [R] No non-zero octal constants or octal escape sequences** (**--misra_2004=7.1**) — Enforced.

  Associated diagnostics include 1718.

MISRA C 2004 Rules - 8 Declarations and Definitions

- **8.1 [R] Functions must always have prototype declarations** (**--misra_2004=8.1**) — Enforced.

  Associated diagnostics include 223, 1547, 1791, 1800, and 1828.
- **8.2 [R] Every function must have an explcit return type** (**--misra_2004=8.2**) — Enforced.

  Associated diagnostics include 77, 260, and 938.
- **8.3 [A] Function declaration and definition prototypes match** (**--misra_2004=8.3**) — Enforced.

  Associated diagnostics include 1844.
- **8.4 [R] Multiple declarations of an object or function must be compatible** (**--misra_2004=8.4**) — Enforced within a translation unit.

  Associated diagnostics include 147.

- **8.5 [R] No object or function definitions in a header file** (**--misra_2004=8.5**) — Enforced.

  Associated diagnostics include 1846.
- **8.6 [R] Functions always declared at file scope** (**--misra_2004=8.6**) — Enforced.

  Associated diagnostics include 1731.
- **8.7 [R] Use function or block scope definitions for objects whenever possible** (**--misra_2004=8.7**) — Enforced for non-static variables in **elxr**. If you enable **-auto_sda**, this rule might not be enforced.
- **8.8 [R] External objects and functions declared in no more than one file** (**--misra_2004=8.8**) — Not enforced.
- **8.9 [R] Only one external definition for an external identifier** (**--misra_2004=8.9**) — Enforced in the compiler and **elxr**.

  Associated diagnostics include 1797.
- **8.10 [R] Static linkage of file scope declarations when possible** (**--misra_2004=8.10**) — Enforced in `elxr` .
- **8.11 [R] All objects and functions with internal linkage declared `static'** (**--misra_2004=8.11**) — Enforced.

  Associated diagnostics include 172.
- **8.12 [R] Arrays with external linkage must have known size at compile time** (**--misra_2004=8.12**) — Enforced.

  Associated diagnostics include 1824.

MISRA C 2004 Rules - 9 Initialization

- **9.1 [R] Automatic variables initialized before used** (**--misra_2004=9.1**) — Enforced for simple cases.

  Associated diagnostics include 549.
- **9.2 [R] Braces used to match structure in initialization of arrays or structs** (**--misra_2004=9.2**) — Enforced.

Associated diagnostics include 146, 991, and 1736.

- **9.3 [R] All or only first member of an enumeration may be explicitly initialized** (**--misra_2004=9.3**) — Enforced.

  Associated diagnostics include 1726.

MISRA C 2004 Rules - 10 Arithmetic Type Conversions

- **10.1 [R] Restrict implicit conversions for integer type expressions** (**--misra_2004=10.1**) — Enforced.

  Associated diagnostics include 1863, 1864, 1865, 1866, 1867, 1868, 1869, and 1870.
- **10.2 [R] Restrict implicit conversions for floating type expressions** (**--misra_2004=10.2**) — Enforced.

  Associated diagnostics include 1871, 1872, 1873, 1874, and 1875.
- **10.3 [R] Restrict explicit casts for integer type expressions** (**--misra_2004=10.3**) — Enforced.

  Associated diagnostics include 1847, 1878, 1879, and 1880.
- **10.4 [R] Restrict explicit casts for floating type expressions** (**--misra_2004=10.4**) — Enforced.

  Associated diagnostics include 1848.
- **10.5 [R] Bitwise `~' and `<<' expressions on unsigned char or unsigned short types must be cast to underlying type** (**--misra_2004=10.5**) — Enforced.

  Associated diagnostics include 1849.
- **10.6 [R] Apply `U' suffix to all constants of `unsigned' type** (**--misra_2004=10.6**) — Enforced.

  Associated diagnostics include 1775.

MISRA C 2004 Rules - 11 Pointer Type Conversions

- **11.1 [R] No conversions between pointer to function and non-integral types** (**--misra_2004=11.1**) — Enforced.

  Associated diagnostics include 1832.
- **11.2 [R] No conversions between pointer to object and any type other than integral, pointer to object, or pointer to void** (**--misra_2004=11.2**) — Enforced.

  Associated diagnostics include 1833.
- **11.3 [A] No casting between pointer and integral types** (**--misra_2004=11.3**) — Enforced.

  Associated diagnostics include 1834 and 1877.
- **11.4 [A] No casting between different pointer to object types** (**--misra_2004=11.4**) — Enforced.

  Associated diagnostics include 1835.
- **11.5 [R] No casting that removes any `const' or `volatile' qualification from the type addressed by a pointer** (**--misra_2004=11.5**) — Enforced.

  Associated diagnostics include 1836.

MISRA C 2004 Rules - 12 Expressions

- **12.1 [A] Limited dependence on C precedence rules** (**--misra_2004=12.1**) — Enforced per the guidelines in the supporting text of the rule.

  Associated diagnostics include 1737.
- **12.2 [R] No expressions with values dependent on evaluation order** (**--misra_2004=12.2**) — Not enforced.
- **12.3 [R] No side effects in the operand of `sizeof'** (**--misra_2004=12.3**) — Enforced.

  Associated diagnostics include 1725.
- **12.4 [R] No side effects in the right hand operand of `&&' or `||'** (**--misra_2004=12.4**) — Enforced.

  Associated diagnostics include 1724.

- **12.5 [R] Operands of `&&' and `||' must be primary expressions** (**--misra_2004=12.5**) — Enforced.

  Associated diagnostics include 1729.
- **12.6 [A] Operands of logical operators must be Boolean, and Boolean expressions may not be used as operands of other operators** (**--misra_2004=12.6**) — Enforced.

  Associated diagnostics include 1853 and 1854.
- **12.7 [R] No bitwise operations on signed integer types** (**--misra_2004=12.7**) — Enforced.

  Associated diagnostics include 1730.
- **12.8 [R] Value of right hand operand of a shift operator must be equal to or greater than zero and less than the size of the underlying type of the left hand operand** (**--misra_2004=12.8**) — Enforced. Both compile-time and run-time checks are performed. The run-time check is disabled when the **--no_misra_runtime** option is passed. The underlying type of the constant `1` is `signed char`, therefore the expression `(1<<x)` is valid only for `x<8`.

  Associated diagnostics include 62 and 63.
- **12.9 [R] No unary minus on unsigned expressions** (**--misra_2004=12.9**) — Enforced.

  Associated diagnostics include 1702.
- **12.10 [R] No comma operators** (**--misra_2004=12.10**) — Enforced.

  Associated diagnostics include 1825.
- **12.11 [A] No wrap-around in constant unsigned expression evaluation** (**--misra_2004=12.11**) — Enforced.

  Associated diagnostics include 1735.
- **12.12 [R] No use of underlying bit representation in floating point expressions** (**--misra_2004=12.12**) — The Green Hills compilers do not allow any use of bit-wise operators (such as `&`, `|`, `^`, `~`, etc.) by default. When this rule is enabled, a diagnostic is issued for union members of floating-point type.

  Associated diagnostics include 1820.
- **12.13 [A] No mixing of increment (++) and decrement (--) operators with other operators** (**--misra_2004=12.13**) — Enforced.

Associated diagnostics include 1856.

MISRA C 2004 Rules - 13 Control Statement Expressions

- **13.1 [R] Assignment operators not used in Boolean expressions** (**--misra_2004=13.1**) — Enforced.

  Associated diagnostics include 1738.

- **13.2 [A] Explicit test of a value against zero unless the expression is Boolean** (**--misra_2004=13.2**) — Enforced.

  Associated diagnostics include 1855 and 1881.

- **13.3 [R] Floating-point values not tested for (in)equality** (**--misra_2004=13.3**) — Enforced.

  Associated diagnostics include 1704.

- **13.4 [R] No floating-point variables in `for' loop control expressions** (**--misra_2004=13.4**) — Enforced.

  Associated diagnostics include 1750.

- **13.5 [R] Only loop control expression in `for' statement header** (**--misra_2004=13.5**) — Not enforced.

- **13.6 [R] No modification of numeric control variables in `for' loop body** (**--misra_2004=13.6**) — Not enforced.

- **13.7 [R] No Boolean operations with invariant results** (**--misra_2004=13.7**) — Enforced.

  Associated diagnostics include 1857.

MISRA C 2004 Rules - 14 Control Flow

- **14.1 [R] No unreachable code** (**--misra_2004=14.1**) — Enforced.

  Associated diagnostics include 111 and 177.

- **14.2 [R] All non-null statements must have a side-effect** (**--misra_2004=14.2**) — Enforced.

Associated diagnostics include 174.

- **14.3 [R] Null statement must occur on a line by itself** (**--misra_2004=14.3**) — Enforced.

  Associated diagnostics include 1746.

- **14.4 [R] No `goto' statements** (**--misra_2004=14.4**) — Enforced.

  Associated diagnostics include 1705.

- **14.5 [R] No `continue' statements** (**--misra_2004=14.5**) — Enforced.

  Associated diagnostics include 1706.

- **14.6 [R] At most one break statement per iteration statement** (**--misra_2004=14.6**) — Enforced.

  Associated diagnostics include 1845.

- **14.7 [R] Functions must have a single point of exit** (**--misra_2004=14.7**) — Enforced.

  Associated diagnostics include 1734.

- **14.8 [R] Dependent statements of loop and switch statements must have braces** (**--misra_2004=14.8**) — Enforced.

  Associated diagnostics include 1709.

- **14.9 [R] Dependent statements of if statements must have braces** (**--misra_2004=14.9**) — Enforced.

  Associated diagnostics include 1826.

- **14.10 [R] All `if'...`else if' constructs must have an `else'** (**--misra_2004=14.10**) — Enforced.

  Associated diagnostics include 1710.

MISRA C 2004 Rules - 15 Switch Statements

- **15.1 [R] Switch labels must be at top level compound statement of `switch' statement** (**--misra_2004=15.1**) — Enforced.

  Associated diagnostics include 1838.

- **15.2 [R] Every non-empty switch clause terminates with an unconditional `break' statement** (**--misra_2004=15.2**) — Enforced.

  Associated diagnostics include 1711 and 1884.

- **15.3 [R] Every `switch' statement must contain a final `default' clause** (**--misra_2004=15.3**) — Enforced.

  Associated diagnostics include 1712 and 1837.

- **15.4 [R] No Boolean values in `switch' expressions** (**--misra_2004=15.4**) — Enforced.

  Associated diagnostics include 1733.

- **15.5 [R] `switch' statements must have at least one `case'** (**--misra_2004=15.5**) — Enforced.

  Associated diagnostics include 1713.

MISRA C 2004 Rules - 16 Functions

- **16.1 [R] No variable argument function definitions** (**--misra_2004=16.1**) — Enforced.

  Associated diagnostics include 1697.

- **16.2 [R] No direct or indirect recursion** (**--misra_2004=16.2**) — Enforced in the compiler and `elxr` linker. Using function pointers might cause the linker to miss certain recursive functions, or report errors for nonrecursive functions. Legacy Coverage Profiling may also cause nonrecursive functions to be reported incorrectly as recursive (see “Profiling - Legacy Coverage” on page 276).

- **16.3 [R] Identifiers must be given for all function parameters** (**--misra_2004=16.3**) — Enforced.

  Associated diagnostics include 1727.

- **16.4 [R] Parameter names in function declaration and definition must match** (**--misra_2004=16.4**) — Enforced.

  Associated diagnostics include 1745.

- **16.5 [R] Declarations of functions with no parameters must have a `void' parameter (--misra_2004=16.5)** —Enforced.

  Associated diagnostics include 1728.
- **16.6 [R] Number of arguments must match the function prototype (--misra_2004=16.6)** — Always enforced.
- **16.7 [A] Pointer parameters to functions declared as pointer to `const' if possible (--misra_2004=16.7)** — Not enforced.
- **16.8 [R] Return expression must match function type (--misra_2004=16.8)** — Enforced.

  Associated diagnostics include 117 and 940.
- **16.9 [R] Function identifiers may only be used for calls or with preceding `&' operator (--misra_2004=16.9)** — Enforced.

  Associated diagnostics include 1774.
- **16.10 [R] If error information returned by a function, it must be tested (--misra_2004=16.10)** — Not enforced.

MISRA C 2004 Rules - 17 Pointers and Arrays

- **17.1 [R] No pointer arithmetic on pointers that don't address an array element (--misra_2004=17.1)** — Not enforced.
- **17.2 [R] No pointer subtraction on pointers that don't address elements of the same array (--misra_2004=17.2)** — Not enforced.
- **17.3 [R] Relational operators not used on pointers that don't point to the same array (--misra_2004=17.3)** — Not enforced.
- **17.4 [R] No pointer arithmetic other than array indexing (--misra_2004=17.4)** — Enforced.

  Associated diagnostics include 1752 and 1858.
- **17.5 [A] No more than 2 levels of pointer indirection in an object declaration (--misra_2004=17.5)** — Enforced.

  Associated diagnostics include 1741.

- **17.6 [R] Address of automatic variable not used out of scope** (**--misra_2004=17.6**) — Enforced.

  Associated diagnostics include 1056 and 1780.

MISRA C 2004 Rules - 18 Structs and Unions

- **18.1 [R] No incomplete struct or union types at end of translation unit** (**--misra_2004=18.1**) — Cannot be disabled.
- **18.2 [R] No assignments between overlapping objects** (**--misra_2004=18.2**) — Not enforced.
- **18.3 [R] No reuse of memory for unrelated purposes** (**--misra_2004=18.3**) — Not enforced.
- **18.4 [R] Unions may not be used** (**--misra_2004=18.4**) — Enforced.

  Associated diagnostics include 1827.

MISRA C 2004 Rules - 19 Preprocessing Directives

- **19.1 [A] Only preprocessing directives and comments before `#include'** (**--misra_2004=19.1**) — Enforced.

  Associated diagnostics include 1748.
- **19.2 [A] Only standard characters in file names for `#include'** (**--misra_2004=19.2**) — Enforced.

  Associated diagnostics include 1747.
- **19.3 [R] `#include' directive only followed by <filename> or "filename"** (**--misra_2004=19.3**) — Always enforced.
- **19.4 [R] Restrict macro syntax** (**--misra_2004=19.4**) — Enforced.

  Associated diagnostics include 1859, 1860, and 1886.
- **19.5 [R] No `#define' or `#undef' within a block** (**--misra_2004=19.5**) — Enforced.

  Associated diagnostics include 1732.

- **19.6 [R] `#undef' cannot be used** (**--misra_2004=19.6**) — Enforced.

  Associated diagnostics include 1715.

- **19.7 [A] Function used instead of function-like macro when possible** (**--misra_2004=19.7**) — Enforced.

  Associated diagnostics include 1862.

- **19.8 [R] Function-like macro must be called with all of its arguments** (**--misra_2004=19.8**) — Enforced.

  Associated diagnostics include 54 and 76.

- **19.9 [R] No preprocessing directives in function-like macros** (**--misra_2004=19.9**) — Enforced.

  Associated diagnostics include 10.

- **19.10 [R] In function-like macro definition, wrap each parameter reference in parentheses** (**--misra_2004=19.10**) — Enforced.

  Associated diagnostics include 1829.

- **19.11 [R] Identifiers in preprocessing directives defined before used** (**--misra_2004=19.11**) — Enforced.

  Associated diagnostics include 193.

- **19.12 [R] At most one `#' or `##' operator in a macro** (**--misra_2004=19.12**) — Enforced.

  Associated diagnostics include 1716.

- **19.13 [A] No `#' or `##' preprocessor operators** (**--misra_2004=19.13**) — Enforced.

  Associated diagnostics include 1830.

- **19.14 [R] Correct use of `defined' preprocessor operator** (**--misra_2004=19.14**) — Enforced.

  Associated diagnostics include 1831.

- **19.15 [R] Prevent contents of header files from being included more than once** (**--misra_2004=19.15**) — A diagnostic is issued if the front end does not recognize an `#ifndef...#define...#endif` include guard. Because **assert.h**

does not conform to this rule, a diagnostic is issued if you include this header file and enable this rule.

Associated diagnostics include 1882.

- **19.16 [R] Conditionally excluded preprocessor directives must be syntactically valid** (**--misra_2004=19.16**) — Enforced.

  Associated diagnostics include 11.

- **19.17 [R] All related conditional preprocessor directives must reside in the same file** (**--misra_2004=19.17**) — Enforced.

  Associated diagnostics include 36 and 37.

MISRA C 2004 Rules - 20 Standard Libraries

The following MISRA rules are enforced both by `#error` directives in the standard library headers, and by diagnostics emitted by the `elxr` linker. The linker diagnostics prevent programs from circumventing these rules by calling forbidden functions without including the standard headers (after suppressing MISRA 2004 Rule 8.1). The linker prevents any use of the forbidden functions in any library modules that are pulled into the link.

- **20.1 [R] No definition, redefinition, or undefinition of reserved words and standard library names** (**--misra_2004=20.1**) — Enforced.

  Associated diagnostics include 45, 46, 1861, and 1885.

- **20.2 [R] Standard library macro, object, and function names cannot be reused** (**--misra_2004=20.2**) — Not enforced.
- **20.3 [R] Check validity of values passed to library functions** (**--misra_2004=20.3**) — Not enforced.
- **20.4 [R] Dynamic heap memory allocation cannot be used** (**--misra_2004=20.4**) — Enforced.
- **20.5 [R] The error indicator `errno' cannot be used** (**--misra_2004=20.5**) — Enforced.
- **20.6 [R] Macro `offsetof' in <stddef.h> cannot be used** (**--misra_2004=20.6**) — Enforced.

- **20.7 [R] The `setjmp' and `longjmp' facilities cannot be used (--misra_2004=20.7)** — Enforced. Because the debugging memory library **libdbmem.a** uses `setjmp()`, building an otherwise conforming program with **-check=alloc** might trigger the associated **elxr** diagnostic.
- **20.8 [R] The facilities of <signal.h> cannot be used (--misra_2004=20.8)** — Enforced.
- **20.9 [R] No use of <stdio.h> in production code (--misra_2004=20.9)** — Enforced.
- **20.10 [R] Functions `atof', `atoi', and `atol' are not used (--misra_2004=20.10)** — Enforced.
- **20.11 [R] Functions `abort', `exit', `getenv', and `system' are not used (--misra_2004=20.11)** — Enforced. Because the default startup code uses `exit()`, an otherwise conforming program might trigger the associated **elxr** diagnostic.
- **20.12 [R] The facilities of <time.h> cannot be used (--misra_2004=20.12)** — Enforced.

MISRA C 2004 Rules - 21 Run-time Failures

- **21.1 [R] Run-time checking (--misra_2004=21.1)** — Enforced by performing run-time error checks for array bounds, assignment bounds, NULL pointer dereference, divide-by-zero, and unresolved switch statement condition. For more information about run-time error checking, see “Run-Time Error Checks” on page 159. The **--no_misra_runtime** option disables this rule.

MISRA C - Required Rules Level

Controls the diagnostic messages generated upon a violation of the MISRA *required* rules. Permitted settings for this option are:

- **Errors (--misra_req=error)** — [default]
- **Warnings (--misra_req=warn)**
- **Silent (--misra_req=silent)**

MISRA C - Advisory Rules Level

Controls the diagnostic messages generated upon a violation of the MISRA *advisory* rules. Permitted settings for this option are:

- **Errors** (**--misra_adv=error**)
- **Warnings** (**--misra_adv=warn**) — [default]
- **Silent** (**--misra_adv=silent**)

MISRA C - Run-Time Checks

Controls the performance of run-time checks for the MISRA rules that require them. Permitted settings for this option are:

- **On** (**--misra_runtime**) — [default]
- **Off** (**--no_misra_runtime**)

MISRA C 1998

This option category is contained within **C/C++ Compiler**.

This section contains options that allow you to check your code for compliance with the Motor Industry Software Reliability Association (MISRA) rules, a set of guidelines for the C programming language that is designed to enforce good practices in the development of embedded automotive systems. For more information, see “MISRA C 1998” on page 895.

Note

Support for MISRA 1998 Rules is deprecated and may be removed in future versions of **MULTI**. It is recommended that you use the MISRA 2004 Rules (see “MISRA C 2004” on page 168).

MISRA C 1998 Rules - Environment

Permitted settings for this option are:

- **1. [R] ISO 9899 C conformance w/o extensions** — Enables Strict ANSI C. This option enables **-ANSI**, which disables recognition of many non-standard keywords and extensions. Errors issued because of this check are not identified as MISRA errors, and their severities are unaffected by MISRA classification or the **--misra_req** option.
- **2. [A] Code other than C must conform to standard interface** — Not enforced.
- **3. [A] Inline assembly only in functions with no other code**
- **4. [A] Run-time checking** — Implemented through run-time checking. This option sets **-check=assignbound**, **-check=bound**, **-check=nilderef**, **-check=zerodivide**, and **-check=switch**. For more information about run-time error checking, see "Run-Time Error Checks" on page 159. The option **--no_misra_runtime** turns off this rule completely. The option **--no_misra_runtime** disables both this rule and rule 107.

MISRA C 1998 Rules - Character Set

Permitted settings for this option are:

- **5. [R] Only ISO C characters and escape sequences used** — Disables recognition of the `\e` escape in GNU mode. Issues a MISRA diagnostic on each use of a multibyte character literal (such as a Kanji character).
- **6. [R] Values of char types restricted to subset of ISO 10646-1** — All Green Hills compilers use the ASCII standard for mapping character sets to numeric values. This property cannot be turned off.
- **7. [R] No trigraphs**
- **8. [R] No multibyte chars and wide strings** — Issues a MISRA diagnostic on each use of a multibyte character literal (such as a Kanji character).

MISRA C 1998 Rules - Comments

Permitted settings for this option are:

- **9. [R] No nested comments**
- **10. [A] No 'commented out' sections of code** — Disallows the following characters inside of a comment:

- semicolon followed by a new-line character
- open or close curly braces

MISRA C 1998 Rules - Identifiers

Permitted settings for this option are:

- **11. [R] No more than 31 chars to determine an identifier** — Enforced by truncating all identifiers at 31 characters.

  Errors issued because of this check are not identified as MISRA errors, and their severities are unaffected by MISRA classification or the **--misra_req** option.

  Associated diagnostics include 1772.
- **12. [A] No identifiers with the same names in different namespaces**

MISRA C 1998 Rules - Types

Permitted settings for this option are:

- **13. [A] Basic types used only in `typedef's**
- **14. [R] `char' always used as `signed char' or `unsigned char'**
- **15. [A] Floating point implementation complies to a standard** — The Green Hills compilers use the IEEE floating-point representation. This rule cannot be turned off.
- **16. [R] No underlying use of bits in floating point expressions** — The Green Hills compilers disallow any use of bit-wise operators (such as `&`, `|`, `^`, `~`, etc.) by default. When this rule is enabled, a diagnostic is issued for union members of floating-point type.
- **17. [R] `typedef' names shall not be reused**

MISRA C 1998 Rules - Constants

Permitted settings for this option are:

- **18. [A] Numeric constants need suffixes when appropriate**
- **19. [R] No octal constants (other than zero)**

MISRA C 1998 Rules - Declarations and Definitions

Permitted settings for this option are:

- **20. [R] All objects and functions declared before used**
- **21. [R] No use of same id name in inner and outer scope**
- **22. [A] Function scope declarations whenever possible** — Not enforced.
- **23. [A] Static linkage of file scope declarations when possible** — Not enforced.
- **24. [R] No internal and external linkages of same identifiers**
- **25. [R] Only one external definition of external identifier** — Enforced in the compiler and **elxr**.

  Associated diagnostics include 1797.
- **26. [R] Compatible multiple declarations of the same object/function** — The Green Hills compilers allow only compatible declarations of multiply-declared objects and functions within the same translation unit (by default). No such checks are performed across different translation units.
- **27. [A] External objects declared in no more than one file** — Not enforced.
- **28. [A] The `register' storage class specifier should not be used**
- **29. [R] The use of tag shall agree with its declaration**

MISRA C 1998 Rules - Initialization

Permitted settings for this option are:

- **30. [R] Automatic variables initialized before used** — Enforced for simple cases.
- **31. [R] Braces used in non-zero initialization of arrays/structs**
- **32. [R] All or only first enumerator may be explicitly initialized**

MISRA C 1998 Rules - Operators

Permitted settings for this option are:

- **33. [R] No side effects in right hand operand of `&&' or `||'**
- **34. [R] Operands of `&&' and `||' shall be primary expressions**
- **35. [R] Assignment operators not used in Boolean expressions**
- **36. [A] Logical and bitwise operators should not be confused** — Not enforced.
- **37. [R] No bitwise operations on signed integer types**
- **38. [R] Right hand value of shift operand must be in range** — Both compile-time and run-time checks are performed. The run-time check is turned off when the option **--no_misra_runtime** is passed.
- **39. [R] No unary minus operand on unsigned expressions**
- **40. [A] No side effects in the `sizeof' operand**
- **41. [A] Implementation of division determined and documented** — This behavior is dependent on the hardware configuration of the target. Most Green Hills compilers, however, round the division result toward 0 (including Power Architecture, MIPS, TriCore, and x86).
- **42. [R] No comma operators except in control expr of `for' loops**

MISRA C 1998 Rules - Conversions

Permitted settings for this option are:

- **43. [R] No implicit conversions which might lose information**
- **44. [A] Redundant explicit cast should not be used**
- **45. [R] Type casting to or from pointers should not be used**

MISRA C 1998 Rules - Expressions

Permitted settings for this option are:

- **46. [R] No expression with values dependent on evaluation order** — Not enforced.

- **47. [A] No dependence placed on C precedence rules**
- **48. [A] Mixed precision arithmetic must use explicit casting**
- **49. [A] Test value against zero unless expression Boolean** — Not enforced. The Green Hills C compilers do not have built-in types which would represent the Boolean type.
- **50. [R] Floating point values not tested for (in)equality**
- **51. [A] No wraparound in constant unsigned expressions**

MISRA C 1998 Rules - Control Flow

Permitted settings for this option are:

- **52. [R] There shall be no unreachable code**
- **53. [R] All non-null statements shall have a side-effect**
- **54. [R] Null statement must occur on line by itself**
- **55. [A] No labels except in `switch' statements**
- **56. [R] The `goto' statement shall not be used**
- **57. [R] The `continue' statement shall not be used**
- **58. [R] No `break' statement except in `switch'**
- **59. [R] Dependent statements always enclosed in braces**
- **60. [A] All `if' and `else if' constructs must have an `else'**
- **61. [R] Every non-empty `case' clause terminated with `break'**
- **62. [R] All `switch' statements should contain a `default' clause**
- **63. [A] No Boolean values in `switch' expressions**
- **64. [R] `switch' statements need at least one `case'**
- **65. [R] No floating-point variables in loop counters**
- **66. [A] Only loop control expression in `for' statement header** — Not enforced.
- **67. [A] No modification of control variables in `for' loop body** — Not enforced.

MISRA C 1998 Rules - Functions

Permitted settings for this option are:

- **68. [R] Functions always declared at file scope**
- **69. [R] No variable argument functions**
- **70. [R] No direct or indirect recursion**
- **71. [R] Functions always have prototype declarations** This option sets **--prototype_errors** or **--prototype_warnings**, depending on the MISRA C Required Rules Level (see **Functions Without Prototypes** in "C/C++ Messages" on page 237).
- **72. [R] Function declaration and definition match prototypes** — Always enforced.
- **73. [R] Identifiers given for all or none of function parameters**
- **74. [R] Declaration and definition match parameter names**
- **75. [R] Every function shall have an explicit return type**
- **76. [R] Functions with no parameters should have `void' parameter**
- **77. [R] Unqualified arguments of callee and caller compatible** — Not enforced.
- **78. [R] Number of arguments matches the function prototype** — Always enforced.
- **79. [R] Values returned by `void' functions not used** — Always enforced.
- **80. [R] No void expressions passed as parameters** — Always enforced.
- **81. [A] `const' present when needed on reference parameters** — Not enforced.
- **82. [A] A function should have a single point of exit**
- **83. [R] Return expression matches function type**
- **84. [R] No expressions returned from `void' functions** — Always enforced.
- **85. [A] Functions called with empty () if no parameters**
- **86. [A] If error information returned, it should be tested** — Not enforced.

MISRA C 1998 Rules - Preprocessor

Permitted settings for this option are:

- **87. [R] Only preprocessing directives before #include**
- **88. [R] Only standard characters in file names for #include**
- **89. [R] #include directive followed by <filename> or 'filename'** — Always enforced.
- **90. [R] C macros only as constant, function-like, or specifier/qualifier**
- **91. [R] Macros not `#define'd or `#undef'd within a block**
- **92. [A] #undef should not be used**
- **93. [A] Function used instead of function-like macro when possible** — Not enforced.
- **94. [R] Function-like macro called with all of its arguments** — Always enforced.
- **95. [R] No preprocessing directives in function-like macros**
- **96. [R] Use of parentheses in function-like macros**
- **97. [A] Identifiers in pre-processing directives defined before used**
- **98. [R] Only one use of `#' or `##' operator in any one macro**
- **99. [R] Uses of #pragmas documented and explained** — This book lists all `#pragma` constructs, their usage and effects (see “Pragma Directives” on page 749).
- **100. [R] Correct use of `defined' preprocessor operator** — Always enforced.

MISRA C 1998 Rules - Pointers and Arrays

Permitted settings for this option are:

- **101. [A] Pointer arithmetic should not be used**
- **102. [A] No more than 2 levels of pointer indirection**
- **103. [R] Relational operators not used except on same object** — Not enforced.
- **104. [R] No non-constant pointers to functions**

- **105. [R] Function and a pointer to it match in prototypes**
- **106. [R] Address of automatic variable not used out of scope** — The compiler checks if an address of a local variable is assigned to a global value.
- **107. [R] The NULL pointer should not be dereferenced** — Performed via a run-time check (**-check=nilderef**). For more information about run-time error checking, see "Run-Time Error Checks" on page 159. The option **--no_misra_runtime** disables both this rule and rule 4.

MISRA C 1998 Rules - Structs and Unions

Permitted settings for this option are:

- **108. [R] All members of structures/union fully specified** — Not enforced.
- **109. [R] Overlapping variable storage should not be used** — Not enforced.
- **110. [R] Unions not used to access sub-parts of larger data types** — Not enforced.
- **111. [R] Bit fields can only have `unsigned int' or `signed int' types**
- **112. [R] Bit fields of `signed int' type must be at least 2 bits long**
- **113. [R] All members can be accessed only through their name**

MISRA C 1998 Rules - Standard Library

Permitted settings for this option are:

- **114. [R] No redefinition of reserved words and standard library names** — Always enforced. The Green Hills compilers do not allow redefinitions of C reserved words. Furthermore, the predefined macro names defined, `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, and `__STDC__` cannot be redefined.
- **115. [R] Standard library function names shall not be reused** — Not enforced.
- **116. [R] All library production code conforms to MISRA** — Not enforced.
- **117. [R] Check validity of value passed to library functions** — Not enforced.
- **118. [R] Dynamic heap memory allocation shall not be used**
- **119. [R] The error indicator `errno' shall not be used**

- **120. [R] Macro `offsetof' in <stddef.h> shall not be used**
- **121. [R] <locale.h> and `setlocale' function shall not be used**
- **122. [R] The `setjmp' and `longjmp' facilities shall not be used**
- **123. [R] The facilities of <signal.h> shall not be used**
- **124. [R] No use of <stdio.h> in production code**
- **125. [R] Functions `atof', `atoi', and `atol' not used**
- **126. [R] Functions `abort', `exit', `getenv', and `system' not used**
- **127. [R] The facilities of <time.h> shall not be used**

Data Types

This option category is contained within **C/C++ Compiler**.

These options control the treatment of particular data types.

Signedness of Char Type

Specifies the signedness of the `char` type. Permitted settings for this option are:

- **Signed** (**--signed_chars**) — [default]
- **Unsigned** (**--unsigned_chars**)

Note that the standard libraries are built with the default signedness. While most library routines work the same way regardless of the signedness of the `char` type, `localeconv()` uses the default value of `CHAR_MAX`, which will not match `CHAR_MAX` in the application if the signedness is changed.

Signedness of Bitfields

Specifies the treatment of bitfields that are not explicitly declared `signed` or `unsigned`. Permitted settings for this option are:

- **Signed** (**--signed_fields**) — [default] — Multiple-bit bitfields declared with an integer type have the signedness of their declared type. Bitfields declared

with an enumeration type may be either signed or unsigned to accommodate the sign and the magnitude of their enumerators.

- **Unsigned** (**--unsigned_fields**) Bitfields declared with an integer type are unsigned. Bitfields declared with an enumeration type are unsigned unless the enumeration type has a negative enumerator.

For more information, see "Signedness of Bitfields" on page 647.

Signedness of Pointers

Specifies the signedness of pointers and addresses. Permitted settings for this option are:

- **Signed** (**--signed_pointer**) — Pointers are signed. Because the libraries are built with **--unsigned_pointer**, operations that rely on memory spanning `0` may not work consistently between user code and libraries when using this option.
- **Unsigned** (**--unsigned_pointer**) — [default] Pointers are unsigned.

Use Smallest Type Possible for Enum

Controls the allocation of enumerations. Permitted settings for this option are:

- **On** (**--short_enum**) — Store enumerations in the smallest possible type.
- **Off** (**--no_short_enum**) — [default] Store enumerations as integers.

Long Long Support

Controls support for the `long long` data type. Permitted settings for this option are:

- **On** (**--long_long**) — [default] Support the `long long` data type.
- **Off** (**--no_long_long**) — Do not support the `long long` data type.

--no_long_long is supported only in ANSI and GNU C modes. It is not supported on any CPU with 64-bit registers, nor is it supported in INTEGRITY, u-velOSity,

or embedded Linux platforms. In addition, certain target-specific header files may require the `long long` type to exist.

Size of time_t

By default, `time_t` has the type `long`. For targets where `long` is 32 bits, this option allows you to change the size of `time_t` to `long long`. Permitted settings for this option are:

- **32 bits** (**-time32**) — [default] `time_t` has the type `long`.
- **64 bits** (**-time64**) — On targets where `long` is 32 bits, `time_t` has the type `long long` (otherwise, this option is ignored). This option is not supported with Embedded Linux.

The linker associates calls to C library functions that use `time_t` to definitions of the function with an alternate name. For more information, see "Standard Function Names Converted by the Linker" on page 811.

Alignment and Packing

This option category is contained within **C/C++ Compiler**.

Packing (Maximum Structure Alignment)

Controls the default maximum alignment of all objects of types `struct` and `class`. Permitted settings for this option are:

- **None** (**-pack=none**)
- **1-byte** (**-pack=1**)
- **2-byte** (**-pack=2**)
- **4-byte** (**-pack=4**)
- **8-byte** (**-pack=8**)

No `struct` or `class` or member of a `struct` or `class` will have an alignment greater than the specified value, unless the setting is overridden by the `#pragma pack` directive (see "General Pragma Directives" on page 750).

If you specify **None**, no packing alignment is specified to the compiler; structures, classes, and their members are aligned to their alignment requirements.

Misaligned Memory Access

Specifies whether or not the compiler generates extra code to handle misaligned data accesses. Permitted settings for this option are:

- **On** (**-misalign_pack**) — The compiler does not generate code to handle misaligned data accesses; they are handled by the hardware. When specifying this option, continue to use the `__packed` keyword when pointing to a packed structure, because the compiler must avoid generating certain assembly instructions in this case. For more information, see "Pointing to Packed Structures with the __packed Type Qualifier" on page 31.
- **Off** (**-no_misalign_pack**) — The compiler generates extra code to handle misaligned data accesses.

-misalign_pack is useful on processors that support unaligned memory accesses, such as the V850ES. Accessing unaligned data still requires more instructions on average than accessing data that is aligned.

Warning

Attempting to use this option on processors that do not support unaligned memory accesses will result in code that does not execute correctly.

When using the Green Hills V850 and RH850 simulator (**sim850**) to run code compiled with this option, the simulator must be configured to support unaligned memory accesses. See **-v850e_misalign** in "Additional Commands for Simulator for V850 and RH850 (sim850) Connections" in Chapter 6 of the *MULTI: Configuring Connections for V850 and RH850 Targets* book.

C/C++ Data Allocation

This option category is contained within **C/C++ Compiler**.

These options control the treatment of data in C and C++.

Allocation of Uninitialized Global Variables

Controls the allocation of uninitialized global variables. Permitted settings for this option are:

- **Treat as Common Data (--commons)** — [default] Treats a declaration at the outermost level of a C file such as `int foo;` as *common* data. Common data can be defined multiple times, and all declarations will be merged by the linker. For example, multiple files may have the declaration `int foo;` and the linker will merge them all into the same data entity.
- **Treat as Unique Definitions (--no_commons)** — Allocates uninitialized global variables to a section and initializes them to zero at program startup. This may improve optimizations by giving the compiler optimizer more information about the location of the variable.

  However, if your code contains multiple declarations such as `int foo;` (which would cause the linker to find multiple definition spots for `foo`), this setting may also cause `multiply defined symbol` linker errors. In particular, you may not use a definition such as `int foo;` in a header file if it will be included in more than one source file. You can avoid these linker errors by using the `extern` keyword for declarations and having a single definition point. For example:

```
int foo;         /* used in just one place, or */

int foo=0;       /* used in just one place */

extern int foo; /* used in all other places */
```

Uniquely Allocate All Strings

This option is deprecated and may be removed in future versions of MULTI.

Controls the creation of a separate space for all strings, even those which are equivalent. Permitted settings for this option are:

- **On (--unique_strings)** — Create a separate space for all strings.
- **Off (--no_unique_strings)** — [default] Do not create separate spaces for each instance of an equivalent string.

Retain Symbols for Unused Statics

Controls the retention of symbols for variable and routines that are declared `static`, but are not used. Permitted settings for this option are:

- **On** (**--keep_static_symbols**) — Retain unused static variables and routines.
- **Off** (**--no_keep_static_symbols**) — [default] Discard unused static variables and routines.

Support Variable Length Arrays

Enables or disables support for variable length arrays (VLAs) in **-c99**, **-C99**, and **-gcc** modes. Permitted settings for this option are:

- **On** (**--vla**) — [default] Enables support for VLAs.
- **Off** (**--no_vla**) — Disables support for VLAs.

Special Tokens

This option category is contained within **C/C++ Compiler**.

These options control the treatment of special tokens.

Alternative Tokens

Controls support for digraphs in C and C++. In C++, this option also controls whether operator keywords `and`, `or`, `bitand`, `bitor`, `and_eq`, `or_eq`, etc, are recognized. In C, the same tokens are provided as `#define` directives in **iso646.h**. Permitted settings for this option are:

- **On** (**--alternative_tokens**) — [default] Enables you to write C and C++ without using punctuation that might not be available on some international keyboards.
- **Off** (**--no_alternative_tokens**)

The following table covers alternatives enabled by this option:

Punctuation	Alternative
[	<:
]	:>
{	<%
}	%>
#	%:
&&	and
&=	and_eq
&	bitand
|	bitor
~	compl
!	not
!=	not_eq
||	or
|=	or_eq
^	xor
^=	xor_eq

Host and Target Character Encoding

Controls the character encoding for multibyte extended characters. This option also affects the `mbtowc()` and `wctomb()` library functions defined in **libind.a**, **libutf8.c**, and **lib8bit.a**. Permitted settings for this option are:

- **EUC on Host and Target (-kanji=euc)** — [default for Solaris hosts] Specifies Kanji character interpretation using the EUC format on the host and target.
- **Shift-JIS on Host and Target (-kanji=shiftjis)** — [default for non-Solaris hosts] Specifies Kanji character interpretation using the Shift-JIS format on the host and target.
- **EUC on Host and Shift-JIS on Target (-kanji=euc/shiftjis)** — Specifies Kanji character interpretation using the EUC format on the host and the Shift-JIS format on the target.

- **Shift-JIS on Host and UTF-8 on Target** (**-kanji=shiftjis/utf8**) — Specifies Kanji character interpretation using the Shift-JIS format on the host, UTF-8 as the target multi-byte character encoding, and UTF-32 as the target wide character encoding.
- **UTF-8 on Host and Target** (**-kanji=utf8**) — Specifies UTF-8 as the multi-byte character encoding and UTF-32 as the wide-character encoding.
- **8-bit Codepage on Host and Target** (**-kanji=none**) — Use 8-bit code pages (for example, ISO 8859-1). In this mode there are no multibyte characters; the mapping from bytes to characters is one-to-one.

C++

This option category is contained within **C/C++ Compiler**.

These options control aspects of compilation that relate to C++.

Namespaces

This option category is contained within **C/C++ Compiler→C++**.

These options control the treatment of namespaces.

Namespace Support

Controls support for namespaces. Permitted settings for this option are:

- **On** (**--namespaces**) — [default for C++ and EEC++]
- **Off** (**--no_namespaces**) — [default for EC++]

Implicit Use of 'std' Namespace

Controls the implicit use of the `std` namespace when standard header files are included. Permitted settings for this option are:

- **On** (**--using_std**)
- **Off** (**--no_using_std**) — [default]

Keyword Support

This option category is contained within **C/C++ Compiler→C++**.

These options control the treatment of various C++ keywords.

Bool Type Support

Controls support for the `bool` type. Permitted settings for this option are:

- **On** (**--bool**) — [default] Also defines the preprocessor symbol `_BOOL`, allowing code to determine when a `typedef` statement should be used to define the `bool` type.
- **Off** (**--no_bool**)

restrict Keyword Support

Controls support for the `restrict` keyword. Permitted settings for this option are:

- **On** (**--restrict**)
- **Off** (**--no_restrict**) — [default]

This keyword is a C99 feature that can be used to indicate that a pointer is not aliased.

Support __noinline Keyword

Controls support for the `__noinline` keyword. This option is automatically enabled when you set the **Optimization→Optimization Strategy** option to **Optimize for Size** (**-Osize**). Permitted settings for this option are:

- **On** (**--enable_noinline**)
- **Off** (**--disable_noinline**) — [default] The compiler ignores the `__noinline` keyword.

For more information, see “Additional C++ Inlining Information” on page 305.

Instantiate Extern Inline

Controls instantiation of `extern inline` functions and inline member functions of classes that have external linkage. Permitted settings for this option are:

- **On** (**--instantiate_extern_inline**) — Produces exactly one out-of-line copy in the program for each `inline` function that requires one. The prelinker ensures that there is one copy, unless you use the **--link_once_templates** option, in which case this is enforced with link-once sections (see "Link-Once Template Instantiation" on page 210). This option might increase compile time.
- **Off** (**--no_instantiate_extern_inline**) — [default] Produces an out-of-line copy of each `extern inline` function and each inline member function of a class with external linkage in each module that requires a copy. These out-of-line copies share static data. As a result:
  - If your code compares pointers to such functions from different modules, they will not be equal.
  - Your code size may be larger.

  For more information about inlining functions in C++, see "Manual Inlining" on page 295.

New-Style Cast Support

Controls the use of new-style casts (like `static_cast< >` and `reinterpret_cast < >`) with Embedded C++. Permitted settings for this option are:

- **On** (**--new_style_casts**)
- **Off** (**--no_new_style_casts**) — [default]

Using this option enables support for templates, which are usually not supported in Embedded C++.

Constructors/Destructors

This option category is contained within **C/C++ Compiler→C++**.

These options control the treatment of C++ constructors and destructors.

Support for Constructors/Destructors

Controls the insertion of code at the beginning of `main()` to call `_main()` which is responsible for invoking constructors on global objects that require them, and for invoking `atexit()` to cause the appropriate destructors to be invoked when the program finishes. Permitted settings for this option are:

- **On** (**--enable_ctors_dtors**) — [default]
- **Off** (**--disable_ctors_dtors**)

This option only controls the insertion of the call to `_main()` (and hence the inclusion of `_main()` from the C++ library) and does not affect the generation of code and data related to invoking constructors for global objects.

Placement of Class Constructor Call to New

Controls the generation of a call to `operator new` in the prologue of each constructor that optionally allocates memory for the object being constructed.

- **Outside** (**--new_outside_of_constructor**) — Does not call `new` from within any constructor, which decreases the size of the constructor but increases the number of instructions required to dynamically allocate a class object. This setting generates faster code when dynamic allocation of single objects is infrequent (for instance, when most class objects are allocated on the stack).
- **Inside** (**--new_inside_of_constructor**) — [default] Places a call to `new` inside each constructor, which increases the size of the constructor but decreases the number of instructions required to dynamically allocate a class object. Using this setting generates faster code when dynamic allocation of single objects is common (for instance, when most objects are allocated via `operator new`).

The compiler also places a call to `operator delete` in the epilogue of each destructor. This code is required for the correct handling of virtual destructors in the presence of a user-defined `operator delete`, and is not affected by **--new_outside_of_constructor**.

Mixing **--new_inside_of_constructor** and **--new_outside_of_constructor** within the same project may lead to undefined behavior. If an expression of the form `p = new C` is compiled with **--new_inside_of_constructor** and the definition of `C::C()` was compiled with **--new_outside_of_constructor**, neither the caller nor the callee

will allocate memory for `*p`, leading to unexpected behavior. However, if all constructors are compiled with the default **--new_inside_of_constructor**, the rest of the program may safely (but inefficiently) be compiled with **--new_outside_of_constructor**.

RTTI Support

This option category is contained within **C/C++ Compiler→C++**.

These options control the treatment of *Run-Time Type Information* (RTTI).

Run-Time Type Information Support

Controls support for Run-Time Type Information (RTTI) features `dynamic_cast` and `typeid`. Permitted settings for this option are:

- **On** (**--rtti**) — [default for C++. Not supported with EC++ or EEC++]
- **Off** (**--no_rtti**) — [default for EC++ and EEC++]

Treatment of RTTI as const

Controls the treatment of RTTI variables. Permitted settings for this option are:

- **On** (**--readonly_typeinfo**) — [default] Forces RTTI variables to be treated as `const` in the compiler. In most cases, `const` variables end up in the read-only data.
- **Off** (**--no_readonly_typeinfo**)

Virtual Tables

This option category is contained within **C/C++ Compiler→C++**.

These options control the treatment of virtual tables.

Virtual Function Definition

Controls the definition of virtual function tables in instances when the compiler heuristic provides no guidance. The virtual function table for a class is defined in a compilation if the compilation contains a definition of the first non-inline, non-pure virtual function of the class. For classes that contain no such function, the default is to define the virtual function table (but to define it as a local static entity). Permitted settings for this option are:

- **Force** (**--force_vtbl**) — Forces the definition of virtual function tables for such classes, but does not force the definitions to be local.
- **Standard** (**--standard_vtbl**) — [default] Defines the virtual function table as a local static entity.
- **Suppress** (**--suppress_vtbl**) — Suppresses the definition of virtual function tables for such classes.

Treatment of Virtual Tables as 'const'

Controls whether virtual tables are treated as `const`. Permitted settings for this option are:

- **On** (**--readonly_virtual_tables**) — [default] Forces virtual function tables to be treated as `const` in the compiler. In most cases, `const` tables end up in the read-only data.
- **Off** (**--no_readonly_virtual_tables**)

Virtual Function Table Offset Size

Controls whether or not there is a limit on the size of classes with virtual inheritance. Permitted settings for this option are:

- **Size of Pointer** (**--large_vtbl_offsets**) — There is no limit on the size of classes with virtual inheritance. Code built with this option is not compatible with INTEGRITY, with code built using Green Hills Compiler 2012.5 or earlier, or with code built using the **--no_large_vtbl_offsets** option.
- **Size of Unsigned Short** (**--no_large_vtbl_offsets**) — [default] There is a 32 kilobyte limit on the size of classes with virtual inheritance. The virtual

inheritance model in code built with this option is compatible with that in code built with Green Hills Compiler 2012.5 and earlier.

If you try to link two modules that are not compatible because of this option, the linker will issue the following diagnostic:

```
[elxr] (error) Possible C++ virtual function table format incompatibility between modules.
```

Templates

This option category is contained within **C/C++ Compiler→C++**.

These options control C++ template instantiation. For a full description of this feature, see "Template Instantiation" on page 702.

Link-Once Template Instantiation

Controls the link-once method of non-export template instantiation. Permitted settings for this option are:

- **On** (**--link_once_templates**)
- **Off** (**--no_link_once_templates**) — [default]

Guiding Declarations of Template Functions

Controls recognition of *guiding declarations* of template functions. Permitted settings for this option are:

- **On** (**--guiding_decls**)
- **Off** (**--no_guiding_decls**) — [default]

A guiding declaration is a function declaration that matches an instance of a function template but has no explicit definition (since its definition derives from the function template). For example:

```
template <class T> void f(T t) {
    // ...
}
void f(int);
```

When regarded as a guiding declaration, `f(int)` is an instance of the template; otherwise it is an independent function for which a definition must be supplied. If this option is disabled and **C/C++ Compiler→C++→Templates→Support for Old-Style Specializations** remains enabled, a specialization of a non-member template function is not recognized. It is treated as a definition of an independent function.

Implicit Source File Inclusion

Controls implicit inclusion of source files as a method of finding definitions of template entities to be instantiated. For more information, see "Implicit Inclusion" on page 708. Permitted settings for this option are:

- **On** (**--implicit_include**)
- **Off** (**--no_implicit_include**) — [default]

Support for Implicit Typenames

Controls the implicit determination, from context, of whether a template parameter-dependent name is a type or a nontype. Permitted settings for this option are:

- **On** (**--implicit_typename**)
- **Off** (**--no_implicit_typename**) — [default]

Support for Old-Style Specializations

Controls the treatment of old-style template specializations (that is, specializations that do not use the `template<>` syntax). Permitted settings for this option are:

- **On** (**--old_specializations**)
- **Off** (**--no_old_specializations**) — [default]

One Instantiation Per Object

Controls whether template instantiations are put in separate object files. Permitted settings for this option are:

- **On** (**--one_instantiation_per_object**) — Puts each instantiation of a template or `extern inline` function (see "Instantiate Extern Inline" on page 206) in a separate template instantiation object file, with a unique filename that corresponds to the name of the instantiated entity. The primary object file (the object file corresponding to the original source file) contains everything else in the compilation; that is, everything that is not an instantiation.

  The **gbuild** options **-clean** and **-cleanfirst** and the **clean** menu items in the Project Manager do not delete template instantiation object files (though they do delete primary object files).
- **Hybrid** (**--hybrid_one_instantiation_per_object**) — Puts each instantiation in the compilation in a separate object file, except where they are created by `#pragma instantiate`. Instances created by `#pragma instantiate` are added to the primary object file generated from the source file in which they appear.
- **Off** (**--no_one_instantiation_per_object**) — [default]

If you are creating a library, we recommend that you set this option to **On** or **Hybrid**. By putting each instantiation in a separate object file within the library, only those instantiations that are needed will be pulled in, thus reducing code size. One of these options must be enabled if two different libraries include some of the same instantiations, in order to avoid multiply defined symbols.

Template Instantiation Output Directory

When **C/C++ Compiler→C++→Templates→One Instantiation Per Object** is enabled (**--one_instantiation_per_object**), this option can be used to specify or create a *directory* into which the generated object files should be put. Multiple projects may not share an instantiation directory. The equivalent driver option is:

- **--instantiation_dir=*directory***

The default instantiation directory is **./template_dir**.

If the directory setting used for **-instantiation_dir** is a relative path, that path is resolved relative to the Object File Output Directory (**-object_dir**). If **-object_dir** is not set, then the Instantiation Directory path is relative to the current working directory.

Prelink with Instantiations

Controls a mode of the Builder or driver that runs the C++ **prelink** utility to instantiate templates without running the linker or archiver. While this approach to template instantiation works with **clearmake**, we recommend that you use `--link_once_templates` instead, if it is available. Permitted settings for this option are:

- **On** (**--prelink_objects**) — Creates object files, but does not perform linking. The object files contain all template instantiations required, and so can be linked later without concern for template requirements.
- **Off** (**--no_prelink_objects**) — [default]

When using the driver, do not use this option in combination with any option that prevents the linker from being run, such as **-E**, **-P**, **-S**, or **-c**.

Dependent Name Processing

Controls the separate lookup of names in templates at the time the template is parsed and at the time it is instantiated. Permitted settings for this option are:

- **On** (**--dep_name**) — [default for **--g++** and **--STD** modes] Also implies **C/C++ Compiler→C++→Templates→Parse Templates in Generic Form** (**--parse_templates**).
- **Off** (**--no_dep_name**) — [default for all other modes]

Parse Templates in Generic Form

Controls the parsing of nonclass templates in their generic form (i.e., even if they are not really instantiated). Permitted settings for this option are:

- **On** (**--parse_templates**)

- **Off** (**--no_parse_templates**) — [default]

This option is implied by **Dependent Name Processing** (see "Dependent Name Processing" on page 213).

Recognition of Exported Templates

Controls recognition of exported templates. Permitted settings for this option are:

- **On** (**--export**)
- **Off** (**--no_export**) — [default]

This option enables **C/C++ Compiler→C++→Templates→Dependent Name Processing** (**--dep_name**) (see above). It cannot be used with **C/C++ Compiler→C++→Templates→Implicit Source File Inclusion** (**--implicit_include**).

No levels of interprocedural optimizations are supported with export templates. If any level is selected it will be disabled.

Prelink File to Create Template Instances

Specifies an external object file or executable to notify the prelinker of existing template instances. The equivalent driver option is:

- **-prelink_against** ***file***

Deferral of Function Template Parsing

Controls whether function templates are parsed immediately upon their definition, or delayed until there is an instantiation of the template. Permitted settings for this option are:

- **On** (**--defer_parse_function_templates**) — Parsing of function templates is delayed until there is an instantiation of the template. This behavior may allow syntax errors in templates that are not instantiated to go unnoticed by the compiler. This option is enabled by default in the **--g++** dialect.

- **Off** (**--no_defer_parse_function_templates**) —Parsing of function templates happens immediately upon their definition. This option is enabled by default in all other dialects.

Assembler Options

This is a top-level option category.

These options control the general operation of the assembler.

For additional, more specialized options, see “Assembler Options” on page 285.

Source Listing Generation

Controls the generation of a source listing. Permitted settings for this option are:

- **Generate Default Listing** (**-list**) — Creates a listing by using the name of the object file with the **.lst** extension.
- **Generate User-Specified Listing** (**-list=*filename***) — Creates a listing with the specified *filename*.
- **Suppress Listing** (**-no_list**) — [default]

Source Listing Generation Output Directory

Outputs assembly listing files in the specified directory. By default, listing files are output in the same directory as the object file. The equivalent driver option is:

- **-list_dir=*directory***

Preprocess Assembly Files

Controls whether assembly files with standard extensions such as **.s** and **.asm** are preprocessed. Permitted settings for this option are:

- **On** (**-preprocess_assembly_files**)
- **Off** (**-no_preprocess_assembly_files**) — [default] Only assembly files with special extensions are preprocessed.

Preprocess Special Assembly Files

Controls whether assembly files with a **.800** extension are preprocessed. Permitted settings for this option are:

- **On** (**-preprocess_special_assembly_files**) — [default]
- **Off** (**-no_preprocess_special_assembly_files**) — Files with a **.800** extension are not preprocessed.

Interleaved Source and Assembly

Controls the interleaving of your original source code with the generated assembly code. Normally used with the **Assembler→Source Listing Generation** option. Permitted settings for this option are:

- **On** (**-passsource**)
- **Off** (**-nopasssource**) — [default]

Not every line of the original source code will appear in the output.

Additional Assembler Options

Passes the specified assembler options to the **ease850** assembler command line. The equivalent driver option is:

- **-asm=*options*** — To pass multiple options, separate the options by spaces and enclose the whole string in quotes, or specify **-asm=*options*** multiple times.

  For example:

  ```
  -asm="-nogen -ref -w"
  ```

  or:

  ```
  -asm=-nogen -asm=-ref -asm=-w
  ```

For a full list of assembler options which may be passed in this manner, see Chapter 6, “The ease850 Assembler” on page 361.

Assembler Command File

Passes the options specified in *file* directly to the assembler. The equivalent driver option is:

- **-asmcmd=*file***

The file must contain only one assembler option per line. For example:

```
-nogen
-ref
-w
```

For a full list of assembler options which may be passed in this manner, see Chapter 6, “The ease850 Assembler” on page 361.

Support for C Type Information in Assembly

Controls assembler support for the `.inspect` and `.struct` directives, and the `offsetof()` and `sizeof()` operators. These allow you to parse C header files to get type information, and use it to define objects in assembly. For more information, see Chapter 7, “Assembler Directives” on page 379, and “C Type Information Operators” on page 373. Permitted settings for this option are:

- **On** (**-asm3g**)
- **Off** (**-noasm3g**) — [default]

Linker Options

This is a top-level option category.

These options control the general operation of the linker. If you use the driver to compile and link your program in multiple steps, pass the same set of options for each step.

For additional, more specialized options, see the advanced linker options section ("Linker Options" on page 286).

Output File Type

Controls the form of linker output. Permitted settings for this option are:

- **Executable / Located Program** (**-locatedprogram**) — [default] Generates an executable program.
- **Relocatable Object File** (**-relobj**) — Generates a relocatable object file, suitable for passing as input to another run of the linker. Implies **-nostdlib**, meaning the linker does not link in any startup files or libraries, and **-undefined**, meaning the linker does not allocate common variables or give errors for undefined symbols, as **Relocatable Program** does.

  If you are generating debug information for a relocatable object file with **-G** and the machine that will be performing the subsequent link does not have access to the original **.o** or **.dbo** files, pass the **-search_for_dba** option. This option creates a **.dba** file that you can distribute with the relocatable object file to provide debug information to MULTI. When performing subsequent links with the relocatable object file, continue to pass the **-search_for_dba** option. This option passes **-r** to the linker.
- **Relocatable Program** (**-relprog**) — Retains relocation information in the output file. The resulting file is suitable for execution. Some of the final link steps, including but not limited to C++ constructors and special symbols, are not guaranteed to have relocations, and thus might not be valid if the output file is loaded at a different address. This option passes **-a** to the linker.

Generate Additional Output

Creates the specified output type in addition to the project executable. The syntax ***-format=name*** allows you to specify the name of the file. Permitted settings for this option are:

- **Memory Image File** (**-memory**, **-memory=*name***) — Generates output file with `.mem` extension containing the output of the image as translated by the **gmemfile** utility program. For more information about **gmemfile**, see "The gmemfile Utility Program" on page 534.
- **S-Record File** (**-srec**, **-srec=*name***) — Generates output file with `.run` extension containing the output of the image as translated by the **gsrec** utility program. For more information about **gsrec**, see "The gsrec Utility Program" on page 554.
- **HEX386 File** (**-hex**, **-hex=*name***) — Generates output file in HEX386 format with `.run` extension containing the output of the image as translated by the **gsrec** utility program with **-hex386** passed to it. For more information about **gsrec**, see "The gsrec Utility Program" on page 554.
- **None** (**--no_additional_output**) — [default] You can use this option in a MULTI Project (**.gpj**) file, but there is no equivalent driver option.

Executable Stripping

Controls *stripping* the executable at the conclusion of linking. Permitted settings for this option are:

- **On** (**-strip**) — Stripping involves removing line number, symbol table, and debugging information to reduce the file size of the executable.
- **Off** (**-nostrip**) — [default]

For information about stripping executables after link-time, see "The gstrip Utility Program" on page 579.

Start Address Symbol

Specifies the *symbol* whose address is used as the program entry point or start address. Permitted settings for this option are:

- **Start Address** (**-e *symbol***) — The default symbol is `_start`.
- **No Entry Symbol** (**-noentry**)

Append Comment Section with Link-Time Information

Controls the printing of the tools version number and a timestamp to the output of the linker. Similar to using `#pragma ident` (see "General Pragma Directives" on page 750). Permitted settings for this option are:

- **On** (**-Qy**)
- **Off** (**-Qn**) — [default]

Preprocess Linker Directives Files

Controls the level of preprocessing performed on linker directives files. Permitted settings for this option are:

- **Full** (**--preprocess_linker_directive_full**) — The C preprocessor preprocesses linker directives files.
- **Partial** (**--preprocess_linker_directive**) — The C preprocessor resolves preprocessing directives such as `#include`, `#if`, and `#ifdef`. The preprocessor expands macros in preprocessing directives, but not in the rest of the file.
- **Off** (**--no_preprocess_linker_directive**) — [default]

Linker Warnings

Controls the display of linker warnings. Permitted settings for this option are:

- **Display** (**-linker_warnings**) — [default]
- **Suppress** (**-no_linker_warnings**)

Raw Import Files

Imports data from arbitrary files (such as binary files) into a section named `.raw`. The equivalent driver option is:

- **-rawimport**

If your section map does not specify a section named `.raw`, the linker issues a warning. To silence the warning, add a section named `.raw` to your section map.

Alternatively, you can select a section for each imported file using section inclusion commands in your linker directives (**.ld**) file. For example, to import the file **rawdata.bin** and place the data in the section `.myrawdata`, use **-rawimport rawdata.bin** and add the following line to your section map:

```
.myrawdata :{rawdata.bin(.raw)}
```

By default, imported sections become read-only data sections. To make a section writable or executable, use the `SHFLAGS` section attribute. For more information, see "Using Section Attributes" on page 456 and "Defining a Section Map with the SECTIONS Directive" on page 447.

Linker Directive Files with Non-standard Extensions

Passes *file* with a non-standard extension to the linker as a linker directives file, instead of any default file. You can use this option multiple times.

The equivalent driver option is:

- **-T *directives_file***

The following list describes how the compiler interprets **-T** depending on the name of *file*:

- *file* has a known linker directives extension (**.ld**) — While the compiler passes the file to the linker, **-T** is not necessary. In this case, pass just the filename to the driver.
- *file* has an extension commonly used for source files or object files (**.c**, **.obj**, **.a**, etc.) — The compiler issues the following warning and ignores **-T**:

  ```
  Warning: Option "-T" ignored due to invalid suffix of argument file.
  Use .ld as suffix
  ```

- *file* begins with a hyphen (-) — The compiler issues the following warning and ignores **-T**:

```
Warning: Option "-T" ignored due to invalid argument -file.
expected directory or filename, without leading -
```

Additional Linker Options (beginning of link line)

Passes the specified linker *options* to the **elxr** linker command line at the very beginning of the line, before any **.ld** files, and before any other options. The equivalent driver option is:

- **-lnk0=*options*** — To pass multiple options, separate the options by spaces and enclose the entire string in quotes, or specify **-lnk0=*options*** multiple times.

  For example:

  ```
  -lnk0="-multiple -undefined"
  ```

  or:

  ```
  -lnk0=-multiple -lnk0=-undefined
  ```

For more information about linker-specific options, see “Linker-Specific Options” on page 437.

For additional information about passing linker options directly, see “Additional Linker Options (before start file)” on page 223.

Additional Linker Options (before start file)

Passes the specified linker *options* to the linker command line after any **.ld** files and after most default options, but before the start files. The equivalent driver option is:

- **-lnk=*options*** — To pass multiple options, separate the options by spaces and enclose the whole string in quotes, or specify **-lnk=*options*** multiple times.

  For example:

  ```
  -lnk="-multiple -undefined"
  ```

  or:

```
-lnk=-multiple -lnk=-undefined
```

There are three options for passing options through to the linker command line. Each of the three options places its argument on the linker command line in a different location as described in the following list:

- **-lnk0=** should be used for options that must appear at the very beginning of the linker command line, before the linker directives files, and before any options that are passed by the driver.
- **-lnk=** causes the options to appear after many of the drivers options, but before any object files. Options that enable or disable a feature have a higher precedence if passed with **-lnk=** than with **-lnk0=**.
- **-Wl** should be used in the rare case that an option must appear in between object files on the linker command line, or if an option must appear at the end of the linker command line. The position of the **-Wl** option on the driver command line determines its position on the linker command line, relative to all source and object files.

For more information about linker-specific options, see “Linker-Specific Options” on page 437.

Additional Linker Options (among object files)

Passes the specified linker options to the linker in approximately the position that it appears on the driver command line. The equivalent driver option is:

- **-Wl,*option*[,*option*]…**

For more information about linker-specific options, see “Linker-Specific Options” on page 437.

For additional information about passing linker options directly, see “Additional Linker Options (before start file)” on page 223.

Linker Command File

Passes the options specified in *file* directly to the linker. The equivalent driver option is:

- **-lnkcmd=*file***

The file must contain only one linker option per line. For example:

```
-nogen
-ref
-w
```

For more information about linker-specific options, see "Linker-Specific Options" on page 437.

Linker Optimizations

This option category is contained within **Linker**.

These options control individual optimizations that can be implemented at link-time. You can enable or disable all of them with the **Optimization→Linker Optimizations** option (see "Optimization Options" on page 145).

Deletion of Unused Functions

Controls the removal from the executable of functions that are unused and unreferenced. Permitted settings for this option are:

- **On** (**-delete**)
- **Off** (**-no_delete**) — [default]

For a list of functions that have been deleted, pass the **-lnk=-v** option. Certain options that create references to functions may inhibit **-delete**. For more information, see "Deleting Unused Functions" on page 471.

Code Factoring

Controls the code factoring optimization, which reduces code size by merging redundant sequences of object code at link-time. We recommend that you use this optimization only if your primary goal is reducing program size; for most programs, code factoring increases branching and decreases execution speed. Permitted settings for this option are:

- **On** (**-codefactor**)
- **Off** (**-no_codefactor**) — [default]

For more information see "Code Factoring" on page 473.

Link-Time Ignore Debug References

Controls whether or not the linker ignores relocations from DWARF debug sections when you are using the **-delete** option. Permitted settings for this option are:

- **On** (**-ignore_debug_references**) — Ignores relocations from DWARF debug sections when using **-delete**. DWARF debug information will contain references to deleted functions that may break some third-party debuggers.
- **Off** (**-no_ignore_debug_references**) — Does not ignore relocations from DWARF debug sections when using **-delete**. DWARF debugging information generated by **-dual_debug** greatly reduces the effectiveness of **-delete**.

Start and End Files

This option category is contained within **Linker**.

These options control the use of start and end files.

Start Files

Controls the start files to be linked into the executable. The default is to use the Green Hills start files. Permitted settings for this option are:

- **Use Specified Start Files** (**-startfiles=*file*[,*file*…]**)

- **Do Not Use Start Files (-nostartfiles)**

End Files

Specifies end files to be linked into the executable. End files are the last object files passed on the command line to the linker. The equivalent driver option is:

- **-endfile=*file*[,*file*…]**

Start File Directory

Specifies the directory that contains start files, end files, and the default linker directives file. The effect of this option on each type of file is overridden by **-startfiles**, **-endfile**, and **-directive_dir**. If you are building an INTEGRITY project that is not a kernel, this option has no effect on the location of the default linker directives file. The equivalent driver option is:

- **-startfile_dir=*directory***

Symbols

This option category is contained within **Linker**.

These options control the linker's treatment of symbols.

Multiply-Defined Symbols

Controls the treatment of multiply-defined symbols. Permitted settings for this option are:

- **Errors (-no_multiple)** — [default]
- **Silent (-multiple)**

We recommend that you do not use **-multiple**, because some optimizations may assume that all objects and functions have one definition (except `weak` functions and `COMMON` variables).

Undefined Symbols

Controls the treatment of undefined symbol references. Permitted settings for this option are:

- **Errors** (**-no_undefined**) — [default]
- **Silent** (**-undefined**) — The linker gives each undefined symbol an address of zero, or a zero offset if you are using PID. If the symbol would normally be assigned to a special data area, it is given an address in the appropriate special data area section.

We recommend that you do not use **-undefined**, because some optimizations may assume that all objects and functions have one definition (except `weak` functions and `COMMON` variables).

Define Linker Constant

Sets a default value that can be used in section and memory maps. This option overrides any default value specified in a linker directives file. The equivalent driver option is:

- **-C*name=value***

Force Undefined Symbol

Forces an undefined symbol reference for *symbol*, as if there had been some use of the symbol in one of the modules to be linked. This may cause modules to be linked in from libraries that would not otherwise be included. The equivalent driver option is:

- **-u *symbol***

Force All Symbols From Library

Pulls in every exported symbol from the specified library, instead of pulling in only those symbols that are required. The equivalent driver option is:

- **-extractall=*library***

Where the *library* parameter must exactly match the way that the library is specified on the command line. For example, use `-ltest` for the *library* parameter instead of `libtest.a`.

To pull in all the symbols from two or more libraries, pass multiple **-extractall** options, or pass one **-extractall** option with a comma-separated list of library names. For example, the following are equivalent:

```
-extractall=-lfoo,-lbar
-extractall=-lfoo -extractall=-lbar
```

Export All Symbols From Library

Exports all symbols from the specified library. The equivalent driver option is:

- **-exportall=*library***

Where the *library* parameter must exactly match the way that the library is specified on the command line. For example, use `-ltest` for the *library* parameter instead of `libtest.a`.

To export all the symbols from two or more libraries, pass multiple **-exportall** options, or pass one **-exportall** option with a comma-separated list of library names. For example, the following are equivalent:

```
-exportall=-lfoo,-lbar
-exportall=-lfoo -exportall=-lbar
```

Import Symbols

Makes available symbol names and addresses from a separately linked, fully located file during linking. The linker does not include the contents of the file in the output; it only imports those symbol addresses necessary for the current link. This option is useful when one linker image must refer to symbols that are located in another separately linked image. The equivalent driver option is:

- **-A *file***

Where *file* must be a file linked with the **-locatedprogram** option.

When linking one INTEGRITY POSIX DLL, you may use **-A** to link against another. For example:

```
--posix_dll -o lib1.dll -A lib2.dll
```

This tells the linker that it is expected that `lib1.dll` will only be loaded into a program which has already loaded `lib2.dll`. The linker will allow unresolved symbols in `lib1.dll` provided that they are exported by `lib2.dll`. This is in preference to doing:

```
--posix_dll -o lib1.dll -undefined
```

Linker Output Analysis

This option category is contained within **Linker**.

These options control various forms of linker output.

Map File Generation

Controls the generation of a map file. Permitted settings for this option are:

- **Generate Default Map File** (**-map**) — [default] Creates a map file with the name of the object file plus a **.map** extension.
- **Generate User-Specified Map File** (**-map=*filename***) — Creates a map file with the specified *filename*.
- **Suppress Map** (**-nomap**)

Map File Retention

Controls the retention of the map file in the event of a link error. Permitted settings for this option are:

- **On** (**-keepmap**)
- **Off** (**-nokeepmap**) — [default]

Map File Page Length

Specifies the length, *n*, of a map file page. The equivalent driver option is:

- **-maplines=*n***

Map File Sorting

Controls the method for ordering the map file. Permitted settings for this option are:

- **Alphabetic** (**-Ma**) — [default]
- **Numeric** (**-Mn**)

Map File Cross-Referencing

Controls the generation of cross-reference information in the map file. Permitted settings for this option are:

- **On** (**-Mx**)

Display Unused Functions in Map File

Marks unreferenced symbols in the map file.

Permitted settings for this option are:

- **On** (**-Mu**) — Adds an extra column to the map file's table of symbols that contains `D` for symbols which are unreferenced and therefore able to be deleted. This includes symbols that have already been deleted by the **-delete** option (if passed). This column contains `U` for any other symbols that remain unresolved, including undefined weak symbols. It is possible that some unreferenced symbols are not marked with `D`, either because they overlap with reachable symbols or because they are referenced from profiling or debugging information. For more information, see “Deleting Unused Functions” on page 471.

Display Local Symbols in Map File

Permitted settings for this option are:

- **On** (**-Ml**) — Adds list of locals to the linker-generated map file. This list does not contain zero-sized symbols or compiler-generated symbols.

Output File Size Analysis

Controls use of the **gsize** utility to determine the size of the output executable. **gsize** creates a file called ***executable_name*.siz**, which contains a list of the sizes of the ROM sections of the output executable. Permitted settings for this option are:

- **On** (**-gsize**)
- **Off** (**-no_gsize**) — [default]

For more information, see "The gsize Utility Program" on page 552.

Call Graph Generation

Controls the generation of a call graph. Permitted settings for this option are:

- **Generate Default Call Graph** (**-callgraph**) — Creates a call graph with the name of the object file plus a **.graph** extension.
- **Generate User-Specified Call Graph** (**-callgraph=*filename***) — Creates a call graph with the specified *filename*.
- **Suppress Call Graph** (**-no_callgraph**) — [default] You can use this option in a MULTI Project (**.gpj**) file, but there is no equivalent driver option.

Link-Time Checking

This option category is contained within **Linker**.

These options control various forms of linker checking.

Section Overlap Checking

Controls the treatment of overlapping sections. Permitted settings for this option are:

- **Errors on Any Overlap** (**-strict_overlap_check**) — Issues errors for all overlapping sections, even when a section is zero bytes in size.
- **Errors on Non-Zero Overlap** (**-nooverlap**) — [default] Issues errors for overlapping sections, except when one or both of the sections is zero bytes in size.
- **Warnings** (**-overlap_warn**) — Similar to **-nooverlap**, but issues warnings instead of errors.
- **Silent** (**-overlap**) — No warnings or errors are issued for overlapping sections.

Checksum

Controls the creation of section checksums. Permitted settings for this option are:

- **On** (**-checksum**) — Appends a 4-byte checksum to the end of every initialized program section using a standard 32-bit CRC algorithm.
- **Off** (**-nochecksum**) — [default]

The algorithm is a standard 32-bit CRC using the polynomial `0x10211021` (by default), or `0x04C11DB7` (for INTEGRITY). For information about how to use these checksums, see “Verifying Program Integrity” on page 115. Sample source code for verifying checksums is included in **src/libstartup/cksum.c** of your installation.

Compiler Diagnostics Options

This is a top-level option category.

These options control the type, form, and quantity of diagnostic messages you may receive during building.

Warnings

Controls the display of *warnings* for most Green Hills tools. Permitted settings for this option are:

- **Display** (**--warnings**) — [default]
- **Suppress** (**-w**)

Remarks

Controls the display of *remarks* for most Green Hills tools. Permitted settings for this option are:

- **Display** (**--remarks**)
- **Suppress** (**--no_remarks**) — [default]

Maximum Number of Errors to Display

Limits to *n* the number of error messages the compiler or linker prints before quitting. The default is 100 and the minimum is 2.

The equivalent driver option is:

- **-errmax=*n***

Some errors are catastrophic, and the compiler stops the compilation process after encountering such an error, regardless of whether the error limit has been reached.

Redirect Error Output to File

Specifies a *file* to which all error output is redirected. This option is not supported when parallel build mode is enabled (**-parallel**). The equivalent driver option is:

- **-stderr=*file***

Quit Building if Warnings are Generated

Controls whether the build will terminate with an error if any warnings are generated, providing the ability to treat all driver warnings as errors. This option does not affect warnings issued by the Builder. To treat Builder warnings as errors, pass the **-strict** option to the Builder (for more information, see "The gbuild Utility Program" on page 509. Permitted settings for this option are:

- **On** (**--quit_after_warnings**)
- **Off** (**--no_quit_after_warnings**) — [default]

Display Version Information

Instructs the compiler and other tools to print its copyright banner and version number. Permitted settings for this option are:

- **On** (**-V**)
- **Off** (**--no_version**) — [default]

Green Hills Standard Mode

Enables Green Hills Standard Mode, which enables warnings and errors that enforce a stricter coding standard than regular C and C++. It was designed to aid you in avoiding common pitfalls when designing complex programs. The versions of the standard are named after the year they were finalized. The year does not necessarily match the version number of the Green Hills Compiler. To suppress a diagnostic issued by this mode, use **--diag_suppress=** (see "Set Message to Silent" on page 241). Permitted settings for this option are:

- **None** (**--ghstd=none**) — Explicitly disables Green Hills Standard Mode. Use this option on children of a project for which Green Hills Standard Mode is enabled to disable it for those children.
- **2010** (**--ghstd=2010**) — Enables the 2010 version of Green Hills Standard Mode. This option is equivalent to `--coding_standard=ghstd2010 -Wformat`. For a list of diagnostics promoted or enabled by this option, see " GHS Standard Mode" on page 892.
- **last** (**--ghstd=last**) — Use the most recent version of Green Hills Standard mode.

Coding Standard Profile

Specifies a coding standard profile. For information about coding standard profile syntax and creating your own profiles, see Chapter 14, "Coding Standards" on page 723. The equivalent driver option is:

- **--coding_standard=*file***

If *file* is the name of a coding standard present in the MULTI install directory (such as `ghstd2010`), the compiler uses that standard. Otherwise, the argument is processed as the name of a coding standard profile relative to the current directory.

Varying Message Format

This option category is contained within **Compiler Diagnostics**.

These options control various aspects of the display of diagnostic messages.

Brief Error Message Mode

Controls a mode in which a shorter form of the diagnostic output is used. When enabled, the original source line is not displayed and the error message text is not wrapped when it is too long to fit on a single line. Permitted settings for this option are:

- **On** (**--brief_diagnostics**)
- **Off** (**--no_brief_diagnostics**) — [default]

Line Wrap Messages

Controls whether diagnostic messages are wrapped when they are too long to fit on a single line. Permitted settings for this option are:

- **On** (**--wrap_diagnostics**) — [default]
- **Off** (**--no_wrap_diagnostics**)

Display Error Message Paths

Controls whether full pathnames are given for files mentioned in diagnostic messages. Permitted settings for this option are:

- **On** (**-error_basename**) — Filenames in diagnostic messages include only the name of the file. No path information is provided.
- **Off** (**-no_error_basename**) — [default] — Filenames in diagnostic messages include the path information that was passed to the driver.

C/C++ Messages

This option category is contained within **Compiler Diagnostics**.

These options control various diagnostic messages specific to the C and C++ languages.

Functions Without Prototypes

Controls the treatment of functions that are referenced or called when no prototype has been provided. Permitted settings for this option are:

- **Errors** (**--prototype_errors**)
- **Warnings** (**--prototype_warnings**) — [default]
- **Silent** (**--prototype_silent**)

Asm Statements

Controls the treatment of `asm` statements. Permitted settings for this option are:

- **Errors** (**--asm_errors**)
- **Warnings** (**--asm_warnings**) — [default for strict ANSI C and strict ANSI C++]
- **Silent** (**--asm_silent**) — [default for all other C and C++ modes]

In strict ANSI C, the spelling `asm` is not part of the C language. It is unknown to the compiler and may be used as the name of a variable or function. To write an `asm` macro in strict ANSI C or strict ISO C99, you must use `__asm`. In non-strict ANSI C and ISO C99, and in C++, the two spellings (`asm` and `__asm`) are identical.

Any correct use of `__asm` may still give a diagnostic indicating that`__asm` is non-standard.

Unknown Pragma Directives

Controls the treatment of `#pragma` directives that are not recognized by the compiler. Permitted settings for this option are:

- **Errors** (**--unknown_pragma_errors**)
- **Warnings** (**--unknown_pragma_warnings**) — [default]
- **Silent** (**--unknown_pragma_silent**)

Incorrect Pragma Directives

Controls the treatment of valid `#pragma` directives that use the wrong syntax. Permitted settings for this option are:

- **Errors** (**--incorrect_pragma_errors**)
- **Warnings** (**--incorrect_pragma_warnings**) — [default]
- **Silent** (**--incorrect_pragma_silent**)

printf and scanf Argument Type Checking

Controls argument type checking against the format string in calls to `printf()` and `scanf()`. The checking is only performed if the format string is a constant. Permitted settings for this option are:

- **On (-Wformat)** — The compiler performs argument type checking against `printf()` and `scanf()` format strings. This option also defines the `__CHECK_PRINTF__` macro.
- **Off (-Wno-format)** — [default] The compiler does not perform argument type checking against `printf()` or `scanf()` format strings.

This option has no effect if Green Hills header files are not used.

Implicit Int Return Type

Controls a warning that is issued if the return type of a function is not declared before it is called (thus causing the return type to be implicitly declared `int`). Permitted settings for this option are:

- **On (-Wimplicit-int)**
- **Off (-Wno-implicit-int)** — [default]

For broader prototype checking, see **Compiler Diagnostics→C/C++ Messages→Functions Without Prototypes**.

Shadow Declarations

Controls a warning that is issued if the declaration of a local variable *shadows* the declaration of a variable of the same name declared at the global scope, or at an outer scope. Permitted settings for this option are:

- **On (-Wshadow)**
- **Off (-Wno-shadow)** — [default]

Trigraphs

Controls a warning that is issued for any use of trigraphs. Permitted settings for this option are:

- **On** (**-Wtrigraphs**)
- **Off** (**-Wno-trigraphs**) — [default]

This option does not control diagnostic 1967, which warns about trigraphs that may unintentionally affect semantics. To disable this diagnostic as well, use **--diag_suppress 1967**.

Undefined Preprocessor Symbols

Controls a warning that is issued for undefined symbols in preprocessor expressions. Such undefined symbols are always given the value of 0. Permitted settings for this option are:

- **On** (**-Wundef**)
- **Off** (**-Wno-undef**) — [default]

Varying Message Severity

This option category is contained within **Compiler Diagnostics→C/C++ Messages**.

These options allow you to increase or decrease the severity of any of the *discretionary* C and C++ compiler diagnostic messages. See "Display Error Message Numbers" for more information.

Values for the error number, *n*, can be obtained by enabling the **Compiler Diagnostics→C/C++ Messages→Varying Message Severity→Display Error Message Numbers** (**--display_error_number**) option (see below). The complete set of error numbers is also listed in the *MULTI: C and C++ Compiler Error Messages* book (not available in print).

Set Message to Error

Sets the specified diagnostics message to the level of *error*. The equivalent driver option is:

- **--diag_error *n[,n2]...***

Set Message to Warning

Sets the specified diagnostics message to the level of *warning*. The equivalent driver option is:

- **--diag_warning *n[,n2]...***

Set Message to Remark

Sets the specified diagnostics message to the level of *remark*. The equivalent driver option is:

- **--diag_remark *n[,n2]...***

Set Message to Silent

Sets the specified diagnostics message to the level of *silent*. The equivalent driver option is:

- **--diag_suppress *n[,n2]...***

Display Error Message Numbers

Controls the display of error message numbers in any diagnostic messages that are generated. Permitted settings for this option are:

- **On** (**--display_error_number**) — [default]
- **Off** (**--no_display_error_number**)

This option may be used to determine the error number to be used when overriding the severity of a diagnostic message.

The *diagnostic* number will be followed by **-D** if the diagnostic is discretionary, meaning that its severity may be modified. For example:

```
"hello.c", line 24: warning #549-D:
variable "i" is used before its value is set
```

The displayed diagnostic is number 549 and is discretionary, so it may be suppressed or have its severity changed by the preceding options.

DoubleCheck (C/C++) Options

This is a top-level option category.

DoubleCheck Level

Each level performs a different degree of processing. The higher the level, the more bugs can be identified, and a longer processing time is required. Permitted settings for this option are:

- **None** (**-double_check.level=none**) — [default] DoubleCheck is disabled.
- **Low** (**-double_check.level=low**) — Looks for all types of bugs, but does not follow execution paths involving function calls. DoubleCheck runs quickly and should not greatly affect the time required to compile files.
- **Medium** (**-double_check.level=medium**) — Looks for all types of bugs, and follows execution paths through calls between functions in all source files. This extra processing requires more time than the low level.
- **High** (**-double_check.level=high**) — Extends the medium level by investigating even more interprocedural execution paths. This requires more processing time than the medium level.

The medium and high levels require two-pass compilation. The first pass summarizes the contents of source files. The second pass uses the summaries of all source files when compiling each file to identify potential intermodule paths.

For more information about DoubleCheck and these options, see Chapter 5, “The DoubleCheck Source Analysis Tool” on page 333.

DoubleCheck Report File

Sets the name of the report file DoubleCheck generates. The equivalent driver option is:

- **-double_check.report=**

For more information about **-double_check.report=**, see “Specifying a DoubleCheck Report File” on page 336.

DoubleCheck Config File

Specifies the configuration file containing function properties. The equivalent driver option is:

- **-double_check.config=**

For more information about **-double_check.config=**, see "Specifying Function Properties in a DoubleCheck Configuration File" on page 337.

DoubleCheck Errors to Ignore

Disables a particular DoubleCheck error message or warning message for a specific file or part of a build hierarchy. The equivalent driver option is:

- **-double_check.ignore=**

For more information about **-double_check.ignore=**, see "Ignoring Source Analysis Errors and Warnings" on page 343.

DoubleCheck Output that Stops Builds

Specifies whether DoubleCheck error or warning messages stop the build process. Permitted settings for this option are:

- **Off** (**-double_check.stop_build=off**) — [default] DoubleCheck error and warning messages do not stop the build.
- **Errors Only** (**-double_check.stop_build=errors**) — Only DoubleCheck error messages stop the build.
- **Warnings and Errors** (**-double_check.stop_build=warnings**) — Both DoubleCheck error and warning messages stop the build.

For more information about **-double_check.stop_build=**, see "Promoting Source Analysis Errors and Warnings" on page 343.

GCores Options

This is a top-level option category.

These options are only applicable to MultiCoreArchive project files.

Please see **gcores** documentation "The gcores Utility Program" on page 494 for more information and examples.

Gcores Output Filename

Specify the name of the final image.

If executed from the command line, the default is **multi_cores.mca**.

The **.mca** extension is compulsory for MultiCoreArchive final image names. If no extension is provided (or provided a different one) it will be automatically changed to **.mca** and a warning will be issued.

The equivalent option is:

- **-o *filename.mca***

Gcores Shared Modules Import Symbols From Cores

Allows the shared module to import symbols from the cores. This is done utilizing a two-pass build process. The shared module is first built with undefined symbol errors disabled. This allows the cores to then be built importing symbols from the shared module. The shared module is then re-built with undefined symbol errors enabled and importing undefined symbols from the core modules. Because of the nature of this method, users should take caution to prevent the shared module from referencing symbols (defined or undefined) unintentionally that may exist in a core module. Permitted settings for this option are:

- **On** (**-shared_imports**)
- **Off** (**-no_shared_imports**) — [default]

GCores Disable Error Messages Concerning Overlapping Sections

By default **gcores** will warn if any symbol or code in a shared module (built as part of the **-share** option) overlaps with any symbol or address in a core-specific module. Permitted settings for this option are:

- **On** (**-allow_overlap**) — Disables these warnings.
- **Off** (**-no_allow_overlap**) — [default]

GCores Cores Import Symbols From Other Cores

By default the core-specific files are not allowed to reference symbols in other core-specific files. This option attempts to allow such symbol referencing. Use with caution, as duplicate or undefined symbol names used between the components may result in undesirable linkage. Permitted settings for this option are:

- **On** (**-cross_core_imports**)
- **Off** (**-no_cross_core_imports**) — [default]

Gcores Common Link Options

Specifies a list of driver options that are used to build each core and the shared module. These options are used together with the options specified in **-core** or **-share** to build the core or the shared module. The option or options have to be enclosed between braces { } . The equivalent option is:

- **-common { *common options* }**

GCores Driver

Specify the driver/compiler for which to build the cores and the shared module. If omitted, the default is `ccv850`. The equivalent option is:

- **-driver=*driver***

GCores Cpu

Specify the processor for which to build the cores and the shared module. The equivalent option is:

- **-cpu=*cpu***

Keep GCores Temporary Files

Do not remove temporary files.

Permitted settings for this option are:

- **On** (**-keeptempfiles**) — **gcores** does not remove temporary files.
- **Off** (**-nokeeptempfiles**) — [default] **gcores** removes temporary files.

Gcores Exported Absolutes Only

Restricts the symbols that can be imported from either another core or from the shared module to those that have been explicitly exported in the module that defines them.

Three methods are available to export symbols:

- Through the `#pragma ghs exported` pragma directive in the source code. This will export the symbol defined under the `#pragma` line.
- Through the driver option **-exportall=library**. This will export all the symbols of the provided library.
- Through a file **library.ghsexports**. If a file named **library.ghsexports** is found in the same directory as the library **library.a** is generated, the symbols (each one in its own line) specified in the file will be exported.

The symbol names included in the file **.ghsexports** must be the same as the ones existing in the symbol table of the library in question. Please add an underscore `("_")` to the beginning of the variable or function.

For instance, if a library **libshared_0.a** is exporting through its **libshared_0.ghsexports** file the variable `common_lib0_var2` and the function

`common_lib0_f0`, then the symbols to add in **libshared_0.ghsexports** will be (contents of the file **libshared_0.ghsexports**) :

```
_common_lib0_f0
_common_lib0_var2
```

In order to add a dependency to the **.ghsexports** file, use the Builder options **:depends**, **:dependsRelative** or **:dependsNonRelative** in the Core or SharedMemory project file.

Permitted settings for this option are:

- **On** (**-exported_absolutes_only**)
- **Off** (**-no_exported_absolutes_only**) — [default]

GCores Generate Additional Output

Creates the specified output type in addition to the MultiCoreArchive final image. The syntax ***-format=name*** allows you to specify the name of the file. Permitted settings for this option are:

- **S-Record File** (**-srec**, **-srec=*name***) — Generates output file with `.run` extension containing the output of the MultiCoreArchive image as translated by the **gsrec** utility program. For more information about **gsrec**, see "The gsrec Utility Program" on page 554.
- **HEX386 File** (**-hex**, **-hex=*name***) — Generates output file in HEX386 format with `.run` extension containing the output of the MultiCoreArchive image as translated by the **gsrec** utility program with **-hex386** passed to it. For more information about **gsrec**, see "The gsrec Utility Program" on page 554.

Advanced Options

This is a top-level option category.

Target Options

This option category is contained within **Advanced**.

These options control advanced settings that affect compilation for your target. For more common options, see “Target Options” on page 119.

Target Processor

Specifies code generation for a particular target *processor* (see “V850 and RH850 Processor Variants” on page 47).

If you specify a processor, the driver selects the appropriate instruction set and a generic **.ld** file. For more information about these **.ld** files, see “Working with Linker Directives Files” on page 74. You can override the default **.ld** file by specifying an alternate linker directives file elsewhere on the command line. For a more detailed discussion on **.ld** files, see “Configuring the Linker with Linker Directives Files” on page 441.

The equivalent driver option is:

- **Generic CPU** (**-cpu=*cpu***) — To obtain a list of supported processors from the command line, enter **-cpu=?** (or **-cpu=\?** in many shells).

Function Prologues

Controls the manner in which the compiler generates each function prologue and epilogue. Permitted settings for this option are:

- **On** (**-inline_prologue**) — Forces the compiler to use inline code sequences. This option may adversely impact the size of the generated code, so it should only be used when necessary (for example, when the routines may not exist in memory yet).

- **Off** (**-no_inline_prologue**) — Do not force the compiler to use inline code sequences.

By default, the compiler selects the most efficient method based on optimization settings and registers that must be saved.

Switch Tables

Controls how the compiler evaluates switch statements. Permitted settings for this option are:

- **On** (**--switch_table**) — [default] The compiler evaluates switch statements using any means including jump tables.
- **Off** (**--no_switch_table**) — The compiler evaluates switch statements using a series of conventional compare and branch instructions.

Advanced Floating-Point

This option category is contained within **Advanced→Target Options**.

Warn for all floating point types

Permitted settings for this option are:

- **On** (**-Wfloat**) — Warns for any use of floating point. This option has no effect on the code generated by the compiler. This option also does not hide declarations in the standard C headers, so warnings will occur if certain header files are used, even if the declarations in those header files are not needed.
- **Off** (**-Wnofloat**) — [default]

Warn for types double or long double

Permitted settings for this option are:

- **On** (**-Wdouble**) — Warns for any use of double precision floating point or long double precision floating point. This option has no effect on the code generated by the compiler. This option also does not hide declarations in the

standard C headers, so warnings will occur if certain header files are used, even if the declarations in those header files are not needed.

- **Off** (**-Wnodouble**) — [default]

Do not allow types double or long double

Permitted settings for this option are:

- **On** (**-fnodouble**) — Gives a fatal error for any use of double precision floating point as well as long double precision floating point, but allows single precision. All floating point literals must have the `f` suffix, or they are rejected with an error. All declarations that use the types `double` or `long double` are hidden in the standard C header files so that the header files may still be included without error.
- **Off** (**-fdouble**) — [default] Takes no special action with respect to double precision floating point or long double precision floating point.

See also the **-fnone** option in "Floating-Point Mode" on page 122; it completely disables floating point in the compiler and may cause different libraries to be selected. The options **-fnone**, **-fdouble**, and **-fnodouble** are independent. If used together, **-fnone** takes precedence.

Project Options

This option category is contained within **Advanced**.

These options control advanced project configuration settings. For more common options, see "Project Options" on page 140.

Binary Code Generation

Controls binary code generation. By default, the toolchain generates binary object files directly from source code rather than via the creation of assembly language files. Since the assembler is not invoked, binary code generation significantly decreases compile time.

Certain options, such as **-dual_debug**, are incompatible with binary code generation, and binary code generation will default to off if these are selected. Also, binary code generation is automatically disabled on individual files if the compiler determines that the file uses features (such as embedded assembly) that are incompatible with binary code generation.

This option does not control the output file format. To generate a binary output file, see the `-memory` driver option in "Generate Additional Output" on page 220.

Permitted settings for this option are:

- **On** (**-obj**) — Enables direct binary generation.
- **Off** (**-noobj**) — Disables direct binary generation.

Temporary Output Directory

Specifies a *directory* for writing temporary files to. This option is useful if your default temporary directory is on a small file system that might run out of disk space during compilations which involve inlining or template processing. The equivalent driver option is:

- **-tmp=*directory***

(Linux/Solaris) By default, temporary files are written to the first of the following directories that exists:

- The directory specified by the environment variable `TMPDIR`.
- The directory specified by the constant value of variable `P_tmpdir` (usually **/var/tmp** on Solaris and **/tmp** on Linux), which is defined in the operating system header file (usually located in **/usr/include/stdio.h**).
- **/tmp**.

(Windows) By default, temporary files are written to the first of the following directories that exists:

- The directory specified by the environment variable `TMP`.

- The directory specified by the constant value of variable `P_tmpdir` (usually the root directory of the current drive), which is defined in the operating system header file.
- The current working directory.

Temporary Output

Controls whether temporary files generated during compilation are retained. Permitted settings for this option are:

- **Retain** (**-keeptempfiles**) — Prevents the deletion of temporary files after they are used. If an assembly language file is created by the compiler, this option will place it in the current directory instead of the temporary directory.
- **Delete** (**-nokeeptempfiles**) — [default]

Input Languages

Explicitly specifies that a program or archive contains object files or libraries that contain C++. The equivalent driver option is:

- **-language=*language*** — where *language* is one of:
  - `c` (this option has no effect; all programs are assumed to contain C object files.)
  - `cxx`

At link time and archive time, the driver must know if any object files or libraries contain languages other than C in order to select the correct libraries and perform any special processing. If your project includes object files or libraries that were compiled from a different language, you must set **-language=*language*** on that project. This option has no effect when set on an input file.

If your project includes source files that contain other languages and those files appear on the command line, you do not need to set this option. If you invoke the driver directly using **cxv850**, **-language=cxx** is implied.

Non-Standard Output Suffix

Controls the acceptance of non-standard output suffixes. Permitted settings for this option are:

- **Accepted** (**--any_output_suffix**)
- **Not Accepted** (**--no_any_output_suffix**) — [default]

Source Directories Relative to Top-Level Project

Specifies a *directory* in which the builder should search for source files relative to the location of your Top Project. The syntax for this option is:

- **:sourceDirNonRelative=*directory***

This is a **Builder**-only option. There is no equivalent driver option.

Intermediate Output Directory Relative to This File

Specifies the path to where object files (and any custom file types) are written, relative to the location of the current project **.gpj** file. The syntax for this option is:

- **:outputDirRelative=*directory***

This option is most useful for custom tools that do not allow you to specify a directory for intermediate output. For projects that use only Green Hills Tools, use **-object_dir** instead. **-object_dir** overrides this option.

This is a Builder-only option. There is no equivalent driver option.

Binary Output Directory Relative to Top-Level Project

Specifies the path to where final executable output files are written, relative to the location of your Top Project (usually **default.gpj**). The syntax for this option is:

- **:binDir=*directory***

This option controls behavior in the Builder. There is no equivalent driver option.

Binary Output Directory Relative to This File

Specifies the path to where final executable output files are written, relative to the location of the current **.gpj** project file. The syntax for this option is:

- **:binDirRelative=*directory***

This option controls behavior in the Builder. There is no equivalent driver option.

Dependencies Relative to Source Directory

Specifies that the present file *must* be rebuilt if the specified file, relative to the location of your source directory, changes. The syntax for this option is:

- **:depends=*filename***

This option controls behavior in the Builder. There is no equivalent driver option.

Dependencies Relative to This File

Specifies that the present file *must* be rebuilt if the specified file, relative to the location of the present **.gpj** project file, changes. The syntax for this option is:

- **:dependsRelative=*filename***

This option controls behavior in the Builder. There is no equivalent driver option.

Dependencies Relative to Top-Level Project

Specifies that the present file *must* be rebuilt if the specified file, relative to the location of your Top Project (usually **default.gpj**), changes. The syntax for this option is:

- **:dependsNonRelative=*filename***

This option controls behavior in the Builder. There is no equivalent driver option.

Self-Dependency

Specifies that changes to this Green Hills Project (**.gpj**) file do not cause it or any of its children to be rebuilt. The syntax for this option is:

- **:noSelfDepend**

This option controls behavior in the Builder. There is no equivalent driver option.

Suppress Dependencies

Specifies that the present file *must not* be rebuilt if the specified file changes. This option overrides a previous **:depends** option on a **.gpj** file specified in a parent project. To prevent changes to a **.gpj** file from causing a rebuild, use **:noSelfDepend** instead. The syntax for this option is:

- **:nodepends=*filename***

This option controls behavior in the Builder. There is no equivalent driver option.

Output Filename for Generic Types

Specifies a name for the output file of a build item. Use this only for file types that do not accept **-o** or any other option that sets an output name. The syntax for this option is:

- **:outputName=*filename***

This option controls behavior in the Builder. There is no equivalent driver option.

Append Default Extension to Output Filename

Appends the default file extension to the output filename if it does not already have an extension, or replaces the existing extension if it does. For example, this option would replace the extension of an object file with **.o** (Linux/Solaris) or **.obj** (Windows). This option is not usually necessary unless you have manually changed the output name of the file. The syntax for this option is:

- **:appendExtension**

This option controls behavior in the Builder. There is no equivalent driver option.

Commands to Execute Before Associated Command

Specifies commands that are executed before the file is processed. The commands are executed immediately before the processing step for the file. If the file has no processing step, the commands are not executed. The following table lists the processing step for common file types:

File Type	Processing Step
C/C++ Source	Compilation
Assembly Source	Assembling
Program	Linking
Library	Archiving
C/C++ Header	None
Project	None

For example:

- If you set **:preExec** on a program, the commands are run after compiling the program, but before linking it.
- If you set **:preExec** on a header file, the commands are not run.

If you are performing a parallel build, and it is not safe to execute the specified commands while other build operations are running, use this option. See also “Safe Commands to Execute Before Associated Command” on page 258.

The syntax for this option is:

- **:preexec=*command***

If *command* must include information about the file being processed, use a context sensitive variable (see “Context Sensitive Variables” on page 933).

If you only want to execute the command when building your project in certain contexts, use a conditional control statement (see “Conditional Control Statements” on page 911). For example, to run **preProc.bat** on a file only when building on a Windows host, use the following option:

```
{streq(__MULTI_HOST__, "win32")} :preExec="preProc.bat filename"
```

This option controls behavior in the Builder. There is no equivalent driver option.

Safe Commands to Execute Before Associated Command

Specifies commands that are executed before the file is processed. These commands are guaranteed to be performed before the processing step, but may not be performed immediately before. Commands specified with this option must be *safe*; they must not modify any files related to the build (such as projects, source files, or object files). If you are performing a parallel build, use this option instead of **:preExec** to execute safe commands while other build operations are running, because it may speed up the build process. The syntax for this option is:

- **:preexecSafe=*command***

For more information about when files are processed, see "Commands to Execute Before Associated Command" on page 257.

This option controls behavior in the Builder. There is no equivalent driver option.

Commands to Execute Before Associated Command (via Shell)

Specifies commands to be executed in a shell before the file is processed. For more information about when files are processed, see "Commands to Execute Before Associated Command" on page 257.

If you are performing a parallel build, and it is not safe to execute the specified commands while other build operations are running, use this option. See also "Safe Commands to Execute Before Associated Command (via Shell)" on page 259.

The syntax for this option is:

- **:preexecShell=*command***

MULTI prepends the following text to *command* depending on your host's operating system:

- Windows — `COMSPEC /c`, where *COMSPEC* is the value of the `COMSPEC` environment variable (usually `cmd`).

- Linux and Solaris — `/bin/sh -c`

This option controls behavior in the Builder. There is no equivalent driver option.

Safe Commands to Execute Before Associated Command (via Shell)

Specifies commands that are executed in a command prompt or shell before the file is processed. These commands are guaranteed to be performed before the processing step, but may not be performed immediately before. Commands specified with this option must be *safe*; they must not modify any files related to the build (such as projects, source files, or object files). If you are performing a parallel build, use this option instead of **:preExecShell** to execute safe commands while other build operations are running, because it may speed up the build process.

- **:preexecShellSafe=*command***

MULTI prepends the following text to *command* depending on your host's operating system:

- Windows — *`COMSPEC /c`*, where *COMSPEC* is the value of the `COMSPEC` environment variable (usually `cmd`).
- Linux and Solaris — `/bin/sh -c`

This option controls behavior in the Builder. There is no equivalent driver option.

Commands to Execute After Associated Command

Specifies commands that are executed after the file is processed. The commands are executed immediately after the processing step for the file. If the file has no processing step, the commands are not executed. The following table lists the processing step for common file types:

File Type	Processing Step
C/C++ Source	Compilation
Assembly Source	Assembling
Program	Linking

File Type	Processing Step
Library	Archiving
C/C++ Header	None
Project	None

For example:

- If you set **:postExec** on an assembly file, the commands are run after assembling the file, but before linking it.
- If you set **:postExec** on a header file, the commands are not run.

If you are performing a parallel build, and it is not safe to execute the specified commands while other build operations are running, use this option. See also "Safe Commands to Execute After Associated Command" on page 260.

The syntax for this option is:

- **:postexec=*command***

If *command* must include information about the file being processed, use a context sensitive variable (see "Context Sensitive Variables" on page 933).

If you only want to execute the command when building your project in certain contexts, use a conditional control statement (see "Conditional Control Statements" on page 911). For example, to run **postProc.bat** on a project's output file when building on a Windows host, use the following option:

```
{streq(__MULTI_HOST__, "win32")} :postExec="postProc.bat $(OUTPUTFILE)"
```

This option controls behavior in the Builder. There is no equivalent driver option.

Safe Commands to Execute After Associated Command

Specifies commands that are executed after the file is processed. These commands are guaranteed to be performed after the processing step, but may not be performed immediately after. Commands specified with this option must be *safe*; they must not modify any files related to the build (such as projects, source files, or object files). If you are performing a parallel build, use this option instead of **:postExec**

to execute safe commands while other build operations are running, because it may speed up the build process. The syntax for this option is:

- **:postexecSafe=*command***

For more information about when files are processed, see "Commands to Execute After Associated Command" on page 259.

This option controls behavior in the Builder. There is no equivalent driver option.

Commands to Execute After Associated Command (via Shell)

Specifies commands to be executed in a command prompt or shell after the file is processed. For more information about when files are processed, see "Commands to Execute After Associated Command" on page 259.

If you are performing a parallel build, and it is not safe to execute the specified commands while other build operations are running, use this option. See also "Safe Commands to Execute After Associated Command (via Shell)" on page 261.

The syntax for this option is:

- **:postexecShell=*command***

MULTI prepends the following text to *command* depending on your host's operating system:

- Windows — `COMSPEC /c`, where *COMSPEC* is the value of the `COMSPEC` environment variable (usually `cmd`).
- Linux and Solaris — `/bin/sh -c`

This option controls behavior in the Builder. There is no equivalent driver option.

Safe Commands to Execute After Associated Command (via Shell)

Specifies commands that are executed in a command prompt or shell after the file is processed. These commands are guaranteed to be performed after the processing step, but may not be performed immediately after. Commands specified with this option must be *safe*; they must not modify any files related to the build (such as

projects, source files, or object files). If you are performing a parallel build, use this option instead of **:postExecShell** to execute safe commands while other build operations are running, because it may speed up the build process.

- **:postexecShellSafe=*command***

MULTI prepends the following text to *command* depending on your host's operating system:

- Windows — `COMSPEC /c`, where *COMSPEC* is the value of the `COMSPEC` environment variable (usually `cmd`).
- Linux and Solaris — `/bin/sh -c`

This option controls behavior in the Builder. There is no equivalent driver option.

Additional Output Files

Specifies any non-standard output files that are generated during building, so that the **Builder** can include them in file cleanup. The syntax for this option is:

- **:extraOutputFile=*filename***

This option controls behavior in the Builder. There is no equivalent driver option.

'Select One' Project Extension List

For use with the Select One project type. Specifies a list of file extensions that determines which file in the project the Builder chooses. The first extension in the list has the highest priority, and the last extension has the lowest. The syntax for this option is:

- **:select=***arg1*`[,`*arg2*`...]`

For example, if you specify `asm,ppc,c`, the Builder searches your **Select One** project for an `.asm` file first, then a `.ppc` file, and finally, a `.c` file.

If you specify multiple **:select** options for the same project, the builder concatenates them in the order they are specified.

For information about Select One project types, see the documentation about building platform specific programs from the same source files in the *MULTI: Managing Projects and Configuring the IDE* book.

This option controls behavior in the Builder. There is no equivalent driver option.

Pass Through Arguments

Passes unknown options to a command (any tool or utility invoked by the Builder). This is one way of quickly defining custom types. The syntax for this option is:

- **:passThrough=*argument***

This option controls behavior in the Builder. There is no equivalent driver option.

Reference

In a **Reference** project file, this option designates the project (**.gpj**) file that is the target of the reference. Create a **Reference** project and use this option when you want to include one project inside of another without carrying over the including project's options. Permitted settings for this option are:

- **:reference=*root_proj*[;*child_proj*]…[;*target_proj*]**

:reference takes a string argument that is a semicolon-separated list of projects. The Builder uses this list as a path to traverse the project build tree and find the project you want to reference (*target_proj*). Each project in the path is the name of a file listed in the build tree, or "`..`", which refers to the parent of the previous project.

If the first project in the list is "`..`", the Builder traverses the list beginning at the **Reference** project's parent.

If the first project is the name of a file, the Builder traverses the list beginning at the Top Project.

Each project in the list must match the name of a project in the build tree exactly as defined in your project files. If a project's location is defined using a relative or absolute disk path, you must include the disk path in the list. For example, if a

project is defined in its parent as **dir/x.gpj**, the **:reference** list item for this project must be **dir/x.gpj**.

This option controls behavior in the Builder. There is no equivalent driver option.

Pattern of Files to Pull Into Project

Specifies the types of files to include in a project. The syntax for this option is:

- **:autoInclude**

For more information about Auto Include project types, see the documentation about auto include in the *MULTI: Managing Projects and Configuring the IDE* book.

This option controls behavior in the Builder. There is no equivalent driver option.

Options to Apply to Project

Specifies the name of an options file that contains a list of builder or driver options and linker directives (**.ld**) files to use when building a project. This option allows you to share a common set of options and linker directives files among multiple projects. In this file:

- Each line must contain only one option or linker directives file.
- If the line contains an option, it must begin with a whitespace.
- If the line contains a linker directives file, it must begin with the filename.

The lines in this file are processed as if they were present in the **.gpj** file **:optionsFile** is set in. These lines may include macros.

The syntax for this option is:

- **:optionsFile=*filename***

where *filename* is a path to an options file relative to your Top Project. Options files often have the **.opt** extension by convention, but this is not a requirement.

This option controls behavior in the Builder. There is no equivalent driver option.

Toolchain Component Locations

This option category is contained within **Advanced→Project Options**.

These options allow you to vary the location of toolchain components. Note that they should not be used to specify components of a different version than the components that are used by default. These options are normally used only for testing purposes at the advice of Green Hills Software staff.

Alternate Compiler Directory

Specifies an alternate location from which to invoke the compiler. The equivalent driver option is:

- **-Y0,*directory***

Alternate Library Directory

Specifies an alternate location to search for the standard system libraries if they are not found in the default location. The equivalent driver option is:

- **-YL,*directory***

To search for libraries in another location before searching the default location, use the **-L*directory*** option.

If you are not using the default startup code, note that this option changes only the search path for libraries. This means that the linker always takes **crt0.o** from the default location, even if it finds **libstartup.a** and **libsys.a** in another specified location. For more information about how to configure your project to use custom startup code, see the documentation about settings for Stand-Alone programs in the *MULTI: Managing Projects and Configuring the IDE* book.

Alternate Assembler Directory

Specifies an alternate location from which to invoke the assembler. The equivalent driver option is:

- **-Ya,*directory***

Alternate Linker Directory

Specifies an alternate location from which to invoke the linker. The equivalent driver option is:

- **-Yl,*directory***

Optimization Options

This option category is contained within **Advanced**.

These options provide very low-level control over various minor optimizations. These optimizations are generally controlled most effectively by the compiler when you specify an **Optimization→Optimization Strategy**, and you should not use these options unless you have very particular optimization needs. For more common options, see “Optimization Options” on page 145.

Inline Tiny Functions

Controls the inlining of very small functions. Permitted settings for this option are:

- **On** (**--inline_tiny_functions**) — This setting is implied by an **Optimization→Optimization Strategy** setting of **Optimize for Speed** (**-Ospeed**), **-Osize**, or **-Ogeneral** unless **Debugging→Debugging Level** is set to **-g** or **-G** or **C++ Inlining** is set to **-no_inlining**.
- **Off** (**--no_inline_tiny_functions**) — [default]

For more information, see “Automatic Inlining” on page 295.

Inline C Memory Functions

Explicitly controls the inlining of C Memory Functions. Permitted settings for this option are:

- **On** (**-Omemfuncs**) — Enables the inlining of C memory functions unless **-Onone**, **-Odebug**, **-Omoredebug**, or **-Omaxdebug** is enabled.
- **Off** (**-Onomemfuncs**) — [default]

For more information, see "Inlining of C Memory Functions" on page 303.

Inline C String Functions

Controls the inlining of some C string functions. Permitted settings for this option are:

- **Off** (**-Onostrfuncs**) — Disables the inlining of C string functions if they have been enabled by an optimization strategy.
- **On** (**-Ostrfuncs**) — Enables the inlining of C string functions, unless **-Omaxdebug**, **-Omoredebug**, **-Odebug**, or **-Onone** is enabled.

For more information, see "Inlining of C String Functions" on page 303.

Simplify C Print Functions

Controls print function optimization. You may need to set this option to **off** if you define your own print functions (such as `fprintf()`, `fputs()`, `printf()`, `puts()`, etc.) Permitted settings for this option are:

- **Off** (**-Onoprintfuncs**) — Disables print function optimizations if they are enabled by another optimization option (such as an optimization strategy).
- **On** (**-Oprintfuncs**) — Cancels the effect of **-Onoprintfuncs**.

Loop Unrolling

Controls the *Loop Unrolling* optimization. Permitted settings for this option are:

- **Off** (**-Onounroll**) — Disables loop unrolling if it has been enabled by an optimization strategy.
- **On** (**-Ounroll**) — Cancels the effect of **-Onounroll**.

For more information, see "Loop Unrolling" on page 318.

Register Allocation by Coloring

Controls the dynamic storage of variables in registers. Permitted settings for this option are:

- **On** (**-overload**) — [default]
- **Off** (**-nooverload**)

This option has limited effect when **-extend_liveness** is enabled, such as with **-Omoredebug** or **-Omaxdebug**.

For more information, see “Register Allocation by Coloring” on page 330.

Automatic Register Allocation

Controls the automatic allocation of local variables to registers. Permitted settings for this option are:

- **On** (**-autoregister**) — [default]
- **Off** (**-noautoregister**)

For more information, see “Automatic Register Allocation” on page 331.

Common Subexpression Elimination

Controls the *Common Subexpression Elimination* optimization. Permitted settings for this option are:

- **Off** (**-Onocse**) — Disables common subexpression elimination if it has been enabled by an optimization strategy.
- **On** (**-Ocse**) — Cancels the effect of **-Onocse**.

For more information, see “Common Subexpression Elimination” on page 324.

Tail Calls

Controls the *Tail Recursion* optimization. Permitted settings for this option are:

- **Off** (**-Onotailrecursion**) — Disables tail recursion if it has been enabled by an optimization strategy.
- **On** (**-Otailrecursion**) — Cancels the effect of **-Onotailrecursion**.

For more information, see "Tail Calls" on page 325.

Constant Propagation

Controls the *Constant Propagation* optimization. Permitted settings for this option are:

- **Off** (**-Onoconstprop**) — Disables constant propagation if it has been enabled by an optimization strategy.
- **On** (**-Oconstprop**) — Cancels the effect of **-Onoconstprop**.

For more information, see "Constant Propagation" on page 326.

C/C++ Minimum/Maximum Optimization

Controls the *C/C++ Minimum/Maximum* optimization. Permitted settings for this option are:

- **Off** (**-Onominmax**) — Disables minimum/maximum optimization if it has been enabled by an optimization strategy.
- **On** (**-Ominmax**) — Cancels the effect of **-Onominmax**.

For more information, see "C/C++ Minimum/Maximum Optimization" on page 327.

Memory Optimization

Controls *Memory* optimizations. This optimization is enabled by all optimization strategies except **-Onone** and **-Omaxdebug**. If no optimization strategy is selected, this option enables **-Ogeneral**. Permitted settings for this option are:

- **On** (**-OM**)
- **Off** (**-Onomemory**) — [default]

For more information, see “Memory Optimization” on page 327.

Optimize All Appropriate Loops

Turns on all the **General Use** optimizations, together with loop strength reduction, loop unrolling, and loop invariant analysis. This optimization is enabled by **-Ospeed**. If no optimization strategy is selected, this option enables **-Ospeed**. Permitted settings for this option are:

- **On** (**-OL**)
- **Off** (**-Onoloop**) — [default]

For more information, see “Loop Optimizations” on page 315.

Peephole Optimization

Controls peephole optimizations. Permitted settings for this option are:

- **Off** (**-Onopeephole**) — Disables peephole optimizations if they have been enabled by an optimization strategy.
- **On** (**-Opeep**) — Cancels the effect of **-Onopeephole**.

For more information, see “Peephole Optimization” on page 320.

Pipeline Instruction Scheduling

Controls pipeline optimizations. Permitted settings for this option are:

- **Off** (**-Onopipeline**) — Disables pipeline optimizations if they have been enabled by an optimization strategy.
- **On** (**-Opipeline**) — Cancels the effect of **-Onopipeline**.

For more information, see “Pipeline Instruction Scheduling” on page 321.

GNU Compatibility Optimizations

For convenience, Green Hills provides compatibility with the general GNU optimization options. For precise control over the compiler, we recommend that you use Optimization Strategies instead (see "Optimization Strategy" on page 145). Permitted settings for this option are:

- **-O0** — Use **-Onone**.
- **-O1** — Use **-Ogeneral**.
- **-O2** — Use **-Ospeed** and **-Onounroll**.
- **-O3** — Use **-Ospeed**, **-OI**, and **-OB**.

Debugging Options

This option category is contained within **Advanced**.

These options provide control over advanced debugging features. For more common options, see "Debugging Options" on page 156.

Generate Target-Walkable Stack

Creates a frame pointer in register `r28`. You can use this option to perform stack traces if you do not have debug information, however, optimizations might cause gaps in the stack trace. The current V850 and RH850 libraries are built with the **-ga** option to support stack traces. Without this option, some routines do not create a stack frame to reduce code size and improve performance. You must be careful if you decide to use this option, however, as it may produce unexpected complications, such as disabling good leaf procedure optimization. Using **-ga** without **-G** may only guarantee a stack trace and use of the return button in the MULTI Debugger.

Permitted settings for this option are:

- **On** (**-gtws**)
- **Off** (**-nogtws**) — [default]

-Olimit= Without Debug Information

Instructs the compiler to limit optimizations for debugging, even when debug information is not being generated. By default, **-Olimit=peephole** and **-Olimit=pipeline** are only meaningful when debug information is enabled. However, the compiler can still make some attempt to restrict optimizations to assist debugging, even when debug information is never generated. Note that the results will not be as good as they would be if debug information were enabled. Permitted settings for this option are:

- **On** (**-glimits**)
- **Off** (**-noglimits**) — [default]

Consistent code without debug information

Instructs the compiler to produce roughly the same code with the **--no_debug** option as it does with either **-g** or **-G**, although the code is not guaranteed to be identical. Do not use this option with **-g**, **-gs**, or **-G**. Permitted settings for this option are:

- **On** (**-consistentcode**)
- **Off** (**-noconsistentcode**) — [default]

Full Breakdots

Controls the number of breakdots available in the `MULTI` Debugger. This option has an effect only when **-Onone**, **-Odebug**, **-Omoredebug**, or **-Omaxdebug** is enabled. Permitted settings for this option are:

- **On** (**-full_breakdots**) — Increases the number of breakdots available in the MULTI Debugger, but may add additional instructions to your code. This is the default setting if **-Omaxdebug** or **-Omoredebug** is enabled.
- **Off** (**-no_full_breakdots**) — This is the default setting if **-Onone** or **-Odebug** are enabled.

Extend Variable Liveness

Controls access to local automatic variables in the MULTI Debugger. This option has an effect only when **-Onone**, **-Odebug**, **-Omoredebug**, or **-Omaxdebug** is enabled. Permitted settings for this option are:

- **On** (**-extend_liveness**) — Local automatic variables are accessible from the MULTI Debugger after they are written. This is the default setting if **-Omaxdebug** or **-Omoredebug** is enabled.
- **Off** (**-no_extend_liveness**) — Local automatic variables are accessible from the MULTI Debugger as long as they might be read before being written to. This is the default setting if **-Onone** or **-Odebug** are enabled.

Search Path for .dbo Files

Specifies a *directory* to search for **.dbo** debugging information files. The equivalent driver option is:

- **-dbopath** ***directory***

This option is only needed if object files are moved between being compiled and linked.

Search for DBA

Controls searching for **.dba** files in incremental link situations and enables creation of a **.dba** file corresponding to the **.o** file created by an incremental link.

If you want to generate debug information for a relocatable object file (using **-G** and **-relobj**) and the machine that will be performing the subsequent link will not have access to the original **.o** or **.dbo** files, pass the **-search_for_dba** option.

For example, the following set of options creates a **myobj.dba** file that you can distribute with **myobj.o** to provide debug information to MULTI:

```
ccv850 file1.c file2.c -G -relobj -o myobj.o
```

When performing subsequent links with **myobj.o**, continue to pass the **-search_for_dba** option to force **dblink** to search for **myobj.dba** instead of only looking for **myobj.dbo**.

-search_for_dba is not necessary if the original **.dbo** files that comprise **myobj.dba** (**file1.dbo** and **file2.dbo** in the above example) are still available.

Permitted settings for this option are:

- **On** (**-search_for_dba**)
- **Off** (**-no_search_for_dba**) — [default]

Force Frame Pointer

Controls the use of a frame pointer during code generation. Permitted settings for this option are:

- **On** (**-ga**) — Always use a frame pointer. Implied by **Debugging→Run-Time Memory Checks**.
- **Off** (**-noga**) — [default] Use of a frame pointer is not guaranteed.

Scan source files to augment native debug info

Improves the code browsing features of MULTI when using code compiled by third-party compilers. This feature causes the **dwarf2dbo** or **stabs2dbo** Debug Translator to gather information directly from source files. **--scan_source** causes source scanning to occur whenever debugging information is translated as part of the project build. Permitted settings for this option are:

- **On** (**--scan_source**)
- **Off** (**--no_scan_source**)

You can use **--scan_source** if the root of your sources has changed. For more information about **--scan_source**, see “Building and Debugging on Different Hosts” on page 53 and “Source Scanning” on page 53.

Add Extra File to Debug Info

When this option is used, an entry for the specified file is added to the debug information generated by the compiler. This allows the debugger to find that file when you click its name or use the **e** command. This option is useful for including notes or text-based custom files in to the set of files the debugger can display. By default, the builder (command line or graphical) includes all source code and "Text" files in the debug information. The equivalent driver option is:

- **-extra_file=*filename***

Set Maximum DBO Not Found Warnings

Limits the number of "`dbo not found`" warnings to the given number. The default limit is 9. Setting the limit to 0 hides all such warnings. The equivalent driver option is:

- **-warn_dbo_not_found_max=*n***

Wait for Debug Translation Before Exiting

Controls whether the compiler driver waits for debug information to be generated before exiting. The program can be executed immediately after the driver exits, but debugging is not possible until **dblink** has finished generating debug information. Permitted settings for this option are:

- **On** (**-wait_for_dblink**) — [default] The compiler driver waits for **dblink** to finish generating debug information before exiting.
- **Off** (**-do_not_wait_for_dblink**) — The compiler driver does not wait for **dblink**. This option is not supported with **-strip**. When using this option, do not open MULTI or other executables that require debug information until debug translation has completed.

Profiling - Call Count

These options are deprecated and will be removed in a future release.

Controls *call count* profiling, which records how many times each function is called. Call count profiling is not thread safe. Permitted settings for this option are:

- **On with Call Graph** (**-pg**) — Generates call count information that includes the name of each caller. This information can be used to generate a call graph.
- **On** (**-p**) — Generates basic call count information.
- **Off** (**-pnone**) — [default] Disables call count profiling.

Profiling - Legacy Coverage

This option has been replaced by the **-coverage** option (see "Profiling - Block Coverage" on page 156).

Controls legacy *coverage* profiling, which records whether your application contains lines of code that are not used. Permitted settings for this option are:

- **On** (**-a**)
- **Off** (**--no_coverage_analysis**) — [default]

Legacy coverage profiling might not be reliable when used in conjunction with compiler optimizations, link-time optimizations, and options that cause code instrumentation (for example, run-time error checking).

Reducing Debug File Size

This option category is contained within **Advanced→Debugging Options**.

These options allow you to vary the amount of debugging information that will be generated for use with the MULTI Debugger. Note that reducing the amount of information collected may speed up performance, but it may also reduce the depth of analysis available through the Debugger.

Cross-Reference Information

Controls the amount of debugging cross-referencing information that is generated. By default, the compiler generates cross-reference data for each user-defined construct (describing where the construct is declared, defined, read from, written

to, and so on), to allow the Debugger to display cross references. Permitted settings for this option are:

- **Generate for All Objects** (**--xref=full**) — [default] Generates cross-referencing data for all objects.
- **Generate for Object Declarations and Definitions** (**--xref=declare**) — Generates cross-referencing data only for the declarations and definitions of the objects.
- **Generate for Objects in the Global Scope** (**--xref=global**) — Generates cross-referencing data for the objects that are declared in the global scope.
- **Do Not Generate** (**--xref=none**) — Disables all cross-referencing data.

Debugging Information

Controls the generation of debugging information for types and defines that are never used in a file. Permitted settings for this option are:

- **All** (**-full_debug_info**) — Enables the generation of debugging information for types and defines that are never used in a file.
- **Reduced** (**-no_full_debug_info**) — [default] Disables the generation of debugging information for types and defines that are never used in a file.

Native Debugging Information

This option category is contained within **Advanced→Debugging Options**.

These options control the generation of alternate native debugging formats. Use them only if you use third-party tools that require native debugging information. The MULTI Debugger only uses the **.dbo** format.

Generate MULTI and Native Information

Enables the generation of DWARF debugging information in the object file (in addition to the Green Hills **.dbo** format), according to the convention for your target. Permitted settings for this option are:

- **On** (**-dual_debug**)

- **Off** (**-no_dual_debug**) — [default]

Requires that the **Debugging→Debugging Level** be set to Plain or MULTI for the option to have any effect (see "Debugging Options" on page 156).

Convert DWARF Files to MULTI (.dbo) Format

Controls the conversion of DWARF debugging information to the MULTI **.dbo** format. Permitted settings for this option are:

- **On** (**-dwarf_to_dbo**)
- **Off** (**-no_dwarf2dbo**) — [default]

This option will normally be invoked automatically when necessary by the builder.

Convert Stabs Files to MULTI (.dbo) Format

Controls the conversion of Stabs debugging information to the MULTI **.dbo** format. Permitted settings for this option are:

- **On** (**-stabs2dbo**)
- **Off** (**-no_stabs_to_dbo**) — [default]

This option will normally be invoked automatically when necessary by the builder.

Preprocessor Options

This option category is contained within **Advanced**.

These options control advanced preprocessor settings. For more common options, see "Preprocessor Options" on page 163.

Files to Pre-Include

Includes the source code of each specified *filename* at the beginning of the compilation. The equivalent driver option is:

- **-include *filename*[,filename]...**

This can be used to establish standard macro definitions, and so on. The *filename* is searched for in the directories on the include search list.

Assembly Files to Pre-Include

Includes each *filename* at the beginning of any preprocessed assembly files, similar to **-include**. The equivalent driver option is:

- **--preinclude_asm *filename*[,filename]...**

Definition of Standard Symbols

Controls the definition of the set of default symbols. Permitted settings for this option are:

- **Define** (**-stddef**) — [default] You can use this option in a MULTI Project (**.gpj**) file, but there is no equivalent driver option.
- **Do Not Define** (**-nostddef**)

Definition of Unsafe Symbols

Controls the predefinition of preprocessor macros without leading underscores. Permitted settings for this option are:

- **Define** (**--unsafe_predefines**) — Predefine macros both with and without leading underscores.
- **Do Not Define** (**--no_unsafe_predefines**) — [default] Predefine macros only with leading underscores.

When compiling C++ code in Strict Standard Mode, this option has no effect.

Retain Comments During Preprocessing

Controls the retention of comments by the preprocessor. Permitted settings for this option are:

- **Retain** (**-C**)
- **Strip** (**--no_comments**) — [default]

Advanced Include Directories

This option category is contained within **Advanced→Preprocessor Options**.

By default, the Builder and driver search for header files in various standard directories provided with your distribution. These options allow you to vary the location of these standard directories.

Standard Include Directories

Controls whether the standard include directories are searched. Permitted settings for this option are:

- **Search** (**-stdinc**) — [default]
- **Do Not Search** (**-nostdinc**)

C Include Directories

Changes the default location in which the compiler searches for C header files. The equivalent driver option is:

- **-c_include_directory** ***directory***

C++ Standard Include Directories

Changes the default location in which the compiler searches for C++ header files. The equivalent driver option is:

- **-std_cxx_include_directory** ***directory***

System Include Directories

Changes the default location in which the compiler searches for system header files. The equivalent driver option is:

- **-sys_include_directory** ***directory***

C/C++ Compiler Options

This option category is contained within **Advanced**.

These options control advanced C and C++ compilation settings. For more common options, see "C/C++ Compiler Options" on page 165.

C++ Inlining Level

Controls the inlining of C++ functions. Permitted settings for this option are:

- **Maximum** (**--max_inlining**) — Considers all appropriate functions for inlining. Appropriate functions include `inline` functions and functions defined inside of classes.
- **Maximum Unless Debugging** (**--max_inlining_unless_debug**) — Performs maximum inlining unless debugging information is being collected.
- **Standard** (**--inlining**) — [default] Considers only relatively small functions without control flow statements for inlining.
- **Standard Unless Debugging** (**--inlining_unless_debug**) — Performs standard inlining unless debugging information is being collected. If debugging information is being collected, this option disables inlining of C++ functions, but unlike `--no_inlining`, it does not disable `--inline_tiny_functions`, `-OI`, `-Oipsmallinlining`, and `-Oiponesiteinlining`).
- **None** (**--no_inlining**) — Disables inlining of C++ functions. This option also disables **--inline_tiny_functions**, **-OI**, **-Oipsmallinlining**, and **-Oiponesiteinlining**. This option does not disable **-inline_trivial** in the linker.

For more information, see "Additional C++ Inlining Information" on page 305.

Emulated GNU Version

Controls the version of the GNU compiler that the Green Hills compiler provides compatibility with. Permitted settings for this option are:

- **GNU 3.3** (**--gnu_version=30300**) — When using a GNU dialect, provide close compatibility with the GNU 3.3 compiler.
- **GNU 4.3** (**--gnu_version=40300**) — [default] When using a GNU dialect, provide close compatibility with the GNU 4.3 compiler.

-gnu99 Inline Semantics

Controls the method of inlining for the **-gnu99** C Dialect. Permitted settings for this option are:

- **ISO C99 Inline Semantics** (**-fno-gnu89-inline**) — [default] This option causes **-gnu99** to use C99-style inlining for the standard-defined inline keywords, which corresponds to the **GNU99** column in the table at the end of "Manual Inlining In C" on page 295.
- **GNU 89 Inline Semantics** (**-fgnu89-inline**) — This option causes **-gnu99** to use GNU-style inlining, which corresponds to the **GNU** column in the table at the end of "Manual Inlining In C" on page 295.

Prepend String to Every Section Name

Specifies a *string* to prepend to every section name. The equivalent driver option is:

- **--section_prefix** ***string***

Append String to Every Section Name

Specifies a *string* to append to every section name. The equivalent driver option is:

- **--section_suffix** ***string***

WChar Size

Controls the type of `wchar_t`, which is known to the compiler, declared in **stddef.h**, and used in various C library functions. By default, `wchar_t` has a signed 32-bit type (`long` or `int`, depending on the target). Permitted settings for this option are:

- **-wchar_u16** — `wchar_t` is `unsigned short`. This option is not supported with INTEGRITY or Embedded Linux.
- **-wchar_s32** — Cancels the effect of **-wchar_u16**; `wchar_t` is signed and 32 bits.

This option also determines the C and C++ libraries selected by the driver (see "Standard Function Names Converted by the Linker" on page 811) and causes the linker to associate calls to some standard C library functions to alternate names (see "Libraries" on page 770). When this option is set to a non-default value, the deprecated **-no_floatio** is not supported.

longjmp() Does Not Restore Local Vars

Controls whether the values of local non-volatile variables are modified by `longjmp` calls. ANSI C standards require only that the values of volatile variables are not modified. Protecting non-volatile values requires additional overhead. Permitted settings for this option are:

- **On** (**-locals_unchanged_by_longjmp**) — Guarantees that the values of all local variables are not modified by `longjmp` calls.
- **Off** (**-no_locals_unchanged_by_longjmp**) — [default] Guarantees only that the values of volatile local variables are not modified by `longjmp` calls.

Accept GNU __asm__ statements

Provides a way to enable the GNU extended `asm` syntax support. This support is unconditionally enabled in **-gcc** and **--g++** modes, and enabled by default in non-GNU modes. Permitted settings for this option are:

- **On** (**--gnu_asm**) — [default in GNU and non-GNU modes] Enables GNU extended `asm` syntax support for most common use cases. Not all use cases and constraints are supported.

- **Off** (**--no_gnu_asm**) — Disables GNU extended `asm` syntax support. The **-gcc** and **--g++** options override this option, because GNU mode implies support for GNU `asm` syntax.

Identifier Definition

Passes the arbitrary string *string* to the output file. The equivalent driver option is:

- **-ident=*string***

This is the same as using the directive `#pragma ident "`*`string`*`"` in C (see “General Pragma Directives” on page 750). This option can be used to place the date of the source file in the object file.

Identifier Support

Controls support for identifier definition and insertion (via the Builder and driver options above, and through `#pragma ident "`*`string`*`"`). Permitted settings for this option are:

- **On** (**-identoutput**) — [default]
- **Off** (**-noidentoutput**)

Advanced C++ Options

This option category is contained within **Advanced→C/C++ Compiler Options**.

These options control advanced C++ compilation settings.

Demangling of Names in Linker Messages

Controls whether linker error output is written to a file and filtered after the link is finished, or printed to standard output at link time. This option is most commonly enabled for C++ programs; output is written to a file and filtered so that symbol names can be demangled before you view them. The error messages usually indicate undefined or multiply defined symbols. Permitted settings for this option are:

- **On** (**--link_filter**) — [default]
- **Off** (**--no_link_filter**)

C++ Linking Method

Controls the method for generating constructors/destructors at link-time. Permitted settings for this option are:

- **Use Linker** (**--linker_link**) — [default]
- **Create Array of Static Constructors/Destructors for Post-Link Phase** (**--munch**)
- **Do not Generate Constructors/Destructors** (**--nocpp**)

Distinct C and C++ Functions

Controls whether identical C and C++ functions are treated as distinct. Permitted settings for this option are:

- **On** (**--c_and_cpp_functions_are_distinct**) — Treat function types as distinct if their only difference is that one has `extern "C"` linkage and the other has `extern "C++"` routine linkage.
- **Off** (**--no_c_and_cpp_functions_are_distinct**) — [default] Do not treat such functions as distinct.

Support for Implicit Extern C Type Conversion

Controls an extension to permit implicit type conversion in C++ between a pointer to an `extern C` function and a pointer to an `extern C++` function. Also, see "Distinct C and C++ Functions" on page 285. Permitted settings for this option are:

- **On** (**--implicit_extern_c_type_conversion**)
- **Off** (**--no_implicit_extern_c_type_conversion**) — [default]

Assembler Options

This option category is contained within **Advanced**.

These options control advanced assembler settings. For more common options, see "C/C++ Compiler Options" on page 165.

Assembler Warnings

Controls the display of assembler warning messages. Permitted settings for this option are:

- **Display** (**--assembler_warnings**) —
- **Suppress** (**--no_assembler_warnings**) [default]

Linker Options

This option category is contained within **Advanced**.

These options control advanced linker settings. For more common options, see "Linker Options" on page 219. If you use the driver to compile and link your program in multiple steps, pass the same set of options for each step.

Support floating point types in scanf

Controls the inclusion of floating-point `scanf()` code. Permitted settings for this option are:

- **Off** (**-no_float_scanf**) — Inform the compiler and linker that no calls to `scanf()` or its related functions, such as `fscanf()` or `sscanf()`, use floating point, therefore it is not necessary to bring in the support code for floating-point `scanf()`, saving space in the target program. This option is implied by **-fnone**.
- **On** (**-float_scanf**) — [default] Bring in the floating-point support code for `scanf()` if there is any call to `scanf()` or its related functions.

Linker Directives Directory

Specifies the directory containing the default linker directives file or files. This option has no effect if you explicitly specify a linker directives file in your project or on the command line. The equivalent driver option is:

- **-directive_dir=*directory***

Output Archives in 64-Bit Format

Specifies when the archiver should output archives in 64-bit format, and when to print related warning messages. Permitted settings for this option are:

- **When Needed (With Warnings)** (**-auto_large_archive**) — [default] The archiver detects when to output a 64-bit archive and prints a warning when it does. If the archiver takes a 64-bit archive as input but detects that it does not need a 64-bit output file, it prints a warning and does not use the 64-bit format.
- **When Needed (Suppress Warnings)** (**-quiet_auto_large_archive**) — The archiver detects when to output a 64-bit archive. If the archiver takes a 64-bit archive as input but detects that it does not need a 64-bit output file, it does not use the 64-bit format. The archiver does not print any warnings related to 64-bit archives.
- **Always** (**-large_archive**) — The archiver always uses 64-bit archives. The archiver does not print any warnings related to 64-bit archives.
- **Never** (**-no_large_archive**) — The archiver never uses 64-bit archives. The archiver does not print any warnings related to 64-bit archives, but it does print an error if it needs to truncate any values because the archive format does not provide enough space.

Link in Standard Libraries

Controls the linking-in of any Green Hills standard startup files or libraries. User-specified libraries are not affected. Permitted settings for this option are:

- **Link** (**-stdlib**) — [default] Use Green Hills standard libraries.
- **Do Not Link** (**-nostdlib**) — Do not use Green Hills standard libraries, or link against the Green Hills or any user-defined start or end files (see “Start and End Files” on page 226).

  When you use this option, the linker may return an unresolved symbol error due to references to architecture support routines, even if your code has no apparent external library dependencies.

Link in Minimum Libraries

This option causes only the target-specific run-time support library **libarch.a** to be searched. No other Green Hills libraries will be searched. **libarch.a** is the most basic library; it provides definitions of internal symbols that may be required by any file compiled with the Green Hills compilers. This option does not remove the **-L** option that points to the Green Hills library directories, or the start-up files (which can be done with **-nostartfiles**), and the option does not change how the linker directives files are handled.

Permitted settings for this option are:

- **On** (**-minlib**)
- **Off** (**-nominlib**) — [default]

Link Mode

Controls overwriting of an existing executable. By default, the linker overwrites any existing file with the same name as the specified output file. However, the driver can also save a temporary backup copy of the existing file in order to restore it if the link fails. If the link succeeds, the backup copy is silently deleted. Permitted settings for this option are:

- **Overwrite the existing executable** (**--link_output_mode_reuse**) — [default]
- **Save a temporary copy unless the existing file is a symlink** (**--link_output_mode_safe**)
- **Save a temporary copy, even if the existing file is a symlink** (**--link_output_mode_linksafe**)
- **Remove the existing executable before writing a new version** (**--link_output_mode_unlink**)

Linking Sections with Different Types

Allows the linker to combine and link sections from different object files that have different section types (for example, `SHT_PROGBITS` and `SHT_NOBITS`). Without this option, the linker will provide an error message and not complete the link. The equivalent driver option is:

- **--allow_different_section_types**
- **--no_allow_different_section_types**

Physical Address Offset From Virtual Address

Instructs the linker to make `p_paddr` (the physical address) in the ELF program headers equal to `p_vaddr` (the virtual address) plus this offset, *n*. Specifying 0 for the offset makes `p_paddr` equal to `p_vaddr`. The equivalent driver option is:

- **-paddr_offset=*n***

Follow Section

The equivalent driver option is:

- **-follow_section *x=y***
- **-follow_section "*x=y* [*start_expression*] [*attributes*] : [{ *contents* }] [> *memname* | >.]"**

Inserts a definition for section *y* into the section map immediately following the definition of section *x*. The *start_expression*, *attributes*, and *contents* of section *y* may be specified explicitly, using standard linker directives file syntax. If section *x* has the `ABS`, `CLEAR`, or `NOCLEAR` attributes, section *y* inherits them. If the section map contains a section *z* with the attribute `ROM(`*x*`)`, `CROM(`*x*`)`, or `ROM_NOCOPY(`*x*`)`, then a new definition for the section `.ROM`*y*, `.CROM`*y*, or `.ROM_NOCOPY`*y* is inserted following *z* as well.

For more information about these attributes and linker directives file syntax, see Chapter 9, “The elxr Linker” on page 435.

Compiler Diagnostics Options

This option category is contained within **Advanced**.

Redirect Error Output to Overwriting File

Redirects error output to a file, overwriting any existing file, unlike **-stderr=** which just appends output to a file. The equivalent driver option is:

- **-stderr_overwrite=*file***

Because the overwriting takes effect when compiling each individual source file, **-stderr_overwrite** is probably only desired in a makefile situation where the `stderr` output filename is named after the input filename. For example:

```
ccv850 file1.c -stderr_overwrite=file1.err -c
ccv850 file2.c -stderr_overwrite=file2.err -c
```

as opposed to how you might normally use **-stderr**:

```
ccv850 file1.c -stderr=err.out -c
ccv850 file2.c -stderr=err.out -c
```

Support Diagnostics Options

This option category is contained within **Advanced**.

You should use these options only if instructed to do so by Green Hills support staff.

X-Switches

Specifies one or more internal compiler switches. The equivalent driver option is:

- **-X*switch***

EDG Front End Options

Specifies one or more internal EDG front end options. The equivalent driver option is:

- **--option=*option***

Change Assembler

Specifies the assembler to use. Permitted settings for this option are:

- **Use Old Assembler** (**-old_assembler**) — [default]
- **Use New Assembler** (**-new_assembler**)

Debug Information (.dbo) Tracing Diagnostics

Controls the collection of diagnostic information pertaining to a search for **.dbo** files. Permitted settings for this option are:

- **On** (**-dbo_trace**)

This option is useful if you are seeking support for an incident involving missing **.dbo** files.

Dbo File Version

Specifies an older version of **.dbo** debugging information files, for use with an old MULTI Debugger. The equivalent driver option is:

- **--dbo_version=*n***

HTML Compiler Options

This option category is contained within **Advanced**.

This options in this section are deprecated and provided for backwards compatibility only.

Alternate Source Name

This option is deprecated and provided for backwards compatibility only.

Renames an output file. The equivalent HTML compiler option is:

- **-o**

HTML File Compression

This option is deprecated and provided for backwards compatibility only.

Omits HTML file compression. Permitted settings for this option are:

- **No HTML Compression (-c)**
- **HTML Compression (-C)**

Alternate Compression Table

This option is deprecated and provided for backwards compatibility only.

Filename of alternate compression table. The equivalent HTML compiler option is:

- **-p**

Alternate Header Name

This option is deprecated and provided for backwards compatibility only.

Renames the generated include file. The equivalent HTML compiler option is:

- **-h**

Chapter 4

Optimizing Your Programs

Contents

The Green Hills Compiler and toolchain provide many ways to reduce the size and increase the speed of your code. This chapter begins by explaining how to enable optimization strategies and goes on to cover specific optimizations and how they transform code.

Optimization Strategies

Begin optimizing your code by selecting an *optimization strategy*, an option that groups several optimizations together into one setting. The strategies provided by Green Hills strike different balances among speed, size, and debugging ability that cover the needs for many different types of projects and development cycles. To enable a strategy:

Set the **Optimization→Optimization Strategy** option to one of the provided strategies. For details about each strategy, see "Optimization Strategy" on page 145.

Note

Enabling debugging information with the **-g** or **-G** options disables some optimizations in order to make code easier to debug.

The remaining sections in this chapter provide detailed descriptions of the optimizations available with the Green Hills V850 and RH850 compiler. These sections are provided for advanced users who are interested in the code transformations that these optimizations perform, and who want to control individual optimizations.

Inlining Optimizations

The term *inlining* refers to the process of substituting a call to a function or subroutine with the actual content of that function or subroutine. This eliminates the overhead of a subroutine call and may result in faster code.

Typically, the best candidate for inlining is a small, frequently executed function or subroutine that the program calls from only a few locations. Inlining can then increase efficiency in high usage areas without significantly increasing program size.

The Green Hills implementation of inlining is language independent within the Green Hills family of compilers. Routines of one language may be freely inlined into programs of another language. Additionally, the Green Hills compilers perform inlining across modules. Therefore, if a program defines a function within one module, and several modules use that same function, the compiler can inline the function within all the appropriate modules.

What Can Be Inlined

Although you can request that any number of functions be considered for inlining using the methods described below, the compiler will not necessarily inline all such functions. The overall size of the function to be inlined, the calling function, and other factors are always taken into consideration.

Automatic Inlining

This option is enabled by default with all optimizations (**-Osize**, **-Ospeed**, and **-Ogeneral**), unless debug information (**-G** or **-g**) is set. Additionally, the optimization can be enabled explicitly with **--inline_tiny_functions**.

To disable this optimization, set the **Advanced→Optimization Options→Inline Tiny Functions** option to **Off** (**--no_inline_tiny_functions**).

The compiler may automatically inline the following functions:

- Static functions that a file references only one time.
- Very small functions (for example, single simple assignments).

Manual Inlining

This section explains how to manually suggest functions for inlining. You cannot force the compiler to inline any function.

Manual Inlining In C

To manually indicate that the compiler should consider a function for inlining, place an inline keyword immediately before the declaration of the function. A list of inline keywords is provided in the table at the end of this section.

If the compiler decides that a function declared with an inline keyword cannot be inlined, it must either create an out-of-line copy in the module or externally link to another module's out-of-line copy. There are three different methods the toolchain uses to link these functions, depending on how the function was declared. The table at the end of this section describes the declarations, and the following list describes the methods:

- *Exported Inline* — The compiler always generates an out-of-line copy of the function, even if it is not required by the module. If you have two exported inline functions with the same name, the linker will report that you have multiply defined symbols.
- *Imported Inline* — The compiler never generates an out-of-line copy of the function. It always assumes that a copy exists in another module. If you declare an imported inline function and there is no exported inline definition of the function in any translation unit, the linker may report that you have an undefined symbol.
- *Static Inline* — The compiler generates a normal static function only when it is required by the module (for example, if you take the address of the function or if the function is recursive). While you will not get linker errors with static inline functions, there are some important consequences:
  - If you compare two static inline functions in different modules, they will not equate because they have different addresses.
  - Your code size might increase, because there might be multiple out-of-line copies of the function.

If you want to manually inline functions, we recommend that you use static inlining whenever possible. Static inlined functions are portable, do not cause linker errors, and behave exactly the same as static functions when they cannot be inlined. If you want to explicitly control the linkage of one of these functions to decrease code size:

- Make one exported inline definition of the function.
- For all other inline declarations, make imported inline declarations.
- If you do not want to inline the function in a certain translation unit, provide a non-inline external declaration in the translation unit. For example:

```
/* do not inline the function in this translation unit */
extern int inline_max(int, int);
```

The following table lists which linking behavior the compiler uses for inline functions based on your C language dialect and the keyword you use when declaring the function.

Keyword	Dialect		
	GNU	**GNU99**	**C89/C99**
`inline`	exported	imported	imported
`__inline`	exported	imported	static
`__inline__`	exported	imported	static
`extern inline`	imported	exported	exported
`extern __inline`	imported	exported	static
`extern __inline__`	imported	exported	static
`static inline`	static	static	static

Manual Inlining in C++

To manually indicate that the compiler should consider a C++ function for inlining, place the `inline` keyword immediately before the declaration of the function. Member functions that are defined inside `class` or `struct` type definitions are also considered inline, even if you do not declare the function with an inline keyword. Such functions are known as *implicitly inline member functions*.

In C++, the compiler generates an out-of-line copy of the function only when it is required by the module (for example, if you take the address of the function or if the function is recursive). Static data within the function is shared among all copies. While you will not get linker errors if you internally inline functions, there are some important consequences:

- If you compare two functions in different modules, they will not compare as equal, because they have different addresses.
- Your code size might increase, because there might be multiple out-of-line copies of the function.

If you want force the compiler to use just one out-of-line copy of an inline function for the whole program, use the **--instantiate_extern_inline** option and declare that function with `extern inline` (see “Instantiate Extern Inline” on page 206).

Single-pass Inlining

This optimization is enabled by default and cannot be disabled.

Only those functions indicated by the user in the source code will be considered for inlining.

Example 4.1. Inlining Principles

The following program illustrates the basic principles of inlining. The main program in this case contains a simple loop that calls the function `sub()`. The call itself occurs only once in the program code, but the function is executed for each iteration of the loop. The call is easily replaced by the code for `sub()` itself, eliminating both the need for parameter passing and the overhead of a jump-to-subroutine. The reduced overhead per execution results in an increase in program speed.

Initial C source code	Optimized C source code

`__inline void sub(int x) {`
`  printf("x=%d\n",x);`
`  return;`
`}`
`int main() {`
`  int i;`
`  for (i=1;i<10;i++)`
`  sub(i);`
`  return 0;`
`}`
	`int main() {`
	`  int i;`
	`  for (i=1;i<10;i++)`
	`  printf("x=%d\n",i);`
	`  return 0;`
	`}`

Two-Pass Inlining (Intermodule Inlining)

Two-pass inlining involves a special precompilation pass over all of the source files to generate intermediate code for all of the functions to be inlined. This code is stored in files with the **.inf** extension. On the second pass, each source file is compiled normally, but the compiler is given all of the **.inf** files as additional input. In this way, the compiler is able to inline functions across source files and across programming languages.

In two-pass inlining, compiler warnings are suppressed on the first pass so that duplicate warnings will not be issued.

If you have different projects that share the same source file and you are using two-pass inlining, set a different **:outputDir** option for each of the projects to prevent linker errors.

Selecting All Small Functions for Two-Pass Inlining

To control this optimization, use the **Optimization→Intermodule Inlining** option (**-OI** /**-Onoinline**).

This optimization instructs the compiler to consider most small functions as candidates for inlining. You do not need determine which functions to inline, or modify the source code.

Any functions that have been manually specified will also be considered for inlining.

Selecting a Specific Function for Two-Pass Inlining

To specify particular functions for consideration for inlining:

Specify the functions to consider for inlining in the **Optimization→Individual Functions→Inline Specific Functions** option (**-OI=*fn1*[,*fn2*…]**).

This optimization allows you to specify a list of functions to be considered for inlining. This is similar to manual inlining in that the user determines whether or not each function will be inlined, except that it does not require modification of the source code. Like **-OI**, **-OI=** does two-pass compilation.

Any functions that have been manually specified (see "Manual Inlining" on page 295) will also be considered for inlining.

Tip

You can specify both **-OI** and **-OI=fn[,fn2...]** on the same command line.

Example 4.2. Two-Pass Inlining: Combining the Inlining Option Variants

Given the following contents of **file_a.c** and **file_b.c**:

```
file_a.c:
int a_very_big_function()
{
    /* lots of code in here, not shown for brevity... */
    return 0;
}
int b()
{
    return 2;
}

file_b.c:
extern int a_very_big_function(), b();
static inline int c()
{
    return 3;
}
int main()
{
    return a_very_big_function() + b() + c();
}
```

there are several ways to use the inlining option variants.

Given the command line:

```
ccv850 -OI file_a.c file_b.c:
```

- The function `a_very_big_function()` will not be inlined, even though **Optimization→Intermodule Inlining** (**-OI**) is enabled, because the compiler deems it to be too large.
- The function `b()` will be inlined because **Optimization→Intermodule Inlining** (**-OI**) is enabled and `b()` is small.
- The function `c()` will be inlined because it was declared with `inline` in the same module as the caller.
- The functions `a_very_big_function()` and `b()` will be generated out-of-line in module `a`. The function `c()` will not be generated out-of-line because it is declared with `inline` and it was successfully inlined everywhere.

Given the command line:

```
ccv850 -OI=a_very_big_function file_a.c file_b.c
```

- The function `a_very_big_function()` will be inlined because it was specified with **Optimization→Individual Functions→Inline Specific Functions** (**-OI=a_very_big_function**).
- The function `b()` will not be inlined because **Optimization→Intermodule Inlining** (**-OI**) was not enabled.
- The function `c()` will be inlined because it was declared with `inline` in the same module as the caller.
- The functions `a_very_big_function()` and `b()` will be generated out-of-line in module `a`. The function `c()` will not be generated out-of-line.

Given the command line:

```
ccv850 file_a.c file_b.c
```

- The functions `a_very_big_function()` and `b()` will not be inlined because **Optimization→Intermodule Inlining** (**-OI**) was not enabled.
- The function `c()` will be inlined because it was declared with `inline` in the same module as the caller.
- The functions `a_very_big_function()` and `b()` will be generated out-of-line in module `a`. `c()` will not be generated out-of-line.

Varying Inlining Thresholds

To increase the size threshold beneath which the compiler will consider functions for inlining:

Set the **Optimization→Optimization Scope→Inline Larger Functions** option to **On** (**-OB**).

The compiler will still not consider very large functions for inlining unless they are specified using the **Optimization→Individual Functions→Inline Specific**

Functions option (**-OI=*fn1*[,*fn2*…]**). For example, if **Optimization→Intermodule Inlining** (**-OI**) is not inlining sufficiently large functions for your needs, pass:

```
ccv850 -OI -OB
```

If you need even larger functions to be inlined, then you must specify them individually using **Optimization→Individual Functions→Inline Specific Functions** option (**-OI=*fn1*[,*fn2*…]**). For example, to enable automatic two-pass inlining with the higher **Optimization→Optimization Scope→Inline Larger Functions** (**–OB**) threshold, and to additionally inline the function `a_very_big_function()`, pass:

```
ccv850 -OI -OB -OI=a_very_big_function
```

Advantages of Inlining

Inlining is traditionally considered an optimization that improves program speed at the cost of increasing program size. However, although inlining creates more code (because the compiler may generate a single function multiple times within the program), it may actually decrease the overall program size. When a call is replaced by inlined code, the compiler can usually avoid saving and restoring several registers before and after the call. Parameters which normally must be passed on the stack to a called routine can be accessed directly by the inlined routine in their original location. Furthermore, because Green Hills compilers perform inlining before most global optimizations, the process of inlining may significantly enhance the opportunities for additional optimizations. This can result in very efficient code.

For example, if one or more parameter values are constant, large portions of the inlined routine may be reduced or eliminated at compile time, and loops which normally execute a variable number of times may become constant.

Register allocation may improve because inlining eliminates the overhead associated with a call. On most architectures, any function call within a routine reduces the number of registers available for local variables and temporaries. If all function calls can be eliminated by inlining, the number of registers available for variables and temporaries increases.

Pessimistic assumptions made by the compiler when compiling the caller may not be necessary if no call is made. Normally the compiler must assume that when a call is performed, global variables may be changed. This prevents the compiler from

optimizing the values of expressions containing global variables across a call to a function. When the function is inlined, the call is eliminated and the global variables can be optimized more freely.

Note

Source line number information related to inlined routines is deleted. When a program is executed under the control of a source debugger, no source code is available for the inlined routine. Single stepping by source line will cause the entire inlined call to be executed as a single statement. You can, however, debug the inlined call by stepping through the sequence of inlined machine instructions at the point of the source-level call.

Inlining of C Memory Functions

By default, all optimization strategies except `-Onone`, **-Odebug**, **-Omoredebug**, **-Omaxdebug**, and **-Osize** enable the inlining of calls to C memory functions (`memmove()`, `memchr()`, and `memcmp()`) if **string.h** is included. This option has no effect unless an optimization strategy other than the ones mentioned above is enabled. To explicitly enable or disable this optimization:

Set the **Advanced→Optimization Options→Inline C Memory Functions** option to **On** or **Off** (**-Omemfuncs** or **-Onomemfuncs**).

This option may produce multiple copies of memory functions; do not use it if those functions require unique addresses. This option's behavior is not strictly standards-compliant.

Inlining of C String Functions

This optimization enables automatic inlining of calls to C string functions (if **string.h** is included): `strlen()`, `strchr()`, `strrchr()`, `strncpy()`, `strcmp()`, `strncmp()`, `strcat()`, `strncat()`, and `strstr()`. It is usually controlled by the optimization strategy you select, unless you explicitly set **Advanced→Optimization Options→Inline C String Functions** to **On** (**-Ostrfuncs**) or **Off** (**-Onostrfuncs**).

This option may produce multiple copies of string functions; do not use it if those functions require unique addresses. This option's behavior is not strictly standards-compliant.

Note

If **-Omaxdebug**, **-Omoredebug**, **-Odebug**, or **-Onone** is enabled, string functions are not inlined automatically and **-Ostrfuncs** has no effect.

Additional C++ Inlining Information

Although the C++ standard dictates that all member functions defined inside class definitions are implicitly declared inline, the compiler does not consider all such functions for inlining. The reason is that small amounts of code in C++ can expand into dozens of lines in assembly code. If the compiler were to inline every function that is implicitly declared inline, it might create unacceptably bloated code (especially when size is more important than speed). As a consequence, the compiler inlines by default functions composed entirely of straightline code.

If the default C++ inlining behavior does not inline enough functions (the program is too slow) or inlines too many functions (the program is too big), you can override the default behavior with the options documented in "C++ Inlining Level" on page 281.

You can use these options to control C++ inlining for all modules that make up an executable, or module-by-module. You can also combine them. For example, you could enable maximum inlining for modules that contain only small functions and disable all inlining for modules that contain large functions.

To exclude certain functions from inlining within a C++ file, even if they are member functions defined inside a class, place the `#pragma ghs startnoinline` directive before those functions and the `#pragma ghs endnoinline` directive after. The `__noinline` keyword may be used as a synonym, and has the same effect as wrapping a function declaration with the `startnoinline` and `endnoinline pragma` directives.

To exclude template member functions or member functions of template classes, use the `__noinline` keyword by adding it to the source code in the same place where you would normally use the `inline` keyword. The `__noinline` keyword treats an inline member function as though it were defined outside of the class body. This has the effect of generating a single, global, out-of-line copy for that template instantiation. This is preferable to disabling all inlining, because it does not generate duplicate copies in different modules.

To make the `__noinline` keyword have this effect:

Set the **C/C++ Compiler→C++→Keyword Support→Support __noinline Keyword** option to **On** (**--enable_noinline**).

Note

This option is automatically enabled when you set the **Optimization→Optimization Strategy** option to **Optimize for Size** (**-Osize**), unless you have enabled **--instantiate_extern_inline**.

Consequently, you can compile the same source code differently without having to remove the `__noinline` keyword. To disable the effect of the `__noinline` keyword, set the **C/C++ Compiler→C++→Keyword Support→Support __noinline Keyword** option to **Off** (**--disable_noinline**).

The Green Hills C++ library header files make use of the `__noinline` keyword on some member functions which are known to cause large code bloats.

Interprocedural Optimizations

Interprocedural optimizations allow optimizations based on knowledge of functions being called, such as interprocedural alias analysis. **-Ointerproc** does not require that the entire program is available during compilation (see "Wholeprogram Optimizations" on page 310). Because interprocedural optimizations require analysis of many files at the same time, your host machine may run out of memory if your project is large.

To enable interprocedural optimizations:

Set the **Optimization→Interprocedural Optimizations** option to **Interprocedural** (**-Ointerproc**).

Interprocedural options are incompatible with the option **--one_instantiation_per_object**, as well as export templates (enabled with **--export**). Furthermore, interprocedural optimizations are not compatible with the prelinker, and thus require the use of link-once template instantiation.

Related Topics:

- "Link-Once Template Instantiation" on page 705.
- "Exported Templates" on page 709.

This class of optimizations analyzes interaction between functions, and includes:

- **Alias analysis** — Determines what memory can be read and written by function calls. Without this optimization, the compiler assumes that most global memory can be read and written by function calls.

  To disable read-based interprocedural alias analysis:

  Set the **Optimization→Optimization Scope→IP Alias Reads** option to **Off** (**-Onoipaliasreads**).

  To disable write-based interprocedural alias analysis:

Set the **Optimization→Optimization Scope→IP Alias Writes** option to **Off** (**-Onoipaliaswrites**).

To disable interprocedural alias analysis of some standard C functions:

Set the **Optimization→Optimization Scope→IP Alias Lib Functions** option to **Off** (**-Onoipaliaslibfuncs**).

- **Propagation of constant returns from function calls** — When functions return constant values, those values can be propagated to the caller.

  To disable propagation of constant returns from function calls:

Set the **Optimization→Optimization Scope→IP Constant Returns** option to **Off** (**-Onoipconstantreturns**).

  See Example 4.4. Constant Returns on page 309 for a demonstration on how this option works.

- **Inlining small functions** — Small functions can be inlined across modules.

  To disable this inlining of small functions:

Set the **Optimization→Optimization Scope→IP Small Inlining** option to **Off** (**-Onoipsmallinlining**).

The following example displays a program before and after interprocedural alias analysis is applied.

Example 4.3. Interprocedural Alias Analysis

Initial C Source Code:

```
int X = 0;
int Y;
void may_write_X()
{
    Y = 10; // Does not write X
}
void use_global()
{
    X = 1;
    may_write_X();
    if (X != 1)
do_large_computation();
}
```

Optimized C Source Code:

```
int X = 0;
int Y;
void may_write_X()
{
    Y = 10; // Does not write X
}
void use_global()
{
    X = 1;
    may_write_X();
}
```

Example 4.4. Constant Returns

Initial C Source Code

```
int x = 0;
int foo()
{
   x++;
   return 0;
}

void bar()
{
   if (foo())
      x *= 100;
}
```

Optimized C Source Code

```
int x = 0;
int foo()
{
   x++;
   return 0;
}

void bar()
{
   foo();
}
```

Wholeprogram Optimizations

Wholeprogram optimizations analyze program control and data flow at a high level, then optimize your code using several techniques, including:

- Single call-site inlining
- Interprocedural constant propagation
- Interprocedural dead code elimination
- Interprocedural alias analysis

Wholeprogram optimizations can improve program speed and size, often simultaneously.

To enable wholeprogram optimizations:

Set the **Optimization→Interprocedural Optimizations** option to **Wholeprogram** (**-Owholeprogram**).

All function call-sites must be made available to the compiler during each compilation. Therefore, if there are functions or variables in the source code that are referenced from files that are being compiled separately or from assembly files and assembly code, you must use the **-external** option to list those functions to keep invalid optimizations from being applied. For more information, see “Symbols Referenced Externally” on page 150. A weaker set of intermodule optimizations can

be enabled by using **-Ointerproc** (those optimizations are included with **-Owholeprogram**). See "Interprocedural Optimizations" on page 307 for additional information.

Note

All symbols used in assembly files, inline assembly, and separate compilation units must be marked as externally visible to ensure that Wholeprogram optimizations are not incorrectly applied.

You can use **–external**, **–external_file**, `#pragma ghs start_externally_visible`, or the GNU attribute `externally_visible` to specify which symbols are externally visible. Symbols referenced in object files or libraries included during compilation are not required to be listed.

The wholeprogram optimization includes all interprocedural optimizations, as well as:

- **Inlining functions that only have one call-site and are otherwise unreferenced** — If there is only one call to a function, the function call is inlined.

  To disable one-site inlining:

  Set the **Optimization→Optimization Scope→IP One-Site Inlining** option to **Off** (**-Onoiponesiteinlining**).

  This option can be enabled with interprocedural optimizations, although it is not enabled by default. Set the **Optimization→Optimization Scope→IP One-Site Inlining** option to **On** (**-Oiponesiteinlining**). This option is only effective with **-Ointerproc** if most call-sites are visible and you have link-time function deletion enabled (**-delete**).

- **Constant propagation on function parameters** — Conventional constant propagation is performed on the arguments of function calls.

  To disable constant propagation on function parameters:

  Set the **Optimization→Optimization Scope→IP Constant Propagation** option to **Off** (**-Onoipconstprop**).

- **Deletion of unneeded and constant function returns** — When the result of a function call is not required, the return statement is removed.

  To disable deletion of unneeded and constant function returns:

  Set the **Optimization→Optimization Scope→IP Remove Returns** option to **Off** (**-Onoipremovereturns**).

- **Deletion of unneeded and constant parameters** — When certain parameters are not required, the parameters are removed, and function calls do not pass arguments for them.

  To disable deletion of unneeded and constant parameters:

  Set the **Optimization→Optimization Scope→IP Remove Parameters** option to **Off** (**-Onoipremoveparams**).

- **Propagation of constant-value global variables** — When a global variable is initialized to a constant value and never changed, and the variable is not indirectly referenced, the initial value is propagated into the uses of the variable. A limited form of this optimization is also applied to static variables when interprocedural optimizations are enabled.

  To disable the propagation of constant-value global variables:

  Set the **Optimization→Optimization Scope→IP Constant Globals** option to **Off** (**-Onoipconstglobals**).

- **Deletion of unreachable functions** — Code is not produced for functions that are unreachable. A limited form of this optimization is also applied to static functions when interprocedural optimizations are enabled. Functions that have all uses inlined may still be removed.

  To disable the **-Owholeprogram** deletion of unreachable functions:

Set the **Optimization→Optimization Scope→IP Delete Functions** option to **Off** (**-Onoipdeletefunctions**).

- **Deletion of constant-value global variables** — Symbols are not produced for global variables that have their constant values propagated to all of their uses. A limited form of this optimization is also applied to static variables when interprocedural optimizations are enabled.

  To disable the deletion of constant-value global variables:

Set the **Optimization→Optimization Scope→IP Delete Globals** option to **Off** (**-Onoipdeleteglobals**).

- To disable all interprocedural and wholeprogram optimizations:

Set the **Optimization→Interprocedural Optimizations** option to **Off** (**-Onoipa**).

The following example displays a program before and after the interprocedural constant propagation is applied.

Example 4.5. Interprocedural Constant Propagation

Initial C Source Code

```
int can_be_simple(int x, int y)
{
   if (x != 0 || y > 2)
      do_large_computation();
   return x+y;
}
int is_simple()
{
   return can_be_simple(0, 1) + can_be_simple(0, 2);
}
```

Optimized C Source Code

```
int can_be_simple(int y)
{
   return y;
}
int is_simple()
{
   return can_be_simple(1) + can_be_simple(2);
}
```

Using gbuild with Interprocedural Optimizations

When using Wholeprogram optimizations, you must make all parts of your program available to the compiler for analysis, or the compiler might apply those optimizations incorrectly.

To configure the compiler to perform Wholeprogram analysis on all parts of your program, but prevent the compiler from performing Wholeprogram optimizations on certain parts of that program:

- Set **Optimization→Interprocedural Optimizations** to **Wholeprogram** (**-Owholeprogram**) for your program.
- Set **Optimization→Interprocedural Optimizations** to **Analysis Without Optimizations** (**-Oip_analysis_only**) for the parts of the program that you want to exclude from Wholeprogram optimizations.

When you set **Optimization→Interprocedural Optimizations** to **Interprocedural** (**-Ointerproc**), **-Oip_analysis_only** is not necessary because the compiler does not need to analyze the entire program to perform optimizations on each part. If you want the compiler to optimize some parts of a program with **-Ointerproc**, but not others:

- Set **-Ointerproc** for your program.
- Set **Optimization→Interprocedural Optimizations** to **Off** (**-Onoipa**) for each part of the program on which you do not want to have interprocedural optimizations applied.

Note

When the compiler compiles files together, those files cannot use different levels of interprocedural optimization. For example, you cannot enable **-Owholeprogram** on some files and **-Ointerproc** on others. However, you can do so with **-Owholeprogram**, **-Oip_analysis_only**, and **-Onoipa**, or **-Ointerproc** and **-Onoipa** as in the preceding paragraphs.

If a program contains libraries, there are two interprocedural optimization modes that **gbuild** can run in when compiling those libraries:

- When you set **-Ointerproc** for a program, the compiler compiles that program's libraries separately from the rest of the program.
- When you set **-Owholeprogram** for a program, the compiler compiles that program's libraries separately from the rest of the program. **gbuild** also sets **-Oip_analysis_only** for each of those libraries. As a result, the compiler considers information in those libraries when performing Wholeprogram optimizations on the rest of the program, without performing those optimizations on the libraries themselves.

-Owholeprogram is intended for programs only. If you enable it for both a program and an included library, the compiler analyzes and optimizes that library on its own, without regard to the program it is included in.

If you have different projects that share the same source file and you are using interprocedural optimizations, set a different **:outputDir** option for each of the projects to prevent linker errors.

Loop Optimizations

The compiler uses most of its resources to optimize code in the innermost loops in your source files. Therefore, this optimization is most effective for code containing many loop structures that are executed frequently.

This optimization includes:

- strength reduction
- loop invariant analysis
- loop unrolling

Automatic Loop Optimization

This optimization is generally controlled by the setting you specify in the **Optimization→Optimization Strategy** option (see "General Optimizations" on page 320). To control it individually:

Use the **Advanced→Optimization Options→Optimize All Appropriate Loops** option (**-OL** /**-Onoloop**).

Enabling automatic loop optimization enables loop unrolling. If code size is a priority but loop optimization is still desired, you can force off loop unrolling by setting **Advanced→Optimization Options→Loop Unrolling** to **Off** (**-Onounroll**).

Loop Optimizing Specific Functions

To specify particular functions for consideration for loop optimizations:

Specify the functions in the **Optimization→Individual Functions→Loop Optimize Specific Functions** option (**-OL=*fn1*[,*fn2*…]**).

-OL=*fn1*[,*fn2*…] has higher precedence than **-Onoloop**. If **-OL=*fn1*[,*fn2*…]** is specified, loops in functions *fn1* and *fn2* will be optimized even if **-Onoloop** is specified. If **-Oloop** is specified, loops in all functions will be optimized and **-OL=*fn1*[,*fn2*…]** is redundant.

For example, the following command line specifies that loop optimizations will be applied to `sub` and `sub2`:

```
ccv850 -OL=sub,sub2 main.c prog1.c prog2.c
```

Strength Reduction

This optimization is included in loop optimization, which is enabled by either **Advanced→Optimization Options→Optimize All Appropriate Loops** or **Optimization→Individual Functions→Loop Optimize Specific Functions**.

This optimization is applied to arrays subscripted with the loop index. Most compilers access the array element by multiplying the size of the element by the

loop index. The Green Hills compilers store the address of the array in a register and add the size of the array element to the register on each iteration of the loop.

Initial C source code	Optimized C source code

`subr()`
`{`
`  int i;`
`  int q[4];`
`  for (i=0;i<4;i++)`
`  q[i]=i;`
`}`
	`subr()`
	`{`
	`  int i;`
	`  int q[4];`
	`  int *_ptr;`
	`  for (i=0, _ptr=q; i<4; i++)`
	`  *_ptr++ = i;`
	`}`

Strength reduction also applies to multiplying a loop invariant with the loop index. The optimizer replaces a multiply operation with add operations.

Loop Invariant Analysis

This optimization is included in loop optimization, which is enabled by either **Advanced→Optimization Options→Optimize All Appropriate Loops** or **Optimization→Individual Functions→Loop Optimize Specific Functions**.

This optimization examines loops for expressions or address calculations that do not change within the loop. Where such computations are identified, they are relocated to outside the loop and their values are stored in registers.

This optimization is especially useful for reducing code generated to access an array element when the array index does not change within the loop.

Initial C source code	Optimized C source code

`subr()`
`{`
`  int i,j;`
`  int q[4],p[4];`
`  for (i=3;i>=0;i--) {`
`  q[i]=i;`
`  for (j=0;j<4;j++)`
`  p[j]=q[i];`
`  }`
`}`
	`subr()`
	`{`
	`  int i,j;`
	`  int q[4],p[4];`
	`  int tmp;`
	`  for (i=3; i>=0; i--) {`
	`  q[i] = i;`
	`  for (j=0, tmp = q[i]; j<4; j++)`
	`  p[j] = tmp;`
	`  }`
	`}`

Loop Unrolling

This optimization is included in loop optimization, which is enabled by either **Advanced→Optimization Options→Optimize All Appropriate Loops** or **Optimization→Individual Functions→Loop Optimize Specific Functions**.

This optimization duplicates the code in innermost loops to produce more straightline code. This removes much of the loop overhead required for testing for stop condition and branching, and allows for more instruction pipelining and better use of the register allocator. It is most effective when the innermost loop is relatively short, causing minimal increase in code size.

The following example uses a loop with a constant iteration count of 100 and an unrolling factor of 4.

Initial C source code	Optimized C source code

`subr(a)`
`int a[];`
`{`
`  int i;`
`  for (i=0;i<100;i++)`
`    a[i]=i;`
`}`
	`subr(a)`
	`int a[];`
	`{`
	`  int i;`
	`  for (i=0;i<100;i+=4) {`
	`    a[i]=i;`
	`    a[i+1]=i+1;`
	`    a[i+2]=i+2;`
	`    a[i+3]=i+3;`
	`  }`
	`}`

The initial code has:

- 1 comparison
- 1 memory store
- 1 increment
- 1 branch

for a total of 4 instructions per iteration. Assuming each instruction takes exactly one cycle to complete, and assuming time outside of the loop is negligible, this code takes 100 * 4 = 400 cycles to complete.

The optimized code has:

- 1 comparison
- 3 additions
- 4 memory stores
- 1 increment
- 1 branch

for a total of 10 instructions per iteration. Using the same assumptions, this code takes 100 / 4 * 10 = 250 cycles to complete, for a 37.5% improvement.

To increase the maximum size of loops that may be unrolled:

Set the **Optimization→Optimization Scope→Unroll Larger Loops** option to **On** (**-Ounrollbig**).

To disable loop unrolling (recommended when code size is a priority):

Set the **Advanced→Optimization Options→Loop Unrolling** option to **Off** (**-Onounroll**).

General Optimizations

To enable the optimizations in this section:

Set the **Optimization→Optimization Strategy** option to one of the following settings:

- **Maximum Debugging and Limited Optimizations (-Omoredebug)**
- **Optimize for Debuggability (-Odebug)**
- **Optimize for Size (-Osize)**.
- **Optimize for General Use (-Ogeneral)**.
- **Optimize for Speed (-Ospeed)**.

The following options have no effect without an optimization strategy selected. They are all enabled when a strategy is selected, except when noted otherwise.

Peephole Optimization

This optimization is generally controlled by the setting you specify in the **Optimization→Optimization Strategy** option (see "General Optimizations" on page 320). To disable it individually:

Use the **Advanced→Optimization Options→Peephole Optimization** option (**-Opeep /-Onopeephole**).

You may find it helpful to select the **Restrict peephole optimization scope to increase debuggability** setting of the **Optimization→Optimization Scope→Pipeline and Peephole Scope** option (**-Olimit=peephole**).

This optimization identifies common code patterns and replaces them with more efficient code. This includes optimizations such as flow of control, algebraic simplifications, and removal of unreachable code. The compiler only performs this optimization when local code analysis ensures that the results will be correct without further analysis of the surrounding code.

Initial pseudo code	Optimized pseudo code
`mul r1 <- r2, r3` `add r4 <- r1, r4`	`mac r4 <- r2, r3`

Pipeline Instruction Scheduling

This optimization is generally controlled by the setting you specify in the **Optimization→Optimization Strategy** option (see "General Optimizations" on page 320). To control it individually:

Use the **Advanced→Optimization Options→Pipeline Instruction Scheduling** option (**-Opipeline** /**-Onopipeline**).

When debugging, you may find it helpful to select the **Restrict pipeline optimization scope to increase debuggability** setting of the **Optimization→Optimization Scope→Pipeline and Peephole Scope** option (**-Olimit=pipeline**).

This is a low-level time optimization from which a majority of architectures benefit. These architectures (which include all RISC implementations, most new CISC implementations, many DSP processors, and even new application-specific cores) are pipelined, and usually expose at least certain aspects of that pipeline to users.

Instruction scheduling involves reordering instructions for these architectures to make more efficient use of the associated pipelines. The optimizer uses three methods for pipeline instruction scheduling:

- branch scheduling
- basic block scheduling
- superscalar scheduling (for certain architecture implementations only)

These are among the most important optimizations for many architectures, and can increase code speed for the majority of applications.

Example 4.6. Branch Scheduling and Basic Block Scheduling

For this generic machine example, assume that:

- there is one delay slot after a branch
- at least one instruction is required between a load into a register and the use of that register
- the last register of a three operand instruction is the destination

	(a) Before Scheduling Pseudo Code	(b) After Scheduling Pseudo Code
1	`load r0, 0(r5)`	`load r0, 0(r5)`
2	`load r1, 4(r6)`	`load r1, 4(r6)`
3	`add r0, r1, r2`	`add r7, r8, r9`
4	`add r7, r8, r9`	`add r0, r1, r2`
5	`sub r0, r3, r4`	`branch L2`
6	`branch L2`	`sub r0, r3, r4`
7	`nop`	

Branch scheduling consists of two different areas: filling branch delay slots and filling a delay between a comparison and a branch based on that comparison.

In this example, the branch delay slot is filled when instruction 5 from sequence (a) is placed in the `nop` slot after the branch at instruction 6 in sequence (b). This causes the normally unused cycles after the branch to be used on the `sub` pseudo instruction.

Certain architectures require a number of cycles to elapse between a comparison and branch based on that comparison. The method shown above is used to move one or more instructions between the compare and the branch to make use of those cycles.

Basic block scheduling is performed within a basic block (a sequence of instructions which always executes as a group). The instructions within a single basic block are sorted so that the new sequence produces the same result as the original sequence.

In this example, this change is made by reversing the order of the two `add` instructions at lines 3 and 4 between sequence (a) and sequence (b) to allow for the load delay in instruction 2. This will prevent a stall after the load and instead use those cycles on the `add` instruction.

Example 4.7. Superscalar Scheduling

Scheduling for superscalar implementations involves taking into account the existence of multiple execution units. Many architectures are now designed with multiple integer, floating-point, or load-store units in a single implementation. The compiler can schedule the instructions of these architectures so that the instructions associated with multiple execution units can execute at the same time.

For this generic superscalar machine example, assume the same architecture characteristics as in the previous example and, also, that:

- one integer and one floating-point instruction can be issued in the same cycle
- the floating-point instruction must follow the integer instruction with which it is paired

This example uses the same instructions as the last example, with the addition of a floating-point `add` instruction and a floating-point `sub` instruction at lines 6 and 7.

	(a) Before Scheduling Pseudo Code	**(b) After Scheduling Pseudo Code**
1	`load r0, 0(r5)`	`load r0, 0(r5)`
2	`load r1, 4(r6)`	`load r1, 4(r6)`
3	`add r0, r1, r2`	`add r7, r8, r9`
4	`add r7, r8, r9`	`addd f0, f1, f2`
5	`sub r0, r3, r4`	`add r0, r1, r2`
6	`addd f0, f1, f2`	`subd f3, f2, f4`
7	`subd f3, f2, f4`	`branch L2`
8	`branch L2`	`sub r0, r3, r4`
9	`nop`	

In this example, the two floating-point instructions at lines 6 and 7 from sequence (a) have been moved to lines 4 and 6 in sequence (b), making the most efficient use of the multiple instruction issue capability of the integer and floating-point execution units.

Common Subexpression Elimination

This optimization is generally controlled by the setting you specify in the **Optimization→Optimization Strategy** option (see "General Optimizations" on page 320). It is not enabled by **-Onone**, **-Odebug**, **-Omoredebug**, or **-Omaxdebug**. To control it individually:

Use the **Advanced→Optimization Options→Common Subexpression Elimination** option (**-Ocse** /**-Onocse**).

This optimization is implemented when the compiler determines that a previously calculated expression is part of a later expression and none of the variable values in the subexpression have changed. The compiler retains the value of the subexpression in a register for reuse.

Initial C source code	Optimized C source code

`int subr(int x, int y)`
`{`
`  int a, b;`
`  x += a+b;`
`  y += a+b;`
`  if (y < 0)`
`  return (y);`
`  return (x);`
`}`
	`int subr(int x, int y)`
	`{`
	`  int a, b, _v6;`
	`  x+=(_v6=a+b);`
	`  y+=_v6;`
	`  if (y<0)`
	`  return (y);`
	`  return (x);`
	`}`

Tail Calls

Tail call optimization is generally controlled by the **Optimization→Optimization Strategy** option (see "General Optimizations" on page 320). It is not enabled by **-Onone**, **-Odebug**, **-Omoredebug**, or **-Omaxdebug**. To control it manually:

Use the **Advanced→Optimization Options→Tail Calls** option (**-Otailrecursion** /**-Onotailrecursion**).

A function `foo()` is considered for tail call optimization if the last executed statement returns the value of a function call `bar()`. With the optimization, the compiler branches from `foo()` to `bar()`, and `bar()` returns directly to `foo()`'s caller. For example:

C source code	Initial assembly	Optimized assembly

`int bar(void);`
`int foo(void)`
`{`
`  return bar();`
`}`
	`call bar()`
	`restore registers`
	`restore stack frame`
	`return to foo()`
	`restore registers`
	`restore stack frame`
	`return to caller()`
		`branch to bar()`
		`restore registers`
		`restore stack frame`
		`return to caller()`

This option also controls tail recursion optimizations. A function is considered tail recursive if the last statement executed is a call to itself followed by a return statement. This optimization replaces the call with a branch instruction and eliminates the return statement. For example:

Initial C source code	Optimized C source code

`int sum(int n)`
`{`
`  if (n <= 1)`
`  return (1);`
`  else`
`  return (n+ sum(n-1));`
`}`
	`int sum(int n)`
	`{`
	`  int _v3=0;`
	`L1:`
	`  if (n <= 1)`
	`  return _v3+1;`
	`  _v3 += n;`
	`  n--;`
	`  goto L1;`
	`}`

Constant Propagation

This optimization is generally controlled by the setting you specify in the **Optimization→Optimization Strategy** option (see "General Optimizations" on page 320). It is not enabled by **-Onone**, **-Odebug**, **-Omoredebug**, or **-Omaxdebug**. To control it manually:

Use the **Advanced→Optimization Options→Constant Propagation** option (**-Oconstprop** /**-Onoconstprop**).

This optimization replaces a variable with a constant, where the compiler determines that the value of the variable does not change.

Initial C source code	Optimized C source code

`int main()`
`{`
`  int i,a,b;`
`  a = 3;`
`  b = 0;`
 	
`  for (i=0;i<1000;i++)`
`  b += a;`
`  printf("%d\n", b);`
`  return 0;`
`}`
	`int main()`
	`{`
	`  int i,b;`
	`  b = 0;`
	`  for (i=0; i<1000; i++)`
	`  b+=3;`
	`  printf("%d\n", b);`
	`  return 0;`
	`}`

C/C++ Minimum/Maximum Optimization

This optimization is generally controlled by the setting you specify in the **Optimization→Optimization Strategy** option (see "General Optimizations" on page 320). To control it individually:

Use the **Advanced→Optimization Options→C/C++ Minimum/Maximum Optimization** option (**-Ominmax** /**-Onominmax**).

This optimization can generate special code for minimum, maximum, and absolute value expressions of the form:

```
(i < j) ? i : j
(i > j) ? i : j
(i < 0) ? -i : i
```

Memory Optimization

This optimization is generally controlled by the setting you specify in the **Optimization→Optimization Strategy** option (see "General Optimizations" on page 320). To control it individually:

Use the **Advanced→Optimization Options→Memory Optimization** option (**-OM** /**-Onomemory**).

Note

This optimization assumes that non-volatile memory locations only change with explicit store instructions and thus are not affected by any external sources (such as memory-mapped I/O) or interrupt handlers). If you are writing an application that makes use of external sources, we recommend that you use the `volatile` keyword appropriately. Alternatively, you can disable this optimization.

This optimization instructs the compiler to optimize repeated memory reads by placing the value in a local temporary location. Subsequent read operations then refer to the register rather than the actual memory location.

Note

This optimization is only enabled through the **Optimization→Optimization Strategy** option for ANSI C and C++. To enable it when compiling K&R C code, you must set the **Advanced→Optimization Options→Memory Optimization** option to **On** (**-OM**).

To disable memory optimization in ANSI C or C++, use the `volatile` keyword to explicitly declare any object that may change without the compiler's knowledge or control (see "Type Qualifiers" on page 644), or set the **Advanced→Optimization Options→Memory Optimization** option to **Off** (**-Onomemory**).

Dead Code Elimination

This optimization is generally controlled by the setting you specify in the **Optimization→Optimization Strategy** option. It is not enabled by **-Onone**, **-Odebug**, **-Omoredebug**, or **-Omaxdebug**, and it cannot be controlled individually. It removes code that computes values which are never used and eliminates blocks that are determined to be unreachable.

In the following example, after the expression `0*x` is constant folded and the value `0` is constant propagated into the `if` conditional, this optimization is able to determine that the `then` block is unreachable and removes the call `foo()` and the `return c`. Then, it can determine that the value of variable `c` is no longer used, and is able to eliminate the expression `c = 3 * x`.

Initial C source code	Optimized C source code

`int subr(int x)`
`{`
`  int a, b, c;`
`  a = 0 * x;`
`  b = 2 * x;`
`  c = 3 * x;`
`  if (a) {`
`  foo();`
`  return c;`
`  } else {`
`  bar();`
`  return b;`
`  }`
`}`
	`int subr(int x)`
	`{`
	`  int b;`
	`  b = 2 * x;`
	`  bar();`
	`  return b;`
	`}`

Static Address Elimination

This optimization is generally controlled by the setting you specify in the **Optimization→Optimization Strategy** option. It is not enabled by **-Onone**, **-Odebug**, **-Omoredebug**, or **-Omaxdebug**, and it cannot be controlled individually. It instructs the optimizer to assign frequently used static variables to registers within the scope of the function. This eliminates the loads and stores required with memory allocation.

In these examples, the address of the static variable `x` is maintained in a register.

Initial C source code	Optimized C source code

`int subr(int q)`
`{`
`  static int x=0;`
`  x++;`
`  q+=x;`
`  return(q);`
`}`
	`int subr(int q)`
	`{`
	`  static int x=0;`
	`  register int x_ = x;`
	`  x_++;`
	`  q+=x_;`
	`  x=x_;`
	`  return(q);`
	`}`

Note

This optimization is performed not only for locally defined static variables, but also for global variables, as shown in the following example.

Example 4.8. Using Global Variables with Static Address Elimination

Initial C source code	Optimized C source code

`int x = 0;`
`int subr(int q)`
`{`
`  x++;`
`  q+=x;`
`  return q;`
`}`
	`int x = 0;`
	`int subr(int q)`
	`{`
	`  register int x_ = x;`
	`  x_++;`
	`  q+=x_;`
	`  x=x_;`
	`  return(q);`
	`}`

Default Optimizations

The optimizations in this section are very basic and are enabled even if no **Optimization→Optimization Strategy** is specified.

Register Allocation by Coloring

This optimization is enabled by default. To disable it:

Set the **Advanced→Optimization Options→Register Allocation by Coloring** option to **Off** (**-nooverload**).

This optimization permanently maintains a selected set of local scalar variables in registers, based on their frequency of reference and their lifetimes. The compiler uses data flow analysis to determine the lifetime of each variable. The register allocator uses this information to assign different variables within a function to the same register if the lifetimes of the variables do not overlap. This increases the opportunity for allocating variables to registers, which in turn enables the compiler to significantly optimize the code, (since additional memory load and store instructions are not required to reference the variables).

In the following example, the variables `a` and `b` are both assigned to the same register because their lifetimes do not overlap (note that the code could be optimized further, but is not here in order to simplify the example).

Initial C source code	Optimized C source code

`int subr(int x)`
`{`
`  int a,b;`
`  a=x;`
`  b=x*2;`
`  return b;`
`}`
	`int subr(int x)`
	`{`
	`  int a;`
	`  a=x;`
	`  a=x*2;`
	`  return a;`
	`}`

Scalar variables are generally considered for register allocation unless their values are accessed with the address operator (`&`).

Automatic Register Allocation

This optimization is enabled by default. To disable it:

Set the **Advanced→Optimization Options→Automatic Register Allocation** option to **Off** (**-noautoregister**).

This optimization enables the automatic allocation of local variables to registers.

Initial C source code	Optimized C source code

`extern int global;`
`int foo(int e)`
`{`
`  int local = e;`
`  if (global)`
`  local = local * 2;`
`  return local;`
`}`
	`extern int global;`
	`int foo(int e)`
	`{`
	`  register int local = e;`
	`  if (global)`
	`  local = local * 2;`
	`  return local;`
	`}`

Register Coalescing

This optimization is enabled by default and cannot be disabled. It eliminates the additional register-to-register copies required when using a temporary register. When evaluating an expression, the compiler uses the destination register as a work register and organizes the instruction sequence so that the result ends up in that register.

Initial C source code	Optimized C source code

`int fun(int a,int b,int c)`
`{`
`  int ret = a+b+c;`
`  return ret;`
`}`
	`int fun(int a,int b,int c)`
	`{`
	`  return a+b+c;`
	`}`

Constant Folding

This optimization is enabled by default, and cannot be disabled. If the compiler determines that an expression is a constant, it substitutes the constant in place of any reference to the expression. In the following example, the expression `SHRT_MAX/2` is a constant with a value of `16383`.

Initial C source code	Optimized C source code

`#define SHRT_MAX 32767`
`short subr()`
`{`
`  int x;`
`  x=SHRT_MAX/2;`
`  return(x);`
`}`
	`short subr()`
	`{`
	`  int x;`
	`  x = 16383;`
	`  return(x);`
	`}`

Loop Rotation

This optimization is enabled by default and cannot be disabled. It locates the termination test and a conditional branch at the bottom of the loop. This causes the loop to process only one branch instruction on each iteration.

Initial C source code	Optimized C source code

`int subr(int i)`
`{`
`  while (i < 10)`
`  i *= i;`
`  return(i);`
`}`
	`int subr(int i)`
	`{`
	`  goto L7;`
	`  do {`
	`  i *= i;`
	`L7:`
	`  } while (i < 10);`
	`  return(i);`
	`}`

If the compiler determines that the loop is executed at least once, it is entered at the top. If not, the compiler generates an unconditional branch before the loop to the termination test.

Chapter 5

The DoubleCheck Source Analysis Tool

Contents

Introduction

DoubleCheck is a tool for identifying bugs in C and C++ code. It finds code that, when executed, might cause unexpected behavior or harmful corruption of other programs. While a typical compiler alerts you to basic potential code problems, DoubleCheck is a powerful static analysis tool that can analyze large pieces of code spanning many source files and find bugs caused by complex interactions between these pieces of code. Without DoubleCheck, you must track down bugs with intensive debugging after they manifest themselves in a running program. By using DoubleCheck, you can catch these bugs without even running the program.

DoubleCheck works by finding potential execution paths through code, and then determining how the values of variables can change over these paths. The variables can be in memory or registers. The execution paths can proceed through function calls in the same or different files.

DoubleCheck runs at the same time a program is compiled so that no separate tool or separate pass is required. After you enable DoubleCheck in your project, any time you build all or part of the project, DoubleCheck quickly analyzes code and issues errors and warnings illustrating how the bugs it finds might occur.

DoubleCheck can also produce a report file containing extensive information about the bugs it identifies. The report explains where and under what circumstances the bugs exist. It also provides statistics about the project, such as the number of lines of source code, and statistics about the bugs it finds, such as the number of errors per line of source code. DoubleCheck breaks the statistics down by source file and directory. You can view the report using the provided report viewer (see “Enabling DoubleCheck” on page 335), which allows you to sort and mask different error types for a custom view that fits your needs.

DoubleCheck is pre-configured with knowledge of properties of standard library functions. It uses this information to detect errors in code that calls these functions. For example, DoubleCheck checks that when calling the function `free()`, the caller only passes a pointer to memory allocated by functions like `malloc()`.

You can also assign these properties to user-defined functions. For example, if you use a custom memory allocation system, you can assign properties to functions in that system to aid DoubleCheck in looking for memory allocation bugs. By assigning properties to functions, you also reduce the number of *false positives*, which are

potential bugs identified by DoubleCheck that do not cause errors during program execution.

Enabling DoubleCheck

When you enable DoubleCheck, the compiler outputs source analysis errors and source analysis warnings along with the normal compiler errors and warnings as it builds source files. To enable DoubleCheck, use the **-double_check.level=*setting*** option, where *setting* is one of the following:

- **None** (**-double_check.level=none**) — [default] DoubleCheck is disabled.
- **Low** (**-double_check.level=low**) — Looks for all types of bugs, but does not follow execution paths involving function calls. DoubleCheck runs quickly and should not affect the time required to compile files.
- **Medium** (**-double_check.level=medium**) — Looks for all types of bugs, and follows execution paths through calls between functions in all source files. This extra processing requires more time than the low level.
- **High** (**-double_check.level=high**) — Extends the medium level by investigating even more inter-function execution paths. This requires more processing time than the medium level.

Each level performs a different degree of processing. At a higher level, DoubleCheck can identify more bugs than at a lower level, but will require more time to analyze your code.

The medium and high levels require a two-pass compilation. The first pass summarizes the contents of source files. The second pass uses the summaries of all source files when compiling each file, which identifies potential intermodule paths.

Adjusting compiler inlining settings may affect DoubleCheck analysis. For example, enabling inlining may provide DoubleCheck with more execution pathways to analyze, while disabling inlining may provide DoubleCheck with more out-of-line function copies to analyze.

Specifying a DoubleCheck Report File

The most useful way to view DoubleCheck output is to generate a report file and open it using the Web-based report viewer. Report files contain more detailed information than the errors and warnings DoubleCheck issues as source files are built.

To set the name of the report file DoubleCheck generates, use the **-double_check.report=*Report_File.gsr*** option, where *Report_File.gsr* is the name of the file. For example:

```
-double_check.report=Project1.gsr
```

After you have specified a report file and built your application, you can view the report by right-clicking a project in the Project Manager and selecting **View DoubleCheck Report**. See "Viewing DoubleCheck Reports" on page 341 for additional report viewing methods.

Note

The report contains information about the most recent build of source files. If you fix an error and rebuild the corresponding source file, the report will no longer show the error. When working with projects that cause a source file to be built multiple times, we recommend that you generate multiple report files in order to maintain errors from each build.

Using Custom Functions with DoubleCheck

DoubleCheck uses pre-configured properties of standard C and C++ library functions to determine how it should analyze calls to those functions, and when to report an error or warning. When running at the medium or high level, DoubleCheck can sometimes infer these properties for other functions. When DoubleCheck is running at the low level, or when functions exist in an external library, it is much less likely that DoubleCheck will be able to infer these properties. In these cases, you can tell DoubleCheck about the properties of other functions to increase the number of relevant bugs it finds in your source code. At a minimum, you should specify properties of functions that do not return, such as error handling functions, if they abort execution.

The following sections describe how to specify properties of functions to DoubleCheck, and what those properties mean:

Specifying Function Properties in a DoubleCheck Configuration File

You can specify function properties to DoubleCheck by using a configuration file. You do not have to modify source code when using a configuration file. Configuration files specify one property per line, except lines that start with `#`, which are ignored. The format of each line is:

```
Property_Type   Argument_Number   Function_Name
```

where:

- *Property_Type* — The property type (for example, `no_return`). See "Property Types" on page 338 for a list of property types.
- *Argument_Number* — The number of the argument to which this property refers. Specify `0` if this property refers to the return value of the function (for example, `needs_null_check`) or to the function itself (for example, `no_return`).
- *Function_Name* — The name of the function to which you are assigning the property.

  When using C++, do not specify class or argument types. For example, specify `MyFunc` instead of `MyClass::MyFunc(int my_var)`. This will match all functions named `MyFunc`. Use mangled function names to differentiate the parent class or argument types. You can find mangled function names by running **gdump -nx** or **gnm** on the compiled object module (see Chapter 11, "Utility Programs" on page 487).

When naming a DoubleCheck configuration file, use the **.dcc** extension. Specify the configuration file to the compiler with the **-double_check.config=*File_Name.dcc*** option, where *File_Name.dcc* is the configuration file.

Example 5.1. DoubleCheck Configuration File

```
# This file defines properties of a function called
# another_malloc to be exactly the same as the standard malloc
needs_null_check 0 another_malloc
malloced 0 another_malloc
malloc_size 1 another_malloc
```

Specifying Function Properties with Pragma Directives

You can also specify function properties to DoubleCheck by inserting `#pragma` directives in your source code. Insert the `#pragma` directive near the declaration of a function in a header file. We recommend this because you are likely to remember to update the `#pragma` directive if you change the function declaration. When you place a `#pragma` directive in a header file, DoubleCheck reads the directive whenever another file `#includes` that header file. The format of the DoubleCheck `#pragma` directives is as follows:

`#pragma double_check` *`Property_Type  Argument_Number  Function_Name`*

where *Property_Type*, *Argument_Number*, and *Function_Name* have the same meaning as in “Specifying Function Properties in a DoubleCheck Configuration File” on page 337.

Property Types

The following sections describe the property types that can be used in *Property_Type* (see “Specifying Function Properties in a DoubleCheck Configuration File” on page 337). Some of these property types (such as `needs_null_check`) help DoubleCheck find more bugs, while others (such as `no_return`) help avoid false positives by preventing DoubleCheck from analyzing infeasible execution paths.

needs_null_check

Indicates that a function's return value might be `NULL`. A program must verify that the function's return value is not equal to `NULL` before it dereferences that value. The argument number of this property must be `0` because it refers to the function's return value.

malloced

Indicates that a function's return value is allocated and must be deallocated before being overwritten or falling out of scope to avoid resource leaks. The argument number of this property must be `0` because it refers to the function's return value.

If a function with the `malloced` property allocates a resource, that resource should be deallocated by a function with the `frees` property. Use the `malloc_size` property if arguments to this function indicate the size of a memory region that this function allocates.

malloc_size

Indicates that a particular argument specifies the size of a memory region allocated and returned by this function. You can use `malloc_size` twice if the two arguments are multiplied together to specify the size. For example, a function that behaves like `calloc` might use the following properties:

```
malloc_size 1 another_calloc
malloc_size 2 another_calloc
```

frees

Indicates that the specified argument contains a pointer that the function deallocates. If a function with the `frees` property deallocates a resource, that resource should have been allocated by a function with the `malloced` property.

derefed

Indicates that a function argument is unconditionally dereferenced. This is true only if the dereference always happens and is not preceded by a `NULL` check. For example, this `#pragma` directive specifies the correct property for the `df` function:

```
#pragma double_check derefed 1 df
int df(int *x)
{
    return *x;
}
```

no_return_when_zero

Indicates that the function never returns when the specified argument has a value of zero. For example, this `#pragma` specifies the correct property for the `my_check` function:

```
#pragma double_check no_return_when_zero 1 my_check
void my_check(int *x)
{
    if (x == 0) {
        printf("An error has occurred\n");
        exit(1);
    }
}
```

no_return_when_non_zero

Indicates that the function never returns when the specified argument has a non-zero value. For example, this `#pragma` specifies the correct property for the `my_check2` function:

```
#pragma double_check no_return_when_non_zero 1 my_check2
void my_check2(int *x)
{
    if (x) {
        printf("An error has occurred\n");
        while(1);
    }
}
```

no_return

Indicates that a function never returns. The function should always enter an infinite loop or call another `no_return` function such as `exit` or `abort`. The argument number of this property must be `0` because it refers to the function itself.

unsaved

Indicates that a particular argument is never copied to global memory, or saved to any other storage accessible outside a particular call to the function. The function

can dereference the argument's value, but should not deallocate the memory region pointed to by the argument. `unsaved` is useful for helping DoubleCheck catch memory leaks. For example, DoubleCheck can catch the following memory leak even when you run it at the **low** level:

```
#pragma double_check unsaved 1 store
void store(int *x, int y)
{
    *x = y;
}
void buggy()
{
    int *x = (int*)malloc(sizeof(int));
    store(x, 5);     // store does not save x anywhere
}                    // x leaked!
```

uninit

Indicates that a function's return value is a pointer to uninitialized memory and the memory must be initialized before being read to avoid unexpected behavior. The argument number of this property must be `0`, because it refers to the function's return value.

Viewing DoubleCheck Reports

You can configure DoubleCheck to output a report file containing information about source analysis errors and warnings. The report contains more detailed information than the errors and warnings DoubleCheck issues as source files are built. The report file also contains some useful metrics and statistics. Viewing the report files with the viewer is the best way to browse DoubleCheck's results.

Launching the Report Viewer from the MULTI Project Manager

To launch the report viewer from the Project Manager, select **Tools → View DoubleCheck Report** or right-click a project and select **View DoubleCheck Report**. This menu item is only enabled if the project generates a report (that is, if the **-double_check.level=*setting*** and **-double_check.report=*Report_File.gsr*** options are set) and has been built.

Launching the Report Viewer From the Command Prompt

The report viewer is a Web server that you execute using the **gsar** command. You can invoke **gsar** on any DoubleCheck report file. For example:

```
gsar Project1.gsr
```

On Windows hosts, **gsar** launches the default Web browser to display the report. On other hosts, you must use the **-browser=** option to specify a browser to launch. For example:

```
gsar -browser=firefox Project1.gsr
```

gsar Options

Use the following options with **gsar** to control report viewing:

- **-browser=*B*** — Launches Web browser *B* on the main report after starting the Web server. On Windows hosts, **gsar** launches the default Web browser if you do not specify this option.
- **-port=*N*** — Starts the Web server using port *N*. The default port is 13065.
- **-tabsize=*N*** — Substitutes each tab character in your source files with *N* spaces. The default size is 8.
- **-quiet** — Stops **gsar** from printing any status messages.

Controlling DoubleCheck Output

By default, the source analysis errors and warnings generated by DoubleCheck during a build do not interrupt or terminate the build like other compiler-generated errors. To get the most benefit out of DoubleCheck, we recommend that you fix all the bugs that DoubleCheck finds, and then raise the priority of source analysis errors and warnings to the same level as normal compiler-generated errors. Raising the priority prevents future code changes from adding bugs, because the code will fail to build.

Promoting Source Analysis Errors and Warnings

To raise source analysis error and/or warning message priority to stop the build, use **-double_check.stop_build=*setting*** option, where *setting* is one of the following:

- **Off** (**-double_check.stop_build=off**) — [default] Source analysis error and warning messages do not stop the build.
- **Errors Only** (**-double_check.stop_build=errors**) — Only source analysis error messages stop the build.
- **Warnings and Errors** (**-double_check.stop_build=warnings**) — Source analysis error and warning messages stop the build.

Ignoring Source Analysis Errors and Warnings

In certain cases it might be useful to ignore source analysis error and warning messages. For example, if certain parts of your project cannot be modified, you can ignore source analysis errors that occur in those parts. You can ignore source analysis errors and warnings with a build option or by inserting `#pragma` directives in your code.

Note

You might also want to disable these messages if they report code paths that you know are impossible. However, if the code is complex enough to fool DoubleCheck, it is probably complex enough to fool a less experienced programmer into making a change that introduces a bug. Instead of ignoring DoubleCheck, we recommend that you change the code's control flow to avoid the possibility of uncovering problems later.

Using a Builder Option to Ignore Source Analysis Errors and Warnings

To disable a particular source analysis error or warning message for a file or part of a build hierarchy, use the **-double_check.ignore=*number*** option, where *number* is the number of the error or warning you want to ignore. For example, to ignore this error:

```
source analysis error #5: potentially NULL pointer "x"
          dereferenced
```

```
"/temp/file038.c", line 24: pointer set to NULL
    int i, *x = 0;
             ^
"/temp/file038.c", line 33: pointer dereferenced
       *x++ = 4;
         ^
```

use the **-double_check.ignore=5** option.

Using #pragma Directives to Ignore Source Analysis Errors and Warnings

To ignore a particular source analysis error or warning on one line of code, place the following `#pragma` before that line:

```
#pragma double_check ignore number
```

where *number* is the number of the error or warning that you want to ignore, or an asterisk (*) if you want to disable all types of errors and warnings.

To ignore a particular source analysis error or warning on several lines of code, place the following `#pragmas` before and after the lines of code:

```
#pragma double_check start_ignore number
/* your code with ignored errors goes here */
#pragma double_check end_ignore number
```

Using DoubleCheck With Run-Time Error Checking

Enabling run-time error checks may cause DoubleCheck to miss potential bugs that it would otherwise report. To use both DoubleCheck and run-time error checking to debug your program, perform two separate builds, where one build uses DoubleCheck and the other uses run-time error checking.

For more information about run-time error checking, see “Run-Time Error Checks” on page 159.

Source Analysis Error and Warning Messages

DoubleCheck can identify many different types of bugs. The following list explains why you should fix these bugs, how you can fix them, and coding practices you can use to avoid them in the future.

Error Messages

1: Pointer dereferenced before first being NULL-checked

DoubleCheck issues this error when your program calls a function that returns a pointer that is potentially NULL, and then dereferences the resulting pointer without performing a *NULL-check*. A NULL-check is an `if` statement that compares the value of a pointer to `NULL` to determine whether or not it is a NULL pointer.

DoubleCheck knows which standard library functions potentially return NULL. For other functions, DoubleCheck looks to see if you have assigned the `needs_null_check` property to that function.

Importance

Unexpected results might occur when a program dereferences a NULL pointer. The dereference might access the value at memory address 0. The dereference might trigger an exception. If you do not check the pointer before you dereference it, you might lose the ability to properly handle a memory allocation failure (for example, by producing a failure message indicating where in the code the failure occurred and how much memory was requested). Knowing where in code the problem occurred makes it easier to track down the cause of memory exhaustion. Knowing how much memory was requested can help you determine if a negative size value was accidentally passed to a function such as `malloc`.

Avoidance

Use a simple function to check the return value of library functions like `malloc`. For example:

mm.h

```
#pragma double_check malloced 0 my_malloc
#pragma double_check malloc_size 1 my_malloc
void *my_malloc(size_t size);
```

mm.c

```
#include <stdlib.h>
#include <stdio.h>
#include "mm.h"

void *my_malloc(size_t size)
{
    void *p = malloc(size);
    if (p == NULL) {
        printf("Allocation of %lu bytes failed.\n",
                (unsigned long)size);
        exit(1);
    }
    return p;
}
```

2 / 3: Access extends beyond end of / Access extends beyond beginning of

DoubleCheck issues this error when your program adjusts a pointer to a variable or allocated memory region so that it points before the beginning or beyond the end of the region, and then dereferences that pointer.

Importance

Reading past the beginning or end of a variable or allocated memory region returns undefined results. Writing past the end or beginning of an allocated memory region will often corrupt data structures internal to `malloc` and potentially other allocated memory regions. This corruption can cause `malloc` to fail in the future. These bugs can be hard to track down because they can occur long after the write that caused the corruption.

Avoidance

We recommend that you define the size of an array or allocated memory region in only one place in one header file. This enables you to reference one definition in all the source files. Viewing one definition eliminates the chance that someone will change only the part of the code that makes the definition of an array and not how it is accessed elsewhere. We suggest that you do not write code that contains constant sizes:

```
for (index = 0; index < 8; index++)
    array[index] = 0;
```

Instead, find the bounds of an array using the following technique:

```
#define DIM(a) (sizeof(a)/sizeof((a)[0]))
for (index = 0; index < DIM(array); index++)
    array[index] = 0;
```

4: Resource leak through pointer

DoubleCheck issues this error when an execution path exists where a resource is allocated and is not freed, and the pointer to this resource is never saved so it cannot be freed later.

Importance

Over time, resource leaks can lead to resource exhaustion. These bugs can be hard to reproduce and track down because they might only happen when a program is run for a long time or only in a certain way.

Avoidance

To avoid resource leaks, use statically allocated resources instead of dynamic ones. For example, use a global-scoped or static function-scoped variable instead of allocating memory resources with calls to `malloc()`. Statically allocating resources also gives you the benefit of knowing the resource requirements of your application before running it.

If a resource must be dynamically allocated, try creating a clear execution path from the allocation to the point where the resource is freed. If you cannot create a clear execution path, use the C++ `auto_ptr` template. In the following example, `obj` is always freed and never leaked after allocation:

```
#include <memory>
...
void f()
{
    std::auto_ptr<Object> obj(new Object);

    if (error1)
        return;             // obj freed here
    if (error2)
        return;             // obj freed here

}                           // obj freed here
```

5: Potentially NULL pointer dereferenced

DoubleCheck issues this error when a feasible execution path exists between setting a pointer to NULL and dereferencing that pointer.

Many compilers do not give errors for obvious cases, such as:

```
int *x = NULL;
*x = 0;
```

DoubleCheck catches these cases, as well as complex cases found through very long execution paths with complicated control flow.

DoubleCheck does not consider an execution path to be feasible if control flow checks guarantee that the program will modify the pointer's value after it is set to NULL, but before it is dereferenced. For example, in the following code it is known that no feasible path from `POINT A` to `POINT B` exists, and so DoubleCheck will not issue this error:

```
#pragma double_check no_return 0 internal_error

void foo(int x)
{
    int a, b;
```

```
    int *y = NULL;  // POINT A

    switch (x) {
        case 0:
            y = &a;
            break;
        case 1:
            y = &b;
            break;
        default:
            internal_error();
            // #pragma indicates we cannot reach this statement
            break;
    }

    *y = 4;         // POINT B
}
```

Importance

Various problems might arise when you dereference a NULL pointer. See “1: Pointer dereferenced before first being NULL-checked” on page 345 to learn more about these problems.

Avoidance

Avoid any potential control flow path between setting a pointer to NULL and dereferencing that pointer. Initializing pointers to NULL avoids the use of uninitialized values. If there is complicated control flow between initializing a pointer to NULL and dereferencing that pointer, perform a NULL-check before dereferencing it:

```
int *x = NULL;
...
// complicated logic that might initialize x
...
if (x != NULL) {
    ...
    // dereference of x
    ...
}
```

6: Access into deallocated memory

DoubleCheck issues this error when your program accesses a memory region after deallocating it. For example, it is unsafe to free all elements in a list with the following code:

```
void free_list(struct List *l)
{
    while (l != NULL) {
        free(l);
        l = l->next;  // l->next is a read of deallocated memory
    }
}
```

A pointer to a deallocated piece of memory is called a *dangling pointer*.

Importance

When your program deallocates memory, heap management functions such as `malloc` and `free` can modify it. This means that if your program reads from the deallocated memory, you might get unexpected results. If a program writes to the deallocated memory, it might corrupt internal data structures used by heap management functions.

In a multi-threaded process, a memory region deallocated by thread A might be allocated by thread B. This means that if thread A writes to the deallocated memory, it might corrupt thread B's data, and if thread A reads from the deallocated memory, it might return pieces of thread B's data.

Avoidance

Having dangling pointers present in your program can be disastrous. We suggest that you structure your code so that the pointer falls out of scope immediately after the program deallocates the memory to which it points. Alternatively, set the pointer to NULL after the program deallocates the memory to which it points. Here is a much safer implementation for freeing all elements in a list:

```
void free_list(struct List *l)
{
    while (l != NULL) {
```

```
        struct List *tmp = l;
        l = l->next;
        free(tmp);
        // tmp goes out of scope here, so no need to clear
    }
}
```

Here are two other examples that demonstrate how to avoid dangling pointers:

```
void *x = malloc(X_SIZE);
...
free(x); x = NULL;  // dangling pointer now cleared

{
    void *y = malloc(Y_SIZE);
    ...
    free(y);
}                   // dangling pointer now inaccessible
```

7: Memory area deallocated twice

DoubleCheck issues this error when your program deallocates the same memory region twice without allocating it again.

Importance

Deallocating the same memory region twice might cause heap management functions such as `malloc` and `free` to function improperly.

Avoidance

Avoid dangling pointers using techniques discussed in the Avoidance section of "6: Access into deallocated memory" on page 350.

9: Attempt to deallocate stack memory

DoubleCheck issues this error when your program deallocates stack memory.

Stack memory is memory that your program uses to hold local variables and function return addresses. The compiler generates code that manages stack memory. *Heap memory* is memory that heap management functions such as `malloc`, `free`, `new`, and `delete` use when allocating and deallocating regions. You should never pass a pointer to stack memory to a heap management function.

Importance

Deallocating stack memory causes heap management functions like `malloc`, `free`, `new`, and `delete` to function improperly.

Avoidance

Avoid using the same pointer variable to refer to both stack memory and heap memory. For example, if a variable contains pointers to heap memory 99 percent of the time and pointers to stack memory 1 percent of the time, you might make the mistake of unconditionally deallocating the memory to which the pointer points.

For example, use a separate variable for heap allocation and deallocation so that they are never confused:

```
int *heap_buf = NULL, *buf, stack_buf[10];
if (...)
    buf = stack_buf;
else {
    buf = heap_buf = (int*)malloc(...);
    if (buf == NULL)
        return;
}
...
// complicated logic that uses buf
...
free(heap_buf);
```

10: Inactive stack address potentially returned

DoubleCheck issues this error when a function returns a pointer to a local variable. For example:

```
char *int_to_string(int val)
{
    char buffer[11];
    for (int i = 9; i >= 0; i--) {
        buffer[i] = '0' + val % 10;
        val /= 10;
    }
    buffer[10] = '\0';
    return buffer;
}                       // inactive stack memory returned!
```

Importance

When a function returns, any pointers to non-static local variables in that function become pointers to inactive stack memory. If the function returns one of these pointers and the caller dereferences it, you might get unexpected results because another function call or interrupt handler might have changed the inactive stack memory. If your program reads this memory, it might not return the desired value. If your program writes to this memory, it might corrupt stack memory that contains values critical to correct program execution.

Avoidance

Do not write functions that return pointers to local variables. One way to avoid accessing inactive stack memory is to declare the variable as static, however, this technique encourages accidental overlapping uses of the static variable. For example, if you modify the `int_to_string()` example so that `buffer` is declared `static`, this function does not correctly print two different integers:

```
void print_two_ints(int a, int b)
{
    printf("%s %s\n", int_to_string(a), int_to_string(b));
}
```

To avoid returning pointers to local variables, require callers to pass in pointers so that the caller can decide how to allocate the memory.

11: Write to potentially read-only memory

DoubleCheck issues this error when the program writes to read-only memory. For example:

```
char *get_string()
{
    return "Hello";
}
char *get_lower_case_string()
{
    char *st = get_string();
    for (char *c = st; *c != '\0'; c++)
        *c = tolower(*c);      // writing to read-only memory!
    return st;
}
```

Importance

Writing to read-only memory can cause exceptions if the memory is write-protected. Writing to read-only memory can cause unexpected future results. For example, calling `get_lower_case_string()` changes what future calls to `get_string()` will return.

Avoidance

Use the `const` keyword when declaring pointers to read-only memory. Using this keyword eliminates the chance of writing through the pointers. To correct the preceding example, define `get_string` with return type `const char *`.

12 / 13: Memory deleted using ``delete[]`` instead of ``delete`` / Memory deleted using ``delete`` instead of ``delete[]``

DoubleCheck issues this error when your program allocates memory with `new[]` and deallocates it with `delete`, or when it allocates memory with `new` and deallocates it with `delete[]`.

Importance

If you mix an allocator with the wrong deallocator, your program may not call the appropriate destructor. This mismatch may not seem like a bug if, for example, a destructor is not present. It is a better coding practice to use the proper `new` and `delete` calls in case you add a destructor in the future.

Avoidance

Always use `new` and `delete` calls that match. To strictly enforce this rule on a certain class, overload the unwanted `new` or `delete` calls so that they output an error message if you use them by mistake.

14: Read of potentially uninitialized variable

DoubleCheck issues this error when a local variable is never written with a value, but is read on a particular execution path.

Importance

Relying on an uninitialized value can produce unexpected results. The uninitialized value might not cause problems 99 percent of the time, which can make reproducing failures very difficult.

Avoidance

It is good practice to always initialize local variables. This practice should not degrade performance since the Green Hills compiler is very good at eliminating unused initializers.

17: Read of uninitialized memory

DoubleCheck issues this error when memory is allocated but not initialized (for example, by `malloc()`) and some piece of that memory is read prior to being initialized.

Importance

Relying on uninitialized memory values can produce unexpected results. The uninitialized value might not cause problems most of the time, making it very difficult to reproduce failures.

Avoidance

It is good practice to always initialize memory. In C code, try using `calloc()` instead of `malloc()`. In C++ code, make sure every non-static data member is initialized or constructed in constructors.

Warnings

Warnings indicate code that will not explicitly cause unexpected behavior, but is worth investigating. The code might not be optimal or might contain checks for problems that can arise in a different form.

8: Useless NULL-check after pointer has been dereferenced

DoubleCheck issues this warning when your program performs a NULL-check on a pointer after it has already dereferenced that pointer.

Importance

When you perform a NULL-check, it indicates that you think the pointer will be NULL in some cases. If you perform this check after dereferencing the pointer, this in turn indicates that your program will dereference a NULL pointer in those cases. Your program might exhibit unexpected behavior if it dereferences a NULL pointer.

Avoidance

To avoid this warning and related potential problems, move the NULL-check before the dereference, and use the NULL-check to avoid executing the dereference when the pointer is NULL.

15: Value set but never read / Value overwritten before being read

DoubleCheck issues this warning when your program writes a variable with a value, but no execution paths leading from this write perform a read of that variable. Execution paths leading from this write might write the variable again, destroying the first value and making it impossible to read.

DoubleCheck does not issue this warning when you clear a dangling pointer using the following code:

```
free(x);
x = NULL;
```

Importance

This warning indicates that you might have intended to use the value written to the variable, so some functionality might be missing from your program.

Avoidance

Avoid writing to a variable unless you intend to use the value stored in the variable for some purpose.

16: Pointer dereferenced unconditionally after NULL-check

DoubleCheck issues this warning when your program performs a NULL-check on a pointer, and then dereferences that pointer independent of the result. To avoid producing these warnings on infeasible execution paths, assign properties to any functions that affect execution paths. For example, the following piece of code will produce a warning without the `#pragma` directive:

```
#pragma double_check no_return 0 my_error_handler
    if (x == NULL)
        my_error_handler();
    *x = 0;
```

Importance

When you perform a NULL-check, it indicates that you think the pointer will be NULL in some cases. If you dereference a pointer after a NULL-check, it indicates that you will dereference a NULL pointer in those cases. Your program might exhibit unexpected behavior if it dereferences a NULL pointer.

Avoidance

When a pointer is found to be NULL, it is good practice to prevent your program from dereferencing that pointer. You can do this by stopping execution with a call that does not return (such as `exit`, `abort`, or a user-defined error handling function), or by jumping to a point later in the control flow that bypasses the dereference.

Part II

Using Advanced Tools

Chapter 6

The ease850 Assembler

Contents

The assembler translates ASCII files containing assembly language instructions into binary files containing relocatable object code.

Running the Assembler from the Builder or Driver

We recommend that you invoke the assembler via the Builder or driver. Simply add assembly files to the Project Manager's source pane, or to the driver's command line. If your assembly file ends with **.800** rather than **.s**, the Builder or driver first invokes the preprocessor to take advantage of preprocessor facilities (such as `#include` and `#define`) that are not normally available to an assembly language programmer. To run the preprocessor on all assembly files (including those with a **.s** extension):

Set the **Assembler→Preprocess Assembly Files** option to **On** (**-preprocess_assembly_files**).

The Builder and driver accept a limited number of assembler-specific options directly (see "Assembler Options" on page 216). To pass the full range of assembler options listed in this chapter:

Enter the required assembler option (see "Assembler Options" on page 364) in the **Assembler→Additional Assembler Options** option (**-asm=*assembler_option***).

When using the driver, you can either precede each assembler option with **-asm=**, or list multiple options separated by whitespace and enclosed within quotes. Hence the two following command lines are equivalent:

```
ccv850 -asm=-l -asm=-w stack.s -c -o stack.o
ccv850 -asm="-l -w" stack.s -c -o stack.o
```

Alternatively, if you have a large number of options to pass to the assembler, you can place them in a text file and enter its filename in the **Assembler→Assembler Command File** option (**-asmcmd=*file***).

Generating Assembly Language Files

At times, you might want to generate and review assembly language output from a high-level language file. To do this:

From the driver, pass the **-S** option. For example, the following command line would compile **main.c** and place the assembly language output in the file **main.s**:

```
ccv850 -S main.c
```

Running the Assembler Directly

Whenever possible, you should invoke the assembler indirectly using the driver or Project Manager. If you must invoke the assembler directly, the syntax is:

```
ease850 [options] [input_file]
```

where *options* are assembler options. When running the assembler directly, do not use the **-asm=** option.

Assembler Options

For information about using the following options with the Builder or driver, see “Running the Assembler from the Builder or Driver” on page 362.

V850 and RH850-Specific Assembler Options

-asm_far_jumps

Instructs the assembler to only use the largest possible JR and JARL jump instructions, when it is not otherwise forced to use a smaller instruction.

-bit_inst_error

Causes the assembler to fail with an error if it encounters a bit manipulation instruction (`TST1/SET1/NOT1/CLR1`).

-cpu=*cpu*

Specifies a particular V850 and RH850 target processor. For a complete list of supported processors and the options to be passed to specify them, see “V850 and RH850 Processor Variants” on page 47.

-noexpandbranch

Disables branch expansion. An error occurs when a branch that would normally require expansion for its destination to be in range is encountered.

-nofpu

Disables support for all FPU instructions. FPU instruction usage will result in an error.

-nofpu_double

Disables support for double-precision FPU instructions only. Double-precision FPU instruction usage will result in an error.

-fpu30

-nofpu30

Enables or disables support for RH850 FPU 3.0 instructions.

-nomacro

Disables macro expansion. An error occurs when a macro is encountered.

-noshortbranches

Instructs the assembler to only use the largest possible instruction when assembling a BCOND local branch instruction.

-rh850_simd

-norh850_simd

Enables or disables support in the assembler for RH850 SIMD instructions.

General Assembler Options

@file

Passes commands listed in *file* to the assembler.

-cpu=*cpu*

Specifies the target *cpu*.

-help

Prints a help message.

-I *dir* (an uppercase letter "`i`")

Searches directory *dir* for files specified in `.include` directives.

-list[=*file*]

Generates a source listing. If `=file` is not specified, then the listing file is written to a file with the same name as the original file, but with an **.lst** extension. If `=file` is specified, then the listing is written to *file*.

-o *file*

Sets the name of the output object file to *file*. Without the **-o** option, the assembler produces an object file that has the name of the assembly language file with an **.o** extension. For example, **foo.o** is produced for **foo.s**.

-source=*name*

Overrides the source file's *name*.

-V

Prints the assembler version number to the standard output.

Assembler Syntax

Identifiers

Identifiers, or symbols, are composed of letters, digits, and the following special characters: dollar sign (`$`), period (`.`), and underscore (_). The first character of an

identifier must be alphabetic, or one of these three special characters. Uppercase and lowercase letters are distinct; the identifier `abc` is not the same as the identifier `ABC`. Characters in reserved symbols, such as directives, machine instructions, and registers, are not case-sensitive.

Examples

The following table shows some valid and invalid identifiers:

Identifier	Validity
`*star`	Invalid (may not contain `*`)
`123test`	Invalid (may not start with digit)
`f-ptr`	Invalid (may not use hyphen)
`_hello`	Valid
`LABEL`	Valid
`test4`	Valid

Reserved Symbols

Operator names are reserved. For a list of operators, see “Integral Expression Operators” on page 370.

In addition, the names of the special purpose registers (`SPR`'s) are reserved. Consult the appropriate microprocessor user's manual for a list of implemented `SPR`'s.

Constants

Assembler constants can be numeric, character, or string constants.

Numeric Constants

A sequence of digits defines a numeric constant. Constants can be specified in hexadecimal, octal, or binary formats, or as floating-point numbers, by preceding the number with one of the following special prefixes:

Type	Prefix	Example
hexadecimal	`0x`	`0xb0b`
octal	`0`	`0747`
binary	`0b`	`0b110011`
floating-point	`0f`	`0f6.02e+23`

Floating-point numeric constants are supported in limited cases, such as by the `.double` and `.float` directives.

String Constants

Some directives take a string constant as one or more of their arguments. A string constant consists of a sequence of characters enclosed in double quotation marks (`"`). A string constant can contain any ASCII character (including ASCII null), except newline. The null character is not appended to strings by the assembler, as it is in C or C++.

Character Constants

A character constant can be used in any location where an integer constant is needed.

A character constant consists of the following items, enclosed within single quotation marks (`' '`):

- either a single ASCII character, or
- a backslash character (`\`) and one of the escape character values

A character constant is considered equivalent to the ASCII value of the character or escape sequence.

For example, the character constant `'a'` is equivalent to the decimal integer 97, and the character constant `'\r'` is equivalent to the decimal integer 13.

Character Escape Sequences

Character and string constants consist of ASCII characters. The ASCII backslash (`\`) is used within character and string constants to escape the quotation marks and

to specify certain control characters symbolically. A backslash followed by any non-escape character is equivalent to that character (for example, `\a` is equivalent to `a` because `\a` is not an escape sequence).

Escape Sequence	Character	ASCII value
`\0`	null	0 (`0x0`)
`\b`	backspace	8 (`0x8`)
`\t`	horizontal tab	9 (`0x9`)
`\n`	newline	10 (`0xA`)
`\v`	vertical tab	11 (`0xB`)
`\f`	form feed	12 (`0xC`)
`\r`	return	13 (`0xD`)
`\`*`nnn`*	octal value *nnn*	n/a
`\x`*`nn`*	hexadecimal value *nn*	n/a
`\'`	single quotation mark	39 (`0x27`)
`\"`	double quotation mark	34 (`0x22`)
`\\`	backslash in a constant or string	92 (`0x5C`)

Source Statements

An assembler source statement consists of a series of fields delimited by spaces and/or horizontal tabs, in the following format:

```
[label:] [operator [arguments]] [comments]
```

Label Field

See “Labels” on page 376.

Operator Field

The operator field starts with the first non-whitespace character after the optional label field and is terminated by the first whitespace character or line terminator encountered after the operator. An operator is any symbolic opcode, directive, or

macro call. In order to avoid ambiguity with the label field, operators should not begin in column 1.

Argument Field

The argument field starts with the first non-whitespace character following the operator field and ends with a line terminator or the beginning of a comment field. Arguments qualify the opcode, directive, or macro call.

Comment Field

The comment field is optional and begins with "--". The assembler ignores all characters to the right of the comment symbol until the end of the line.

Continuation Lines

The assembler does not support continuation lines.

Whitespace

Whitespace consists of spaces and horizontal tabs.

Line Terminators

Assembly input lines are terminated by a line feed or a carriage return and line feed combination. A carriage return or form feed is insufficient.

Expressions

Assignment Statements

An expression is assigned to a symbol by an assignment statement in one of the following forms, where *ident* is the symbol name:

- *ident* =[:] *const-expr*

  An assignment in the form = defines a local constant, while an assignment in the form =: specifies that the symbol is also global. For example:

  ```
  chair = 8             -- set chair to 8 (local constant)
  table =: 5            -- set table to 5 (global)
  ```

- *ident* .equ *const-expr*

  You cannot use this directive multiple times for the same symbol. For example:

  ```
  table .equ 8      -- set table to 8
  table .equ 5      -- try to reset table to 5 (illegal)
  ```

- .set *ident*,*const-expr*

  For example:

  ```
  .set table,8      -- set table to 8
  .set table,5      -- reset table to 5
  ```

const-expr must be an absolute or relocatable expression (for more information, see “Expression Types” on page 375).

Integral Expression Operators

A number of operators are available to form expressions. The type `unary` indicates that the function is recognized when the operator has only a right operand. The type `binary` indicates that the operator has two operands, one to the left of the operator and one to the right.

Operator	Type	Meaning
~	unary	Bitwise-not operator.
-	unary	Negative.
+	binary	Add.
-	binary	Subtract.
*	binary	Multiply.
/	binary	Divide.
%	binary	Modulo.
&	binary	Bitwise-and operator.
|	binary	Bitwise-or operator.
^	binary	Bitwise-exclusive-or operator.
=,==	binary	Equality (0 or 1).
!=	binary	Inequality (0 or 1).
>,>=,<,<=	binary	Signed compare (0 or 1): greater than, greater than or equal to, less than, less than or equal to.
:UGT:, :UGE:, :ULT:, :ULE:	binary	Unsigned compare (0 or 1): greater than, greater than or equal to, less than, less than or equal to.
<<,>>	binary	Signed shift left, signed shift right.
:USHR:	binary	Unsigned shift right (shift 0 into high bit).
:ROTR:, :ROTL:	binary	Rotate right, rotate left.

Operator Precedence

The following table lists the operators in decreasing order of precedence. The binary operators associate from left to right:

~ and - (unary)
*, /, %
+ and - (binary)
<<, >>, :USHR:, :ROTR:, and :ROTL:
=, <, >, <=, >=, ==, !=, :ULT, :UGT:, :ULE:, and :UGE:
&

^
|

Expressions are grouped with matching parentheses ().

Relocation Expression Operators

Operator	Type	Meaning
`lo(`*`value`*`)`	unary	Least significant 16 bits of *value.*
`hi(`*`value`*`)`	unary	Most significant 16 bits of symbol *value* plus most significant bit of `lo(`*`value`*`).`
`hi0(`*`value`*`)`	unary	Most significant 16 bits of *value*.
`pidlo(`*`value`*`)`	unary	Least significant 16 bits of PID symbol *value.*
`pidhi(`*`value`*`)`	unary	Most significant 16 bits of PID symbol *value* plus most significant bit of `pidlo(`*`value`*`).`
`pidhi0(`*`value`*`)`	unary	Most significant 16 bits of PID symbol *value*.
`sdaoff(`*`value`*`)`	unary	16 bits of SDA symbol *value.*
`sdaoff2(`*`value`*`)`	unary	16 bits of read-only SDA symbol *value*.
`sdaoff23(`*`value`*`)`	unary	23 bits of SDA symbol *value.*
`rosdaoff23(`*`value`*`)`	unary	23 bits of read-only SDA symbol *value*.
`zdaoff(`*`value`*`)`	unary	16 bits of ZDA symbol *value.*
`zdaoff23(`*`value`*`)`	unary	23 bits of ZDA symbol *value.*
`tdaoff(`*`value`*`)`	unary	Least significant 16 bits of TDA symbol *value.*

Since V850 immediate data is limited to 5-bit or 16-bit values, two instructions are required to load a 32-bit data value into a register. The `hi` and `lo` operators access the upper 16 and lower 16 bits of a 32-bit value, respectively, making it possible to load a register with a 32-bit address. The following code loads register `r22` with the value of the label `xyzzy`:

```
movhi hi(xyzzy), zero, r22
movea lo(xyzzy), r22, r22
```

C Type Information Operators

If you enable the **-asm3g** option (see "Support for C Type Information in Assembly" on page 218), you can import type information from C header files using the `.inspect` directive and use it in the following operators:

Expression	Meaning
`offsetof(`*type*`, `*ident*`)`	The offset of *ident* from the base type of *type* in bytes, where *type* is a structure tag or a `typedef` identifier.
`sizeof(`*type*`)`	The size of *type* in bytes, where *type* is a built-in type, a structure tag, or a `typedef` identifier.

When specifying a structure tag for a `sizeof()` or `offsetof()` directive, do not precede it with `struct` as you do when writing C code. For example, use `sizeof(foo)` instead of `sizeof(struct foo)`.

To specify a member `M` of a struct type `S`, use `sizeof(S->M)`. You must use `->` instead of `.`, because the assembler considers `.` a valid identifier character.

The second argument to `offsetof` must be a simple identifier. For example, the following code is not allowed:

```
offsetof(myStruct, myArrayMember[5])
```

Example 6.1. Using C Type Information Operators

This example shows you how to use type information in assembly directives. The following header file **foo.h** defines the type `myStruct`:

```
/* foo.h */

struct myStruct {
    char myChar;
    int myInt;
    struct {
        short myShort;
        int myArray[5];
    } nested;
};
```

The following assembly file **foo_asm.s** uses the type information for `myStruct` to define objects:

```
-- foo_asm.s

-- Get type information for myStruct from foo.h
.inspect "foo.h"
.data

-- Use type information from myStruct to write data to objects
myStruct_size:
            .word sizeof(myStruct)
nested_size:
            .word sizeof(myStruct->nested)
myArray_size:
            .word sizeof(myStruct->nested->myArray)
myInt_offset:
            .word offsetof(myStruct, myInt)
myShort_offset:
            .word offsetof(myStruct->nested, myShort)
-- The following two lines are syntax errors
--           .word sizeof(myStruct->myArray[0])
--           .word offsetof(myStruct, nested->myShort)
```

Expression Types

The primary expression types are:

Expression type	Meaning
`absolute`	A value of an identifier or expression that is computed by the assembler during assembly.
`integral constant`	An integral constant expression. Unlike an `absolute`, this may not contain the difference of labels.
`quoted string`	A C-style character string delimited by double quotation marks is used in conjunction with a number of assembler directives. All string escape sequences defined in the C language, such as `\n` for newline, are allowed. These sequences are described in "Character Escape Sequences" on page 367.
`relocatable`	A relocatable expression or identifier assigns a value relative to the beginning of a particular section. These values are determined at link time. All label identifiers are relocatable values.
`undefined`	If an identifier is unassigned, its value cannot be determined until link time. This is an undefined external.

Type Combinations

Constants can be combined with all operators. You can combine a constant with a relocatable or absolute type using any of the operators in the following table:

Operator	Type	Meaning
`+`	binary	addition (if one operand to the plus operator is a constant, the result is the type of the other operand)
`-`	binary	subtraction (if the second operand is constant, the result is the type of the first operand. If both operands are in the same section, then the result is a constant that is the difference of the addresses)

Examples

```
4             -- constant
4*(5+6)       -- constant
label         -- relocatable
```

Labels

A statement can begin with one or more *named labels*, which are identifiers followed by one or two colon characters. Labels defined with one colon are not referenced outside the source module. A second colon specifies that the label is made visible external to its source file, instead of local to that file.

To create temporary labels within a `.macro` directive, use the `.macrolocal` directive.

Current Location

The symbol `.` refers to the current location in the current section, which can be used as an alternative to creating a label when the current location must be referenced. For example:

```
sparse_table:
 .word 1,2

 -- move to address that is 16 bytes from sparse_table
 .space 16-(.-sparse_table)
 .word 3,4

 -- move to address that is 32 bytes from sparse_table
 .space 32-(.-sparse_table)
 .word 5,6
```

Reading the Compiler's Assembly Output

The assembly language file generated by the Green Hills compilers contains many useful comments, which are explained here. The format of these comments is subject to change without notice. In particular, additional comments might be generated by new releases of the compiler.

There are three sections of comments: a header comment at the beginning of the file, a function comment after each function, and a file comment at the end of the file.

Header Comment File

The first comment shows the command line to invoke the compiler driver, **ccv850**:

```
--Driver Command: ccv850 file1.c -S
```

The next few lines provide the name of the source file, the compile-time directory, the date and time the compiler was invoked, and the name and version of the host operating system in use:

```
--Source File:   file1.c
--Directory:     /tmp
--Compile Date:  Thu Dec 13 13:45:02 2001
--Host OS:       SunOS 5.8 Generic_108528-06
```

Then the version of the compiler and the exact date of the compiler is shown:

```
--Version:       target version RELEASE VERSION
--Release:       Version version
--Revision Date: Revision Date
--Release Date:  Release Date
```

Function Comment Section

The function comment section follows the function and lists parameters and local variables. It indicates where they are stored, either in a register, on the stack, or in static memory. However, if the program is compiled with optimization enabled, the generated code might drastically reduce variable lifetimes and make other alterations. In extreme cases, all uses of a variable in the register or memory location to which it is assigned can be optimized away.

```
;_i     d6      local
-- _j     d1      local
-- .L106  .L118   static

-- count  d10     param
-- point  r6      param
```

File Comment Section

This section lists imported and static variables declared at the file level and indicates where they are stored:

```
--y       y       static
--z       z       import
```

Source Line Comments

In addition to the three sections of comments described above, the Green Hills compilers can show the original source code in the assembly language output file in the form of comments. This is enabled with the `-passsource` option. Source lines for `#include` files are never shown. Some blank or inconsequential lines are deleted by the compiler. Otherwise, the source lines are shown exactly as they occur in the original source before preprocessor expansion.

Chapter 7

Assembler Directives

Contents

Assembler directives are instructions to the assembler to perform various functions.The following tables provide detailed descriptions of the assembler directives available to you when using the **ease850** assembler.

Alignment Directives

`.align` *`bound-expr`*

`.align` *`shift-expr`* `mod` *`bound-expr`*

Advances the location counter to the next address that is a multiple of the constant expression *bound-expr*, unless the location counter is already aligned to a multiple of *bound-expr* bytes. If you specify the constant expression *shift-expr*, the location counter is shifted that many additional bytes beyond the aligned address. As the location counter advances, skipped addresses are filled with zeros for the `.data` section and `nop` instructions for the `.text` section.

The linker ensures that any section containing an .align directive begins on a sufficiently aligned address. For example, suppose an assembly source file contains the following:

```
.data
.align 8
```

When the source file is linked, the starting address of the `.data` section is aligned on an 8-byte boundary. If the addressing boundary is defined in more than one source file being linked, the linker aligns the final section to the largest boundary found in the individual sections. For example, suppose two source files, **file_1.s** and **file_2.s**, contain the following:

- **file_1.s**:

  ```
  .data
  .align 8
  ```

- **file_2.s**:

  ```
  .data
  .align 32
  ```

When **file_1.s** and **file_2.s** are linked, the `.data` section is aligned on a 32-byte boundary.

Conditional Assembly Directives

The `.if` *`integral-const`* directive starts a block of assembly statements that are assembled only if *integral-const* is true. The `.endif` directive terminates the conditional block of assembly statements.

Within an `.if` / `.endif` block, the `.else` and `.elseif` directives provide alternative assembly statements that are assembled only if the `.if` *`integral-const`* directive evaluates to false.

```
.if x==2
.str "if directive is TRUE"
.elseif x-6
.str "elseif directive is TRUE"
.else
.str "both if and elseif are FALSE"
.endif
```

You can nest conditional blocks within conditional blocks.

By default, when the assembler creates a source listing, it always lists `.if` / `.endif` assembly statements, but lists the corresponding object code only if the block is assembled. That is, the assembler produces and lists object code only if *integral-const* evaluates to true.

`.else` Assembles the subsequent block of assembly statements if the previous `.if` directives evaluate to false.
`.elseif` *`integral-const`* Assembles the subsequent block of assembly statements if the previous `.if` directives evaluate to false and the `.elseif` *`integral-const`* evaluates to true. The assembler must be able to resolve *integral-const* to a constant without referring to information within the conditional block of code that follows the `.elseif` directive.
`.endif` Terminates the code block that began with the previous `.if` directive.
`.if` *`integral-const`* Starts a conditional block of assembly statements; the assembler generates object code for these statements only if *integral-const* evaluates to true. The assembler must be able to resolve *integral-const* to a constant without referring to information within the `.if` / `.endif` conditional block of code.

Data Initialization Directives

The following directives evaluate expressions and place successive values of the specified type in the assembly output.

Use `.offset` to advance the location counter to the address where you want to place data.

`.byte` *`const-expr,const-expr,...`* Stores the values of constant expressions in successive bytes. Each constant expression must be in the signed range -128 to 127 or in the unsigned range 0 to 255.
`.double` *`flt-expr,flt-expr,...`* Stores the values of floating-point expressions as successive 64-bit IEEE-754 floating-point values. Each floating-point expression must be within double-precision range.
`.hword` *`const-expr,const-expr,...`* Stores the values of constant expressions as successive 16-bit data. The constant expressions must be in the signed range -32768 to 32767 or the unsigned range 0 to 65535.
`.offset` *`const-expr`* Places subsequent data at the address offset *const-expr* from the beginning of the current section in the current compilation unit. Attempting to place subsequent data prior to the current offset will result in an error.
`.single` *`flt-expr,flt-expr,...`* Stores the values of floating-point expressions as successive 32-bit IEEE-754 floating-point values. Each floating-point expression must be within single-precision range.
`.space` *`const-expr`* Generates *const-expr* bytes of zero data.
`.str "`*`string`*`"` Evaluates a C-style string enclosed in double quotation marks (" ") and places the characters in successive bytes. The delimiting double quotation mark characters and the implicit terminating null are discarded.
`.strz "`*`string`*`"` Evaluates a C-style string enclosed in double quotation marks (" ") and places the characters in successive bytes. Unlike the `.str` directive, the terminating null is placed with the string. The delimiting double quotation mark characters are discarded.

`.word` *const-expr,const-expr,...*

Stores the values of constant expressions as successive 32-bit data. The expressions can be absolute expressions, relocatable expressions, or undefined external identifiers. The actual values of relocatable and undefined external identifiers are supplied at link time. The value of each expression must be in the signed range -2147483648 to 2147483647 or the unsigned range 0 to 4294967295.

File Inclusion Directives

`.include "file"`

Inserts the contents of *file* at the location of this directive. The assembler searches for *file* in the current working directory, then searches in directories specified by the **-I*inc_directory*** compiler driver option.

Macro Definition Directives

A `.macro`/`.endm` block defines the contents of a macro. When the name of the macro appears in the same assembly file, the assembler replaces the name of the macro with the macro's contents. This replacement is called *macro expansion*.

A macro might reference another macro. In this case, the assembler processes the enclosed macro when it expands the enclosing macro. Macros can call themselves recursively.

The `.macro` directive marks the beginning of a macro and is followed by the name of the macro and a list of parameters. Next, the body of the macro contains assembly statements that are included in an assembly file during macro expansion. To end the body of the macro, place a `.endm` directive as the first symbol on a line.

If you want the assembler to prematurely terminate macro expansion when a certain condition is true, enclose the `.exitm` directive within an `.if` / `.endif` block in the body of the macro. The `.exitm` directive prematurely terminates macro expansion.

To create a temporary label that changes each time a macro is called, use the `.macrolocal` directive. At every macro invocation, all identifiers throughout the macro body that are listed with the `.macrolocal` directive are appended with a suffix unique to that macro expansion. This allows local identifiers to be created and referenced within the macro without conflicting with the equivalent identifiers local to a subsequent macro expansion.

Example 7.1. Mangling Macros into Unique Labels

Each time the following macro `mymac` is invoked, `label` and `label2` are mangled into unique labels.

```
  .macro mymac
  .macrolocal label, label2
  blt label
  mov r0, 1
  bgt label2
  mov r0, 0
  b label2
label:
  mvn r0, 0
```

```
label2:
  .endm
```

If `mymac` was defined without the `.macrolocal` directive, `label` and `label2` would be defined more than once if the macro `mymac` was invoked more than once.

The following example defines two macros, `trythis` and `log_and`:

```
.macro trythis parm1
st.w r1,parm1*2 [r4]
.endm
.macro log_and parm1 parm2
st.w parm1, parm2 [r4]
.endm
```

To call a macro, put the name of the macro and its arguments in an assembly file. The macro must have been previously defined in the assembly file. You must specify a value for every parameter in the macro.

During macro expansion, the macro's parameters are replaced by the values specified after the macro name. The concatenation operator >< can be used to concatenate two parameters or a parameter with a symbol. The resulting assembly statement is not scanned for further parameter matches.

For example, to invoke the macros `trythis` and `log_and` (defined above) and place the resulting assembly statements in the `.text` section, enter the following:

```
.text
trythis 16
log_and r1, 0
log_and r1, 2
```

The assembler performs macro expansion and generates the following:

```
.text
st.w r1,32[r4]
st.w r1,0[r4]
st.w r1,2[r4]
```

`.endm`

Terminates the body of a macro whose definition began with a preceding `.macro` directive. The `.endm` directive must be the first symbol on its line and cannot have a label.

`.exitm` Causes the assembler to exit a macro without further expanding its body of assembly statements. Use within a `.if` / `.endif` block to specify a condition under which you do not want a macro expanded beyond that point.
`.macro` *`name`* `[`*`param1`*`,` *`param2`*`,...]` Begins the definition of a macro, where *name* is a string representing the name of the macro and *param1* is the first parameter that the macro accepts. Parameters must be separated by a comma or whitespace.

Repeat Block Directives

The following directives specify a block of assembly statements that repeat *integral-const* times.

```
.rept integral-const
...
.endr
```

Repeat blocks can occur within repeat blocks. In this case, the enclosed repeat block is expanded once for each expansion of the enclosing block. The repeat count of an inner block is evaluated each time the block is expanded.

You can define a repeat block within a macro definition as long as the repeat block is completely enclosed within the macro.

`.endr` Terminates a repeat block that began with a preceding `.rept` *`const-expr`* directive. The `.endr` directive must be the first symbol on its line and cannot have a label.
`.rept` *`integral-const`* Begins a block of assembly statements that repeats *integral-const* times, where *integral-const* resolves to a constant number.

Section Control Directives

These directives direct assembly output into the specified section.

`.data` Changes the active section to the `.data` section. Data that follows this directive is placed in the `.data` section.
`.note "`*`string`*`" [,"a|b|w|x"]` Changes the active section to an ELF note section. The new section is called *string*. The attribute letters `a`, `b`, `w`, and `x` function as they do with the `.section` directive.
`.org` *`const-expr`* Creates an absolutely located data section at address *const-expr*. The name of the section is the section's address, with a prefix of `org`. The MULTI Debugger cannot debug code placed at an address with a `.org` directive. Therefore, we recommend that you use the `.section` directive to declare code and data in a section instead, and then define the address of the section in a linker directives file.
`.previous` Changes the active section back to the one in use before the most recent section control directive.

`.section "`*`string`*`"[,"a|b|w|x"][ > `*`const-expr`*`]`

Directs assembly output into the section named *string*, which is defined in terms of the optional attributes as listed below.

- `a`: the section should have memory allocated for it; that is, it should not be used solely for debugging or for symbolic information.
- `b`: the section will have BSS semantics. Although normal data directives such as `.word` and `.byte` are allowed in a `.bss` section, all of the values specified in those directives are discarded by the assembler. In the ELF output file, the assembler records only the size of the section, and its contents are omitted. When the section is downloaded to the target, space is allocated for the section, but no data is downloaded to this section. Instead, the startup code is responsible for initializing all bytes in the section to zero.
- `w` — the section is writable.
- `x` — the section contains executable code.

If none of these letters are specified, then no attributes are set. This is appropriate only for sections containing debugging or other information not intended to be part of the final linked file. Sections that are intended to be part of the final linked output should have at least the `a` attribute. After the attributes are set they cannot be specified again.

The standard sections have predefined attributes as follows:

- `.text` (Program code) — `"ax"`
- `.data` (Initialized data) — `"aw"`
- `.rodata` (Read-only initialized data) — `"a"`
- `.bss` (Uninitialized data) — `"awb"`

If the optional `> `*`const-expr`* form is used, the section is marked as an absolute section located at the address indicated by *const-expr*. This is similar to a section created with the `.org` directive, except that this format allows you to specify the section's name and flags. If the `.section` directive appears more than once in a single assembly language file with the same section name, only the first `.section` directive can use this form. Unlike the `.org` directive, absolute executable sections created in this way can be debugged by the MULTI Debugger.

This example creates a section called `.mytext` with allocation and execute attributes:

```
.section ".mytext","ax"
```

`.text`

Changes the active section to the `.text` section. Assembly code and data that follows this directive is placed in the `.text` section.

Source Listing Directives

The following directives control the format and content of the source listing produced by the assembler. They do not affect how the assembler generates object code.

Use the **-list** compiler driver option to produce a source listing. For example, to have the assembler produce a source listing **file.lst** when it assembles **file.s**, enter:

```
ccv850 file.s -list
```

`.eject` Puts a form feed (`^L`) into the source listing.
`.gen` Causes macros following this directive to be included in the source listing.
`.list` Cancels the **.nolist** directive (re-enables source listing).
`.nogen` Prevents macros in the sections following this directive from being included in the source listing.
`.nolist` Causes the assembler to exclude information from the source listing until a corresponding `.list` directive is encountered.
`.nowarning` Disables warnings.

`.sbttl "`*`string`*`"`
`.subtitle "`*`string`*`"`
Inserts *string* as the subtitle at the top of each page in the source listing. The double quotation marks are discarded.

`.title "`*`string`*`"` Inserts *string* as the title at the top of each page in the source listing. The double quotation marks are discarded.
`.warning ["`*`string`*`"]` Enables warnings. If "*string*" is specified, prints the warning "*string*" to standard error output.

Symbol Definition Directives

The following directives define the type and value of an identifier.

`.comm` *`ident,const-expr[,const-expr]`*

Assigns the specified identifier *ident* to a common area of *const-expr* bytes in length, and causes *ident* to be visible externally. If *ident* is not defined by another relocatable object file, the linker assigns space for the identifier in the `.bss` section. The optional *const-expr* specifies the variable alignment in bytes.

`.dsect`

Begins a block for defining constants. Within a `.dsect` / `.end` block, only labels and `.space` directives are permitted. The symbolic names used for labels are defined to have values equal to the number of bytes from the beginning of the `.dsect` block. No space is actually allocated; only the symbolic names in the label fields are defined.

`.end`

Terminates a block of constants that began with a preceding `.dsect` directive. Defining this directive outside of a `.dsect` directive may result in unexpected behavior.

`.globl` *`ident`*

Causes the identifier *ident* to be visible externally. If the identifier is defined in the current module, this directive allows the linker to resolve references by other modules. If the identifier is not defined in the current module, the assembler resolves it externally.

`.inspect "`*`file.h`*`"`

Parses the C header file *file.h* to get type information. You can use this information in the `.struct` directive, and in the `offsetof()` and `sizeof()` operators (see "C Type Information Operators" on page 373).

This directive is available only when the **-asm3g** option is enabled (see "Support for C Type Information in Assembly" on page 218).

`.lcomm` *`ident,const-expr[,const-expr]`*

Assigns the specified identifier *ident* to an area in `.bss` of *const-expr* bytes in length. The optional *const-expr* specifies the variable alignment in bytes. The `.lcomm` directive is similar to the `.comm` directive, except that *ident* is not exported.

`.lsbss` *`ident,const-expr[,const-expr]`*

Assigns the specified identifier *ident* to an area in `.sbss` of *const-expr* bytes in length. The optional *const-expr* specifies the variable alignment in bytes. The `.lsbss` directive is similar to `.sbss`, except that *ident* is not exported.

`.sbss` *ident*`,`*const-expr*`[,`*const-expr*`]` Assigns the specified identifier *ident* to a common area of *const-expr* bytes in length and causes *ident* to be visible externally. If *ident* is not defined by another relocatable object file, the linker assigns space for the identifier in the `.sbss` section. The optional *const-expr* specifies the variable alignment in bytes.
`.set` *ident*`,`*const-expr* Defines the constant value *const-expr* to the symbol *ident*. You can use multiple `.set` directives to change the value of the same symbol.
`.weak` *ident* Makes the symbol weak. If the *ident* cannot be located in any module at link time, the linker sets the symbol's value to zero.
`.zbss` *ident*`,`*const-expr*`[,`*const-expr*`]` Assigns the specified identifier *ident* to a common area of *const-expr* bytes in length and causes *ident* to be visible externally. If *ident* is not defined by another relocatable object file, the linker assigns space for the identifier in the `.zbss` section. The optional *const-expr* specifies the variable alignment in bytes.

Symbolic Debugging Directives

These directives are used for symbolic debugging.

`.file "`*`string`*`"`

Stores source filename *string* in the object file symbol table. The *string* filename must be delimited by double quotation marks. Use this directive no more than once per file.

`.fsize` *`func`*`,` *`const`*

Specifies that the frame size for the current function is *const* bytes in size. This information may be useful in certain situations when debugging call stacks involving assembly functions.

You must place this directive within the scope of the current function, before you define the next function. The optional *func* parameter can be used to specify the current function's name.

Values specified with `.fsize` may also be used by **gstack** when no `.maxstack` directive is present. We recommend using `.maxstack`, especially when the function allocates additional stack after its frame has been set up. Use of `.fsize` for this purpose may be deprecated in the future.

`.maxstack [`*`func`*`,] `*`const`*

Specifies that the current function uses no more than *const* bytes of stack. This information allows **gstack** to compute accurate results for paths that involve calls to assembly functions. You must place this directive within the scope of the current function, before you define the next function. The optional *func* parameter specifies the current function's name.

`.nodebug`

Turns off debugging.

`.scall` *`func`*`,` *`dest`*

Specifies that *func* makes a static call to *dest*. This information helps **gstack** compute the maximum stack size that the program might use. It also helps the Debugger generate a call graph.

To indicate that *func* makes no static calls, use `0` or `__leaf__` for *dest*.

You can use multiple `.scall` directives to specify multiple calls for a single function, however, if you use `0` or `__leaf__`, you must use just one `.scall` directive for *func*.

`.size` *ident*`, ` *const-expr*

Specifies the size, *const-expr*, of the function or data object *ident*. To successfully debug an assembly language source file, it should either be assembled with the **-g** driver option, (which outputs full debug symbols) or else a `.size` directive should be used to specify the size of each function.

The `.size` directive should appear at the end of the function, just after the last assembly language statement in the function. It should be of the form:

`.size` *ident*`,.-`*ident*

where *ident* is the name of the function. The expression `.-`*ident* will be equal to the size of the function in bytes, allowing the creation of correct debugging information.

Use this directive when defining each symbol for function deletion to work properly.

`.type` *ident*`, [ @function | @object ]`

Specifies whether the symbol *ident* is a function or a data object. The `.type` directive is usually used in conjunction with the `.size` directive to define a function or data object.

Use this directive when defining each symbol to get the best debugging information, and for function deletion to work properly.

Miscellaneous Directives

`.ident "`*`string`*`"`

Adds *string* to the `.comment` section within the object file. The `.comment` section is not allocatable; it does not occupy memory on the target.

`.need` *`ident`*

Defines a relationship between the function *ident* and the current function. The linker will not delete the function *ident* as long as the current function is used, even if *ident* is not used. Suppose a source file contains the following:

```
myfunc:
    .need otherfunc
    -- Insert assembly function here
```

The linker will not delete the function `otherfunc`, even if it is unused, because it is needed by the function `myfunc`.

RH850 Directives

`.rh850_flags` *flags*

Instructs the assembler on which RH850 ABI capabilities are supported in the current file. Please refer to the RH850 ABI from Renesas Electronics for details. Depending on which flags are specified, options may be set in the ELF E_FLAGS field of the current object file's ELF header, or in the ABI features section of the `.renesas_info` ELF section of the current object file. This option is only meaningful when targeting RH850 and later processors. Flags are specified as a comma-separated list, with possible flags being:

CACHE	Sets the ABI-conforming flag corresponding to cache usage.
DATA_ALIGN8	Sets the ABI-conforming flag which indicated an 8-byte aligned stack and 8-byte aligned 64-bit types (e.g.: double, long long).
DATA_ALIGN4	Sets the ABI-conforming flag which indicated a 4-byte aligned stack and 4-byte aligned 64-bit types (e.g.: double, long long).
DOUBLE	Sets the ABI-conforming flag corresponding to double-precision floating point usage (see “Floating-Point Mode” on page 122).
DOUBLE32	Sets the ABI-conforming flag corresponding to double-precision floating point usage (see “Treat Doubles as Singles” on page 122). Indicates that double and long double types are four bytes in size.
DOUBLE64	Sets the ABI-conforming flag corresponding to double-precision floating point usage (see “Treat Doubles as Singles” on page 122). Indicates that double and long double types are eight bytes in size.
EP_FIX	Sets the ABI-conforming flag that indicates that the `EP` register was reserved for the TDA or not used for allocation (see “Special Data Area Optimizations” on page 87 and “Allocate Pointers to the EP Register” on page 134).
FLOAT	Sets the ABI-conforming flag corresponding to single-precision floating point usage (see “Floating-Point Mode” on page 122).
FPU20	Sets the ABI-conforming flag that indicates that the floating-point coprocessor is a legacy version 2.x FPU (see “Special Data Area Optimizations” on page 87).
FPU30	Sets the ABI-conforming flag that indicates that the floating-point coprocessor is a standard version 3.x FPU (see “Special Data Area Optimizations” on page 87).
GP_FIX	Sets the ABI-conforming flag that indicates that the `GP` register was reserved for the SDA (see “Special Data Area Optimizations” on page 87).
GHS_MISALIGN	Sets the GHS-specific flag that indicates that loads and stores may be misaligned (see “Misaligned Memory Access” on page 200).

MMU	Sets the ABI-conforming flag corresponding to usage of the memory management unit.
REG2_RESERVE	Sets the ABI-conforming flag that indicates that the `R2` register was reserved (see “Register r2” on page 119).
REGMODE22	Sets the ABI-conforming flag corresponding to usage of the 22 register mode (see “Register Mode” on page 120).
REGMODE32	Sets the ABI-conforming flag corresponding to usage of the 32 register mode (see “Register Mode” on page 120).
SIMD	Sets the ABI-conforming flag corresponding to SIMD unit usage (see “Enable SIMD instructions (RH850 and later)” on page 136).
TP_FIX	Sets the ABI-conforming flag that indicates that the `TP` register was reserved for the ROSDA (see “Special Data Area Optimizations” on page 87).

Note

Many of the flags referenced above are set automatically either by the compiler or assembler based on command line arguments.

Chapter 8

V850 Assembler Reference

Contents

This chapter provides detailed information about the V850 addressing modes and instruction formats for writing and maintaining Green Hills V850 assembler code, with a complete alphabetical listing of all V850 instructions. For additional information, refer to the *V850 Design Specification* and *V850 Instruction Set*, which cover related topics such as instruction set and exception handling.

Reserved Symbols

The following identifiers are part of the fundamental assembly language and, as such, are reserved symbols in addition to those mentioned in "Relocation Expression Operators" on page 372. These symbols and their meanings are below.

Identifier	Meaning
`r0` — `r31`	Integer registers
`zero`	Zero register (`r0`)
`hp`	Handler stack pointer (`r2`)
`sp`	Stack pointer (`r3`)
`gp`	Global pointer (`r4`)
`tp`	Text pointer (`r5`)
`ep`	Element pointer (`r30`)
`lp`	Link pointer (`r31`)

Register Sets

There are two types of registers explained in detail below: general and system.

General Registers

There are 32 general purpose registers, each 32 bits wide. Although only `r0` has a special usage at the hardware level, a convention was established where the following V850 general registers have reserved usage at the software level:

Register names	Usage
`r0` (`zero`)	Always contains the value zero. Any write to `r0` is ignored.
`r1`	Reserved for the compiler or assembler.

Register names	Usage
r2 (hp)	Temporary register.
r3 (sp)	Stack pointer.
r4 (gp)	Global pointer.
r5	Read-only small data pointer.
r6 — r9	Parameter registers.
r10	Return register. (32-bit data)
r10, r11	Return registers. (64-bit data)
r10 — r19	Temporary registers.
r20 — r29	Permanent registers.
r30 (ep)	Element pointer
r31 (lp)	Procedure return pointer.

The `jarl routine, lp` instruction overwrites the contents of register `lp` (`r31`) with the return address generated by the call. However, the contents of `lp` may also be overwritten by software if required.

If an instruction uses `r1` or writes to `r0`, the V850 assembler **as850** outputs a warning message by default. You can suppress this warning message with the **-w** option.

By default, the V850 driver **ccv850** disables warnings. If you want to enable assembler warnings, pass **--assembler_warnings** to the driver.

The V850 Tool Chain supports three software modes of register usage: 32 registers (default), 26 registers (**ccv850** driver option: **-registermode=26**), and 22 registers (**ccv850** driver option: **-registermode=22**). For more information, see “Register Usage” on page 32.

Vector Registers

There are 32 vector registers on RH850 processors equipped with a SIMD coprocessor. Each vector register is 64 bits wide. These registers, represented as `vr0` — `vr31` in the assembler, are all treated as temporary registers.

System Registers

There are 32 system registers, all 32-bits wide:

Register ID	Name	Usage
0	EIPC	Exception/interrupt PC
1	EIPSW	Exception/interrupt PSW
2	FEPC	Fatal error PC
3	FEPSW	Fatal error PSW
4	ECR	Exception cause register (read-only)
5	PSW	Program status word

In addition to the V850 system registers, the following system registers are valid on the V850E:

Register ID	Name	Usage
16	CTPC	callt PC
17	CTPSW	callt PSW
18	DBPC	Exception / debug trap PC
19	DBPSW	Exception / debug trap PSW
20	CTBP	callt base pointer

In addition to the V850E system registers, the following system registers are valid on the V850E2:

Register ID	Name	Usage
21	DIR	Debug interface register

The V850E2R and V850E2V3 processors introduced the concept of register banks, selected via the new BSEL register. Although the ability to access the system registers now depends on the current value of the BSEL register, the same symbolic names supported with the V850E and their associated register ID values are made available for use with instructions that require a system register operand, as are the additional symbolic names below. However, the assembler assumes the user will have set the BSEL register to the appropriate register bank first.

Register ID	Name	Usage
13	EIIC	EI level exception cause

Register ID	Name	Usage
14	FEIC	FE level exception cause
15	DBIC	DB level exception cause
28	EIWR	EI level exception working register
29	FEWR	FE level exception working register
30	DBWR	DB level exception working register
31	BSEL	Register bank selection
6	FPSR	Floating-point configuration/status
7	FPEPC	Floating-point exception PC
8	FPST	Floating-point operation status
9	FPCC	Floating-point comparison result
10	FPCFG	Floating-point operation configuration

For a description of the purpose of each register, please refer to the appropriate Design Specification documentation from Renesas Electronics.

The RH850 and newer processors replaced the old BSEL banked approach with register groups. Rather than manipulating a register such as BSEL to select a register group, the register group is now specified directly in the load or store instruction used to access the register. The following table lists the symbolic names for these registers supported by the RH850 assembler. When using these symbolic names, users should not specify the group number in the instruction, as the assembler will set it appropriately.

Group Number	Register Number	Name	Usage
0	0	EIPC	Exception/Interrupt PC
0	1	EIPSW	Exception/Interrupt PSW
0	2	FEPC	Fatal Error PC
0	3	FEPSW	Fatal Error PSW
0	5	PSW	Program status word
0	6	FPSR	Floating-point configuration/status register
0	7	FPEPC	Floating-point exception PC
0	8	FPST	Floating-point status

Group Number	Register Number	Name	Usage
0	9	FPCC	Floating-point comparison result
0	10	FPCFG	Floating-point configuration
0	11	FPEC	Floating-point exception control
0	12	SESR	SIMD operation settings and status
0	13	EIIC	EI level exception cause
0	14	FEIC	FE level exception cause
0	16	CTPC	callt PC save register
0	17	CTPSW	callt PSW save register
0	20	CTBP	callt base pointer
0	28	EIWR	EI level exception working register
0	29	FEWR	FE level exception working register
0	30	DBWR	DB level exception working register
0	31	BSEL	Bank select. Reserved for backwards compatibility on the RH850 only. This register is ignored by the hardware
1	0	MCFG0	CPU configuration
1	1	MCFG1	CPU configuration
1	2	RBASE	Reset vector base address
1	3	EBASE	Exception handler vector address
1	4	INTBP	Interrupt handler address table base address
1	5	MTCL	CPU control
1	6	PID	Processor ID
1	7	FPIPR	FPI exception interrupt priority setting
1	10	TCSEL	Hardware thread selection
1	11	SCCFG	SYSCALL operation setting
1	12	SCBP	SYSCALL base pointer
1	13	HVCCFG	HVCALL configuration
1	14	HVCBP	HVCALL base pointer
1	15	VCSEL	Virtual machine context selection
1	16	VMPRT0	Virtual machine hardware thread assignment

Group Number	Register Number	Name	Usage
1	17	VMPRT1	Virtual machine hardware thread assignment
1	23	VMSCCTL	Virtual machine hardware schedule
1	24	VMSCTBL0	Virtual machine schedule table setting
1	25	VMSCTBL1	Virtual machine schedule table setting
1	26	VMSCTBL2	Virtual machine schedule table setting
1	27	VMSCTBL3	Virtual machine schedule table setting
2	0	HTCFG0	Thread configuration
2	5	HTCTL	Hardware thread control
2	6	MEA	Memory error address
2	7	ASID	Address space ID
2	8	MEI	Memory error information
2	10	ISPR	Priority of interrupt being served
2	11	PMR	Interrupt priority masking
2	12	ICSR	Interrupt control status
2	13	INTCFG	Interrupt function setting
2	16	TLBSCH	TLB disable setting
2	23	HTSCCTL	Hardware thread schedule setting
2	24	HTSCTBL0	Hardware thread schedule table
2	25	HTSCTBL1	Hardware thread schedule table
2	26	HTSCTBL2	Hardware thread schedule table
2	27	HTSCTBL3	Hardware thread schedule table
2	28	HTSCTBL4	Hardware thread schedule table
2	29	HTSCTBL5	Hardware thread schedule table
2	30	HTSCTBL6	Hardware thread schedule table
2	31	HTSCTBL7	Hardware thread schedule table
4	0	TLBIDX	TLB operation target index
4	4	TELO0	TLB entry Lo-side access control
4	5	TELO1	TLB entry Lo-side access control
4	6	TEHI0	TLB entry Hi-side access control

Group Number	Register Number	Name	Usage
4	7	TEHI1	TLB entry Hi-side access control
4	10	TLBCFG	TLB configuration
4	12	BWERRL	Bus write access error Lo
4	13	BWERRH	Bus write access error Hi
4	14	BRERRL	Bus read access error Lo
4	15	BRERRH	Bus read access error Hi
4	16	ICTAGL	Instruction cache tag Lo access
4	17	ICTAGH	Instruction cache tag Hi access
4	18	ICDATL	Instruction cache data Lo access
4	19	ICDATH	Instruction cache data Hi access
4	20	DCTAGL	Data cache tag Lo access
4	21	DCTAGH	Data cache tag Hi access
4	22	DCDATL	Data cache data Lo access
4	23	DCDATH	Data cache data Hi access
4	24	ICCTRL	Instruction cache control
4	25	DCCTRL	Data cache control
4	26	ICCFG	Instruction cache configuration
4	27	DCCFG	Data cache configuration
4	28	ICERR	Instruction cache error
4	29	DCERR	Data cache error
5	0	MPM	Memory protection operation mode setting
5	1	MPRC	MPU region control
5	4	MPBRGN	MPU base region number
5	5	MPTRGN	MPU end region number
5	8	MCA	Memory protection setting check address
5	9	MCS	Memory protection setting check size
5	10	MCC	Memory protection setting check command
5	11	MCR	Memory protection setting check result
5	17	MPVADR	MPU violation address

Group Number	Register Number	Name	Usage
5	20	MPPRT0	Protection area assignment
5	21	MPPRT1	Protection area assignment
5	22	MPPRT2	Protection area assignment
6	0	MPLA0	Protection area minimum address
6	1	MPUA0	Protection area maximum address
6	2	MPAT0	Protection area attribute
6	4	MPLA1	Protection area minimum address
6	5	MPUA1	Protection area maximum address
6	6	MPAT1	Protection area attribute
6	8	MPLA2	Protection area minimum address
6	9	MPUA2	Protection area maximum address
6	10	MPAT2	Protection area attribute
6	12	MPLA3	Protection area minimum address
6	13	MPUA3	Protection area maximum address
6	14	MPAT3	Protection area attribute
6	16	MPLA4	Protection area minimum address
6	17	MPUA4	Protection area maximum address
6	18	MPAT4	Protection area attribute
6	20	MPLA5	Protection area minimum address
6	21	MPUA5	Protection area maximum address
6	22	MPAT5	Protection area attribute
6	24	MPLA6	Protection area minimum address
6	25	MPUA6	Protection area maximum address
6	26	MPAT6	Protection area attribute
6	28	MPLA7	Protection area minimum address
6	29	MPUA7	Protection area maximum address
6	30	MPAT7	Protection area attribute
7	0	MPLA8	Protection area minimum address
7	1	MPUA8	Protection area maximum address

Group Number	Register Number	Name	Usage
7	2	MPAT8	Protection area attribute
7	4	MPLA9	Protection area minimum address
7	5	MPUA9	Protection area maximum address
7	6	MPAT9	Protection area attribute
7	8	MPLA10	Protection area minimum address
7	9	MPUA10	Protection area maximum address
7	10	MPAT10	Protection area attribute
7	12	MPLA11	Protection area minimum address
7	13	MPUA11	Protection area maximum address
7	14	MPAT11	Protection area attribute
7	16	MPLA12	Protection area minimum address
7	17	MPUA12	Protection area maximum address
7	18	MPAT12	Protection area attribute
7	20	MPLA13	Protection area minimum address
7	21	MPUA13	Protection area maximum address
7	22	MPAT13	Protection area attribute
7	24	MPLA14	Protection area minimum address
7	25	MPUA14	Protection area maximum address
7	26	MPAT14	Protection area attribute
7	28	MPLA15	Protection area minimum address
7	29	MPUA15	Protection area maximum address
7	30	MPAT15	Protection area attribute

For a description of the purpose of each register, please refer to the appropriate documentation from Renesas Electronics.

Addressing Modes

Introduction

All the addressing modes offered by the V850 processor are supported and are summarized below:

Addressing mode	Notation	Example
No arguments	(*no arguments*)	`di` `halt`
Immediate	*imm5*	`trap 5`
Register indirect jump	[*reg*]	`jmp [lp]`
Two registers	*reg1*, *reg2*	`add r22, r23` `cmp r22, r23`
Register with 5-bit immediate	*simm5*, *reg*	`add 5, r22`
	imm5, *reg*	`shl 31, r22`
Register with 16-bit immediate	*simm16*, *reg1*, *reg2*	`addi -74, r22, r23`
	imm16, *reg1*, *reg2*	`xori 0x8000, r7, r8`
	simm16, *reg1*	`mov -30000, r7`
System register	*reg*, *regID*	`ldsr r22, 5`
	regID, *reg*	`stsr 5, r22`
Register indirect with 16-bit displacement	*simm16*[*reg1*], *reg2*	`ld.w 0x10[r22], r23`
	reg2, *simm16*[*reg1*]	`st.w r23, -0x56[r3]`
Register indirect with 7-bit displacement	*imm7*[ep], *reg2*	`sld.b 0x12[ep], r23`
	reg2, *imm7*[ep]	`sst.b r23, 24[ep]`
Register indirect with 8-bit displacement	*imm8*[ep], *reg2*	`sld.w 4[ep], r23`
	reg2, *imm8*[ep]	`sst.w r23, 0x10[ep]`
9-bit branch displacement	*disp9*	`bne 30`
22-bit branch displacement	*disp22*	`jr 0x12346`
	disp22, *reg2*	`jarl 0x12346, lp`
Condition code	*cccc*, *reg*	`setf z, r1`
Bit manipulation	*bit#3*, *simm16*[*reg*]	`tst1 3, 0x24[r5]`

In addition to the V850 addressing modes, the V850E supports the following addressing modes:

Addressing mode	Notation	Example
One register	*reg*	`zxh r24`
Immediate	*imm6*	`callt 3`
Three registers	*reg1*, *reg2*, *reg3*	`div r22, r23, r14` `mulu r22, r23, r24`
Register indirect with 4-bit displacement	*imm4*[`ep`], *reg*	`sld.bu 1[ep], r6`
Register indirect with 5-bit displacement	*imm5*[`ep`], *reg*	`sld.hu 2[ep], r6`
Register with 32-bit immediate	*imm32*, *reg*	`mov 0x12345678, r7`
Two registers with 9-bit immediate	*simm9*, *reg2*, *reg3*	`mul -65, r22, r23`
	imm9, *reg2*, *reg3*	`mulu 167, r22, r23`
Register list	*list12*, *imm5*	`prepare {r20, r22-r27}, 16`
	imm5, *list12*	`dispose 12, {r20, r22-r27},`
	list12, *imm5*, *sp / imm16 / imm32*	`prepare {r21, r23-r25}, 16, 0x1000` `prepare {r21}, 4, sp`
	imm5, *list12*, [*reg*]	`dispose 4, {r21, r23-r25}, [lp]`
Condition code	*cccc*, *reg1*, *reg2*, *reg3*	`cmov ne, r18, r6, r22`
	cccc, *simm5*, *reg2*, *reg3*	`cmov e, -10, r6, r22`

In addition to the V850 and V850E addressing modes, the V850E2 supports the following addressing modes:

Addressing mode	Notation	Example
32-bit branch displacement	*disp32*	`jr32 0x1234678`
	disp32, *reg1*	`jarl32 0x1234678, lp`
	disp32[*reg1*]	`jmp 0x1234678[r20]`
Four registers	*reg1*, *reg2*, *reg3*, *reg4*	`mac r6, r7, r8, r10`

In addition to the V850, V850E and V850E2 addressing modes, the V850E2R supports the following addressing modes:

Addressing mode	Notation	Example
Floating-point condition code	*fcc*, *reg1*, *reg2*, *reg3*	`cmovf.s fcc0, r7, r8, r9`
	fcond, *reg1*, *reg2*	`cmpf.d ole, r8, r10`
	fcond, *reg1*, *reg2*, *fcc*	`cmpf.d ole, r8, r10, fcc1`
	fcc	`trfsr fcc2`
Register indirect with two registers	[*reg1*], *reg2*, *reg3*	`caxi [r6], r7, r12`

In addition to the V850, V850E, V850E2 and V850E2R addressing modes, the V850E2V3 supports the following addressing modes:

Addressing mode	Notation	Example
Register indirect with 23-bit displacement	*simm23*[*reg1*], *reg2*	`ld.w 0x10000[r22], r23`
	reg2, *simm23*[*reg1*]	`st.w r23, -0x56000[r3]`

In addition to the V850, V850E, V850E2, V850E2R and V850E2V3 addressing modes, the RH850 supports the following addressing modes:

Addressing mode	Notation	Example
16-bit unsigned branch-backwards displacement	*reg1*, *disp16*	`loop r12, .-0x1234`
17-bit branch displacement	*disp17*	`bne .+0x1234`
Register range	*reg1-reg2*	`popsp r4-r14`
Position and width	*reg1, pos, width, reg2*	`bins r7, 3, 5, r14`
Register indirect with no displacement	*[reg1], reg2*	`ldl.w [r10], r11`
	reg1, [reg2]	`stc.w r11, [r10]`
One general register and one vector register	*reg1, vreg1*	`mov.dw r10, vr12`
	vreg1, reg1	`mov.dw vr12, r10`
One general register and two vector registers	*reg1, vreg1, vreg2*	`vmaxge.h r8, vr10, vr12`
One general register and three vector registers	*reg1, vreg1, vreg2, vreg3*	`vconcat.b r8, vr10, vr12, vr14`

One general register, one vector register and a 1-bit immediate	*imm1, reg1, vreg1*	`mov.w 1, r8, vr12`
	imm1, vreg1, reg1	`mov.w 1, vr12, r8`
Two vector registers	*vreg1, vreg2*	`cnvq15q30 vr10, vr12`
Two vector registers and a 1-bit immediate	*imm1, vreg1, vreg2*	`dup.w 1, vr10, vr12`
Two vector registers and a 2-bit immediate	*imm2, vreg1, vreg2*	`dup.h 2, vr10, vr12`
Two vector registers and a 4-bit immediate	*imm4, vreg1, vreg2*	`vsar.dh 2, vr10, vr12`
Two vector registers and a 5-bit immediate	*imm5, vreg1, vreg2*	`vsar.w 2, vr10, vr12`
Two vector registers and a 6-bit immediate	*imm6, vreg1, vreg2*	`vsar.dw 2, vr10, vr12`
Three vector registers	*vreg1, vreg2, vreg3*	`pki16i32 vr10, vr12, vr14`
Four vector registers	*vreg1, vreg2, vreg3, vreg4*	`vcalc.h vr7, vr10, vr12, vr14`
Register indirect with post increment	*[reg1]+, vreg1*	`vld.b [r10]+, vr12`
	vreg1, [reg1]+	`vst.b vr12, [r10]+`
Register indirect with specified post increment	*[reg1]+, reg2, vreg1*	`vld.b [r10]+, r11, vr12`
	vreg1, [reg1]+, reg2	`vst.b vr12, [r10]+, r11`
Register indirect with post decrement	*[reg1]-, vreg1*	`vld.b [r10]-, vr12`
	vreg1, [reg1]-	`vst.b vr12, [r10]-`
Register indirect with modulo addressing	*[reg1]%, reg2, vreg1*	`vld.b [r10]%, r11, vr12`
	vreg1, [reg1]+, reg2	`vst.b vr12, [r10]+, r11`
Register indirect with 8-bit displacement	*simm8[reg1], vreg1*	`vld.dw 8[r10], vr12`
	vreg1, simm8[reg1]	`vst.dw vr12, 8[r10]`
Register indirect with pre-bit-reversal	*vreg1, [reg1]!, reg2*	`vst.dw vr12, [r10]!, r11`
Register indirect with no immediate	*vreg1, [reg1]*	`vst.dw vr12, [r10]`
Register indirect with cache operation	*cacheop, [reg1]*	`cache 0, [r10]`
Register indirect with prefetch operation	*prefop, [reg1]*	`pref 0, [r10]`

Key:

Term	Meaning
reg	The only general register in the instruction.
reg1	The first general register.
reg2	The second general register.
reg3	The third general register.
reg4	The fourth general register.
vreg1	The first vector register.
vreg2	The second vector register.
vreg3	The third vector register.
vreg4	The fourth vector register.
sp	The stack pointer register.
regID	The system register ID; it may take the values 0 through 31.
imm1	An unsigned 1-bit value (0 or 1).
imm2	An unsigned 2-bit value (0 to 3).
imm4	An unsigned 4-bit value (0 to 15).
imm5	An unsigned 5-bit value (0 to 31).
simm5	A signed 5-bit value (-16 to 15).
imm6	An unsigned 6-bit value (0 to 63).
imm7	An unsigned 7-bit value (0 to 127).
imm8	An unsigned 8-bit value (0 to 255).
simm8	A signed 8-bit value (-128 to 127).
imm9	An unsigned 9-bit value (0 to 511).
simm9	A signed 9-bit value (-256 to 255).
list12	A comma-delimited list of registers enclosed in braces.
imm16	An unsigned 16-bit value (0 to 65535).
simm16	A signed 16-bit value (-32768 to 32767).
imm32	An unsigned 32-bit value (0 to 4294967295) or a signed 32-bit value (-2147483648 to 2147483647).
disp9	A signed 9-bit displacement value.
disp17	A signed 17-bit displacement value.

Term	Meaning
disp22	A signed 22-bit displacement value.
disp32	A 32-bit displacement value.
pos	A 9-bit number (0 to 31)
width	A 9-bit number (1 to 32)
cacheop	A number. See your processor or cache documentation for possible values.
prefop	A number. See your processor or cache documentation for possible values.
bit#3	A 3 bit unsigned number (0 to 7) encoding the bit position within they targeted byte.
cccc	A 4 bit unsigned number (0 to 15) encoding the condition code or one of the condition code mnemonics.
fcc	A 3 bit unsigned number (0 to 7) encoding the condition register ID or one of the condition register mnemonics.
fcond	A 4 bit unsigned number (0 to 15) encoding the condition code or one of the condition code mnemonics.

Every V850 instruction is either 2 bytes (16 bits) or 4 bytes (32 bits) long, including those instructions containing immediate data. The V850E family also has 6 byte (48 bits) and 8 byte (64 bits) instructions such as `mov` 32 bit immediate and `prepare`.

In line with the RISC philosophy, there are relatively few instructions and only one or two addressing modes apply to any given one. Also, a given addressing mode applies to the instruction as a whole (in contrast to some architectures that allow different addressing modes to be specified for each operand). The result is very quick and easy instruction decoding. Therefore, the lack of complex instructions and addressing modes is compensated with the attendant increase in performance.

Except for some two register instructions and the `caxi` instruction, data manipulation instructions are non-destructive. In other words, the source operands are not modified. For instance the `addi` instruction adds the contents of one source register and a 16-bit immediate data and stores the result in a second destination register (the destination register could be the same as the source register if required). In contrast, the two register mode `add` instruction both reads and writes the second general register.

Please see the architecture specification for your specific V850 and RH850 processor for more details on the above addressing modes. These specifications can be obtained from Renesas Electronics.

Macro Expansion

The V850 assembler supports several macro expansions described below. Specifying the `-nomacro` and/or `-noexpandbranch` command line switches tells the assembler not to accept any macros but instead report an error when given a macro.

ld, st, tst1, set1, clr1

The `ld` and `st` instructions (with their `.w`, `.h`, `.b` forms) and the `tst1`, `set1`, and `clr1` instructions are expanded to several instructions as follows. (Here, `ld.w` is used in the examples.)

1. Operand is a label.

Instruction	Macro expansion
`ld.w` *`label`*`[`*`reg1`*`], `*`reg2`*	`movhi hi(`*`label`*`), `*`reg1`*`, r1` `ld.w lo(`*`label`*`)[r1], `*`reg2`*

2. Operand is an immediate value.

   If displacement is within -32768 to 32767:

Instruction	Macro expansion
`ld.w` *`disp16`*`[`*`reg1`*`], `*`reg2`*	`ld.w` *`disp16`*`[`*`reg1`*`], `*`reg2`*

3. Otherwise.

Instruction	Macro expansion
`ld.w` *`disp32`*`[`*`reg1`*`], `*`reg2`*	`movhi hi(`*`disp32`*`), `*`reg1`*`, r1` `ld.w lo(`*`disp32`*`)[r1], `*`reg2`*

add

The instruction `add` is expanded to several instructions as follows:

1. Immediate is within -16 to 15.

Instruction	Macro expansion
`add imm5, reg`	`add imm5, reg`

2. Immediate is within -32768 to 32767, except case (1).

Instruction	Macro expansion
`add imm16, reg`	`add imm16, reg, reg`

3. Otherwise

   Bits 0 through 15 are all zeros.

Instruction	Macro expansion
`add imm32, reg`	`movhi hi(imm32), r0, r1` `add r1, reg`

   Otherwise.

Instruction	Macro expansion

`add imm32, reg`	`movhi hi(imm32), r0, r1`
	`movea lo(imm32), r1, r1`
	`add r1, reg`

mov

The instruction `mov` is expanded to several instructions as follows:

1. Immediate is within -16 to 15.

Instruction	Macro expansion
`mov imm5, reg`	`mov imm5, reg`

2. Immediate is within -32768 to 32767, except case (1).

Instruction	Macro expansion
mov *imm16*, *reg*	movea *imm16*, r0, *reg*

3. Otherwise

   Bits 0 through 15 are all zeros.

Instruction	Macro expansion
mov *imm32*, *reg*	movhi hi(*imm32*), r0, *reg*

   Otherwise

   If assembling for the V850E family

Instruction	Macro expansion
mov *imm32*, *reg*	mov *imm32*, *reg*

   Otherwise

Instruction	Macro expansion
mov *imm32*, *reg*	movhi hi(*imm32*), r0, *reg* movea lo(*imm32*), *reg*, *reg*

cmp

The instruction cmp is expanded to several instructions, as follows:

1. Immediate is within -16 to 15.

Instruction	Macro expansion
cmp *imm5*, *reg*	cmp *imm5*, *reg*

2. Immediate is within -32768 to 32767, except case (1).

Instruction	Macro expansion
cmp *imm16*, *reg*	movea *imm16*, r0, r1 cmp r1, *reg*

3. Otherwise

   Same as `add` case (3).

sub, not

The instructions `sub` and `not` are expanded to several instructions as follows:

1. Immediate is zero.

Instruction	Macro expansion
`CMD 0, reg`	`CMD r0, reg`

2. Immediate is within -16 to 15.

Instruction	Macro expansion
`CMD imm5, reg`	`mov imm5, r1` `CMD r1, reg`

3. Immediate is within -32768 to 32767, except cases (1) and (2).

Instruction	Macro expansion
`CMD imm16, reg`	`movea imm16, r0, r1` `CMD r1, reg`

4. Otherwise

   Same as `add` case (3).

or, and, xor

The instructions `or`, `and`, and `xor` are expanded to several instructions as follows.

(`CMD` can be replaced by any of the instructions `or`, `and`, and `xor`).

1. Immediate is within -32768 to -17.

Instruction	Macro expansion
CMD *imm16*, *reg*	movea *imm16*, r0, r1 CMD r1, *reg*

2. Immediate is within -16 to -1.

Instruction	Macro expansion
CMD *imm5*, *reg*	mov *imm5*, r1 CMD r1, *reg*

3. Immediate is zero.

Instruction	Macro expansion
CMD 0, *reg*	CMD r0, *reg*

4. Immediate is within 1 to 65535.

Instruction	Macro expansion
CMD *imm16*, *reg*	CMDi *imm16*, *reg*, *reg*

5. Otherwise

   Same as add case (3).

addi, ori, andi, xori

The instructions addi, ori, andi, and xori are expanded to several instructions as follows.

(CMD can be replaced by any of the instructions addi, ori, andi, and xori.)

1. Immediate is within -32768 to -17.

Instruction	Macro expansion
CMDi *imm16*, *reg1*, reg2	movea *imm16*, r0, *reg2* CMD *reg1*, *reg2*

2. Immediate is within -16 to -1.

Instruction	Macro expansion
CMDi *imm5*, *reg1*, *reg2*	mov *imm5*, *reg2* CMD *reg1*, *reg2*

3. Immediate is within 0 to 65535.

Instruction	Macro expansion
CMDi *imm16*, *reg1*, *reg2*	CMDi *imm16*, *reg1*, *reg2*

4. Otherwise

   Bits 0 through 15 are all zeros.

Instruction	Macro expansion
CMDi *imm32*, *reg1*, *reg2*	movhi hi(*imm32*), r0, *reg2* CMD *reg1*, *reg2*

   Otherwise.

Instruction	Macro expansion

CMDi *imm32*, *reg1*, *reg2*	movhi hi(*imm32*), r0, r1
	movea lo(*imm32*), r1, *reg2*
	CMD *reg1*, *reg2*

movea

The instruction `movea` is expanded to several instructions as follows.

1. Immediate is within -32768 to 32767.

Instruction	Macro expansion
movea *imm16*, *reg1*, *reg2*	movea *imm16*, *reg1*, *reg2*

2. Otherwise

   Bits 0 through 15 are all zeros.

Instruction	Macro expansion
movea *imm32*, *reg1*, *reg2*	movhi hi(*imm32*), *reg1*, *reg2*

Otherwise.

Instruction	Macro expansion
movea *imm32*, *reg1*, *reg2*	movhi hi(*imm32*), *reg1*, r1 movea lo(*imm32*), r1, *reg2*

bcond

The 9-bit branch instructions b*cond* are expanded to several instructions as follows (this macro does not support a displacement that is out of the 9-bit range when expressed as an immediate):

1. Displacement is within -256 to 255.

Instruction	Macro expansion
b*cond disp9*	b*cond disp9*

2. Otherwise.

Instruction	Macro expansion
b*cond disp22*	br*cond* Label jr *disp22*-2 Label:

Here br*cond* is the reverse condition branch of b*cond*.

jbr

The 9-bit branch instruction jbr is expanded to several instructions as follows (this macro does not support an immediate displacement that is out of the 9-bit range):

1. Displacement is within -256 to 255.

Instruction	Macro expansion
`jbr` *`disp9`*	`br` *`disp9`*

2. Otherwise.

Instruction	Macro expansion
`jbr` *`disp22`*	`jr` *`disp22`*

setf, cmov, adf, sbf

The following `setf`, `cmov`, `adf`, and `sbf` macros are supported by **as850**:

(`CMD` can be replaced by any of `setf`, `cmov`, `adf`, and `sbf`. *cond* can be any of the mnemonics listed in the condition code table. *args* are the remaining arguments according to the instruction's addressing mode.)

Instruction	Macro expansion
`CMD`*`cond args`*	`CMD` *`cond, args`*

Programming in Assembly Language

This section provides brief examples with guidelines for writing V850 assembly language subroutines.

For your subroutine to work with the Green Hills V850 toolchain, it must follow the calling convention of the Green Hills V850 compilers. See "Calling Conventions" on page 34.

Throughout this section, including in assembly code, `sp0` means the value of the stack pointer as it was passed on entry to the subroutine, and `sp` means the stack pointer after the subroutine makes any adjustments. The Green Hills V850 assembler does not actually recognize the symbol `sp0`; it is used here only to avoid ambiguity.

In general, the stack looks like this:

High Address	*nth* parameter word	
	(*n*-1)*th* parameter word	
(*previous frame*)	. . .	
	6*th* parameter word	
	5*th* parameter word	`sp0`
	16-byte argument area (only if necessary)	`sp + `*`framesize`*` - 4 (= sp0 - 4)` `sp + `*`framesize`*` - 16`
	return address (`lp`)	`sp + `*`framesize`*` - `*`argarea`*` - 4`
	frame pointer (`fp`)	`sp + `*`framesize`*` - `*`argarea`*` - 8`
	permanent registers	
(*current frame*)	local variables	
	nth argument word	
	(*n*-1)*th* argument word	

	. . .	
	6*th* argument word	
	5*th* argument word	`sp (= sp0 - framesize)`
(*next frame*) Low Address	. . .	

In this diagram, *parameters* are passed to our subroutine, and *arguments* are passed to the subroutines that you call; *argarea* is 16 if you need the 16-byte argument area, or zero if not.

Assembly Code Samples

Example 8.1. C code versus Assembly code with stack usage

C code:

```
int foo(int p)
{
    int i,j;
    i = bar(13, p);
    j = cit();
    wyk();
    return i+j;
}
```

V850 assembly code:

```
      .text
      .align 2
      .globl _foo
_foo:
      add  -12,sp
      st.w  lp,8[sp]  -- save the return address
      st.w  r27,4[sp] -- save permanent register r27
      st.w  r26,0[sp] -- save permanent register r26
      mov   r6,r7     -- move p to r7
```

```
mov   13,r6
jarl  _bar, lp  -- call bar()
mov   r10,r27   -- assign result of bar() to i
jarl  _cit, lp  -- call cit()
mov   r10,r26   -- assign result of cit() to j
jarl  _wyk, lp  -- call wyk()
mov   r26,r10   -- move j to r10
add   r27,r10   -- r10 now contains i+j
ld.w  0[sp],r26 -- restore r26
ld.w  4[sp],r27 -- restore r27
ld.w  8[sp],lp  -- restore lp
add   12,sp
jmp  [lp]       -- return from subroutine
```

The stack, `framesize` = 12:

<table>
<tr><td>High Address</td><td>.
.
.</td><td>sp0</td></tr>
<tr><td rowspan="3"></td><td>lp</td><td>sp + 8</td></tr>
<tr><td>r27</td><td>sp + 4</td></tr>
<tr><td>r26</td><td>sp (=sp0 - 12)</td></tr>
<tr><td>Low Address</td><td>.
.
.</td><td></td></tr>
</table>

The `.text` directive tells the assembler that the code following the directive should be in the text segment. The `.align 2` directive forces the next instruction to align to the next 2-byte boundary. To make the subroutine `_foo` globally accessible, follow the convention of preceding an identifier with an underscore (_) and use `.globl _foo`.

In `_foo`, you need two registers to contain the values of `_i` and `_j`. You can arbitrarily pick `r27` and `r26` from the set of permanent registers (`r20` to `r29`) to contain `_i` and `_j` respectively, because these registers are preserved across subroutine calls.

Just as `_bar`, `_cit`, and `_wyk` cannot destroy `r20` to `r29` when called, you must also preserve the values of `r26` and `r27` as they were passed by the caller before being overwritten. A good place to save these registers is on entry to this subroutine.

Similarly, you need to save the contents of `r31` (or `lp`, the link pointer, which contains the return address) before making any of the calls, because the subroutine calls will overwrite its value.

You can save the contents of these three registers by storing their values onto the stack, so you will need 12 bytes of stack space. The calling convention requires the stack pointer to always align to a 4-byte boundary, so you do not need to adjust the stack any further. Since the stack "grows" downward from higher addresses to lower addresses, subtract 12 from the stack pointer `sp0` (`r3`) on entry to this subroutine.

Now you have 12 bytes of stack space to store `r26`, `r27`, and `lp`. In this case, the Green Hills compiler saves the return address register at the highest address on the local stack frame, so you can store `lp` at `sp+8` (with the instruction `st.w lp,8[sp]`). Store `r26` at `sp+0` and `r27` at `sp+4`.

Next, make a call to `_bar`, passing 13 as the first parameter and `_p` as the second parameter. To pass 13, put it in `r6`, the first parameter register. However, `_p` is the first parameter to this subroutine, and it is in `r6` at this point, so you must first move it from `r6` to `r7`, the second parameter register. Then you can move 13 into `r6`, and make the call to `_bar`.

Since `r10` is the return register, you can find the value of `_bar` and assign the result to `_i`. Register `r10` is moved to `r27` before calling `_cit`. Then you can assign the return value `r10` to `_j` (in `r26`) and call `_wyk`. Finally, add `_i` and `_j` and return the result in `r10`.

On exit from `_foo`, you can restore the values of these three registers by the corresponding loads, adjust the stack pointer back, and return to the address in `lp` (with the instruction `jmp [lp]`).

Example 8.2. Setting up a stack frame to make a function call

C code:

```
void foo(void)
{
    bar(3,1,4,1,5,9);
}
```

V850 assembly code:

```
        .text
        .align 2
        .globl _foo
_foo:
        add   -12,sp
        st.w  lp,8[sp]  -- save return address
        mov   3,r6      -- pass first 4 parameters in registers
        mov   1,r7
        mov   4,r8
        mov   1,r9
        mov   5,r16     -- pass the 5th and 6th on the stack
        st.w  r16,0[sp]
        mov   9,r16
        st.w  r16,4[sp]
        jarl  _bar, lp  -- call bar()
        ld.w  8[sp],lp  -- restore return address
        add   12,sp
        jmp   [lp]      -- return from subroutine
```

The stack, `framesize` = 12:

High Address	.	
	.	
	.	
	lp	sp + 8
	9	sp + 4
	5	sp (= sp0 - 12)
Low Address	.	
	.	
	.	

Here `_foo` passes six int's to `_bar`. You have four parameter registers `r6`, `r7`, `r8`, and `r9`, so passing the first four arguments is trivial. However, the 5*th* and 6*th* arguments must go onto the stack. You also need 8 bytes to do that, so put the 5*th* and 6*th* arguments at `sp+0` and `sp+4` respectively.

Again you need to save the return address `lp`, making a total of 12 bytes on the stack.

Example 8.3. A leaf subroutine

C code:

```
int foo(int i, int j)
{
    return i+j;
}
```

V850 assembly code:

```
      .text
      .align 2
      .globl _foo
_foo:
      mov  r7,r10   -- move j to r10
      add  r6,r10   -- add i to it. r10 now equals i+j
      jmp  [lp]     -- return from subroutine
```

This example demonstrates a *leaf* subroutine, one that does not make any calls. Also, `_foo` does not need to save any registers. In this case you do not have to adjust the stack pointer.

Example 8.4. A function whose arguments are passed on the stack

C code:

```
int foo(a,b,c,d,e,f)
int a,b,c,d,e,f;
{
    return e+f;
}
```

V850 assembly code:

```
      .text
      .align 2
      .globl _foo
_foo:
      ld.w  4[sp],r10  -- load e into r10
      ld.w  0[sp],r15  -- load f into r15
      add   r15,r10    -- add them, leaving the sum in r10
      jmp   [lp]       -- return from subroutine
```

The stack, *framesize* = 0 (no need to create a new stack frame):

High Address	. . .	
	_f	sp0 + 4
	_e	sp0
Low Address	. . .	

On entry to `_foo`, the caller has stored `_e` and `_f` on the stack at `sp0+0` and `sp0+4`, respectively. This is also a leaf subroutine.

Example 8.5. Arguments that must be passed in register pairs

C code:

```
void foo(int i, double d, int j)
{
    bar(2,i,j,d);
}
```

V850 assembly code:

```
      .text
      .align 2
      .globl _foo
_foo:
      add   -12,sp
      st.w  lp,8[sp]    -- save return address
      st.w  r8,0[sp]    -- store d on the stack
      st.w  r9,4[sp]
      ld.w  12[sp],r8   -- load j from the stack
      mov   r6,r7       -- move i to r7
      mov   2,r6
      jarl  _bar, lp    -- call bar()
      ld.w  8[sp],lp    -- restore lp
      add   12,sp
      jmp   [lp]        -- return from subroutine
```

The stack, *framesize* = 12:

High Address	. . .	
	_j	sp0 (= sp + 12)
	lp	sp + 8
	_d(r9)	sp + 4
	_d(r8)	sp (= sp0 - 12)
Low Address	. . .	

In this example, `_d` is passed *not* in (`r7`, `r8`) but in (`r8`, `r9`) because a double, if stored in memory, must have an 8-byte aligned offset relative to the first argument's stack offset. Therefore, the first parameter `_i` has a zero offset, `_d` an offset of 8, and `_j` an offset of 16. So the caller put `_i` in `r6`, skipped `_r7`, put `_d` in (`r8`, `r9`), and stored `_j` onto the stack at `sp0`.

Similarly, to call `_bar`, pass 2, `_i`, and `_j` in `r6`, `r7`, and `r8` respectively, skip `r9`, and then pass `_d` at `sp+0` and `sp+4`.

Example 8.6. Using the homing area

C code:

```
struct T { int i; int j; };

void foo(int k, struct T s)
{
    bar(&s.i, k);
}
```

V850 assembly code:

```
      .text
      .align 2
      .globl _foo
_foo:
      addi  -20,sp,sp
```

```
st.w  r8,12[sp] -- store s.j onto the stack
st.w  r7,8[sp]  -- store s.i onto the stack
st.w  lp,0[sp]  -- save the return address
mov   r6,r7     -- move k to r7
moeva 8,sp,r6   -- move the address of s.i to r6
jarl  _bar, lp  -- call bar()
ld.w  0[sp],lp  -- restore lp
addi  20,sp,sp
jmp   [lp]      -- return from subroutine
```

The stack, `framesize` = 20:

High Address	.	
	.	`sp0 (= sp + 20)`
	.	
	(word for r9)	`sp + 16`
	`s.j` (`r8`)	`sp + 12`
	`s.i` (`r7`)	`sp + 8`
	(word for r6)	`sp + 4`
	`lp`	`sp (= sp0 - 20)`
Low Address	.	
	.	
	.	

Here is a structure parameter passed by value, and part of the structure (in this case the entire structure) is within the first four words (the second word specifically) of the parameter list. Therefore, the caller passed `k` in `r6` and `s` in `r7`. The calling convention requires that you allocate a 16-byte "homing" area immediately following the previous stack frame and save the parts of the structure within the first four words into this area. Below this area you can save the linker pointer (`lp`), etc. as usual.

The following is a variation of this example.

Example 8.7. Another homing area example

C code:

```
struct T { int i; int j; };

void foo(struct T s, struct T v)
{
    bar(&s.i, &v.i);
}
```

V850 assembly code:

```
      .text
      .align 2
      .globl _foo
_foo:
      addi  -20,sp,sp
      st.w  r9,16[sp]   -- store v.j onto the stack
      st.w  r8,12[sp]   -- store v.i onto the stack
      st.w  r7,8[sp]    -- store s.j onto the stack
      st.w  r6,4[sp]    -- store s.i onto the stack
      st.w  lp,0[sp]    -- save the return address
      movea 12,sp,r7    -- move the address of v.i to r7
      movea 4,sp,r6     -- move the address of s.i to r6
      jarl  _bar, lp    -- call bar()
      ld.w  0[sp],lp    -- restore lp
      addi  20,sp,sp
      jmp   [lp]        -- return from subroutine
```

The stack, `framesize` = 20:

<table>
<tr><td>High Address</td><td>.
.
.</td><td>sp0 (= sp + 20)</td></tr>
<tr><td rowspan="5"></td><td>v.j (r9)</td><td>sp + 16</td></tr>
<tr><td>v.i (r8)</td><td>sp + 12</td></tr>
<tr><td>s.j (r7)</td><td>sp + 8</td></tr>
<tr><td>s.i (r6)</td><td>sp + 4</td></tr>
<tr><td>lp</td><td>sp (= sp0 - 20)</td></tr>
<tr><td>Low Address</td><td>.
.
.</td><td></td></tr>
</table>

The following is another variation:

Example 8.8. When more than the homing area is needed

C code:

```
struct A { int i; };
struct B { int a,b,c; };

void foo(struct A s, int k, struct B v)
{
    bar(&s.i, &v.c, k);
}
```

V850 assembly code:

```
     .text
     .align 2
     .globl _foo
_foo:
     addi  -20,sp,sp
     st.w  r9,16[sp]   -- store v.a onto the stack
     st.w  r7,8[sp]    -- store s.j onto the stack
     st.w  r6,4[sp]    -- store s.i onto the stack
     st.w  lp,0[sp]    -- save the return address
     movea 24,sp,r7    -- move the address of v.c to r7
     movea 4,sp,r6     -- move the address of s.i to r6
     jarl  _bar, lp    -- call bar()
     ld.w  0[sp],lp    -- restore lp
     addi  20,sp,sp
     jmp   [lp]        -- return from subroutine
```

The stack, *framesize* = 20:

<table>
<tr><td rowspan="4">High Address</td><td>.
.
.</td><td rowspan="2"></td></tr>
<tr><td>v.c</td><td>sp0 (= sp + 24)</td></tr>
<tr><td></td><td>v.b</td><td>sp + 20</td></tr>
<tr><td rowspan="2"></td><td>v.a (r9)</td><td>sp + 16</td></tr>
<tr><td>(word for r8)</td><td>sp + 12</td></tr>
</table>

	s.j (r6)	sp + 8
	s.i (r6)	sp + 4
	lp	sp (= sp0 - 20)
Low Address	. . .	

Bitfields

The V850 provides special bitfield manipulation instructions for modifying individual bits in memory. Using the driver (**ccv850**) and the preprocessor facility `#define`, you can make bitfield manipulation assembly code easier to read.

In the following example, the address of `bar` is loaded into `r10`, then `bit0`, `bit1`, and `bit10` are defined using the *`bit,byte`* notation to name three single bits into `bar`. To compile this file, **ccv850** is used.

```
      .data
_bar:
      .word 0x12345678
      .text
      .align 2
      .globl _foo
_foo:
      movhi hi(_bar),zero,r10
      movea lo(_bar),r10,r10

#define bit0 0,0[r10]
#define bit1 1,0[r10]
#define bit10 2,1[r10]

      clr1  bit0
      cmp   zero,r6
      bne   L
      set1  bit1
      jmp   [lp]
L:
      clr1  bit10
      jmp   [lp]
```

Chapter 9

The elxr Linker

Contents

The Green Hills **elxr** linker is invoked by the driver and combines ELF object files and libraries into a single executable program suitable for downloading onto your target. To perform this task, it:

1. combines all input files into a single output program that contains all necessary objects.
2. assigns a memory address to each of the objects.
3. resolves relocations by replacing references to each object with the memory address it assigned to that object.

When the linker combines object files, it uses a *section map* to determine which sections in each object file to place into each program section. It also reads the section map to assign program sections to memory blocks or addresses. The linker reads a *memory map* to determine the size and location of memory blocks on your target. The linker reads section maps, memory maps, and options from linker directives (**.ld**) files (see "Configuring the Linker with Linker Directives Files" on page 441).

Linker-Specific Options

The driver recognizes many different options and passes them to the linker. For a list of these driver options, see “Linker Options” on page 219. If you use the driver to compile and link your program in multiple steps, you must pass the exact same set of options for each step. Though you should use driver options whenever possible, the linker accepts a small set of options that the driver does not recognize. Be careful when using linker-specific options, because they are not processed by the driver and may conflict with driver options. To pass an option directly to the linker:

Use the **Linker→Additional Linker Options (before start file)** option (**-lnk=**). See “Additional Linker Options (before start file)” on page 223 for more information about this option.

It is never necessary to invoke the linker directly; always specify linker-specific options through the driver.

Note

During the C compilation process, an underscore (“_”) is added to the beginning of all function and variable names. Consequently, when specifying a function or variable name as the argument to a linker option, you must add an underscore so that the linker can recognize the name in the object file. Linker options that take function or variable names as arguments include **-u**, **-keep**, and **-e**.

For example, to retain the function `foo`, when instructing the linker to perform function deletion, you would pass the following options:

```
elxr -delete -keep=_foo file.o
```

The following table lists linker-specific options:

-allabsolute Imports all absolute symbols, not just those required. For use with the **-A** driver option (see “Import Symbols” on page 229).
-allow_bad_relocations Emits only warnings when encountering bad or unknown relocations. The default behavior is to generate errors.

-any_p_align

Does not set an upper bound on the value of the `p_align` field in ELF program headers.

-max_p_align=*n*

Limits the maximum value of the `p_align` field set in ELF program headers to *`n`* bytes. This field is set based on the maximum section alignment requirement of sections contained by this header, unless it would exceed the `max_p_align` value.

-no_p_align

Always sets the `p_align` field in ELF program headers to `0`. This option may allow a small amount of alignment padding within ELF files, but it does not set the field in the program header.

-append

Suppresses warnings concerning object file sections that do not appear in the section map.

-earlyabsolute

Imports absolute symbols before searching libraries. For use with the **-A** driver option (see “Import Symbols” on page 229).

-extractall

Forces all object files from libraries to be included in the executable. This option is different from the **-extractall=** driver option in that it affects all libraries, not just the one specified (see “Force All Symbols From Library” on page 228).

-extractweak

Pulls in object files from libraries if they provide a definition for an undefined weak reference. Normally, object files are only pulled in from a library if they provide a definition for an undefined non-weak reference.

-help

Displays a list of the most commonly used linker options.

-Help

Displays a list of all available linker options.

-keep=*symbol*

Prevents the deletion of the function pointed to by *symbol* when you specify the **-delete** option. If you are using wholeprogram optimizations, you may also need to use “Symbols Referenced Externally” on page 150.

-no_append

Promotes warnings concerning sections that do not appear in the section map to errors.

-no_crom

Disables the use of compressed ROM. Sections that are specified as compressed ROM copies (through use of the attribute `CROM`) are treated as if they were specified as ROM copies using the `ROM` attribute. For more information about the `CROM` attribute, see "Using Section Attributes" on page 456.

-nosegments_always_executable

By default, the linker sets the execute bit (`p_flags & 0x1`) on all ELF program headers, meaning that code can be executed from any segment. If you pass this option to the linker, it sets the program header execute bit only for segments that have at least one input section that has its execute bit (`sh_flags & 0x4`) set.

-no_uncompressed_copy

By default, the linker creates a notes section called `.UNCOMPRESSED`*secname* for each compressed ROM section *secname*. For example, the linker would create `.UNCOMPRESSED.text` for the `.text` section. This notes section contains an uncompressed copy of the section's contents for use by the debugger. This notes section does not increase the size of the program image in memory, but it does increase the size of the ELF file itself. Use **-no_uncompressed_copy** to disable the generation of `.UNCOMPRESSED` notes sections, resulting in smaller ELF files and limiting your ability to debug CROM sections.

-no_xda_modifications

By default, the linker will accept out-of-range SDA relocations if the destination symbol is within the ROM SDA (if enabled) region or ZDA region. It will also accept out-of-range ROM SDA relocations if the destination symbol is within the SDA (if enabled) region or ZDA region. The linker achieves this by modifying the relocated instruction to use the appropriate base register. Customers requiring that the linker not modify instruction operands other than the immediate may find this option useful. However, **-no_xda_modifications** is necessary for some C++ programs to link without generating a linker error.

-prog2

Produces exactly two segments: one to contain sections `.text` and `.rodata`, and the other to contain `.data` and `.bss`.

-T *file.ld*

Reads additional options, memory maps, and section maps from the linker directives file *file.ld*. You do not need to pass this option manually, because the driver determines the correct **-T** options to pass to the linker.

-unalign_debug_sections

Overrides the section alignment specified in object files and forces `.debug` sections to have a 1-byte alignment.

-underscore

Adds an underscore to the beginning of all symbols created at link time.

-unweak *symbol*

If any undefined weak reference to *symbol* exists, it is promoted to a normal undefined reference.

-v

Prints verbose information about the activities of the linker, including the libraries it searches to resolve undefined symbols.

-V

Displays the version number and copyright information for **elxr**.

-vimport

Prints information about why modules are pulled in from libraries. This information is a subset of the information provided by **-v**.

-w

Suppresses linker warnings.

-wrap *sym*

Resolves external references to `_`*`sym`* to `___wrap_`*`sym`*, and external references to `___real_`*`sym`* to `_`*`sym`*. This corresponds to C symbol names of *`sym`*, `__wrap_`*`sym`*, and `__real_`*`sym`*. This can be used to provide your own wrapper function that can be called instead of the original function from a library.

For example, if a third party library defines:

```
void critical_section(void);
```

you can check the calls to `critical_section()` by passing **-wrap** `critical_section` to the linker, and then define the following:

```
void __wrap_critical_section(void)
{
    assert(is_safe_to_call_critical_section());
    __real_critical_section();
}
```

Specifying the Program Entry Point

Use the **Linker→Start Address Symbol** option (**-e**) to specify the program entry point.

If you do not set this option, the linker sets the entry point to an item in the following list (in descending order of preference):

- The value of the symbol `_start`, if present
- The value of the symbol `start`, if present
- The value of the symbol `_main`, if present
- The value of the symbol `main`, if present
- The zero address

If the entry symbol is copied from ROM to RAM during program startup, the entry point is in the ROM version of the entry symbol.

Configuring the Linker with Linker Directives Files

This section explains how to use linker directives (**.ld**) files to define program sections and assign those sections to memory blocks. The names, address, and sizes of memory blocks are defined in a *memory map*. The names, attributes, and locations of sections are defined in a *section map*. This section also describes linker directives file syntax, describes the four linker directives, and discusses some advanced topics:

- "Linker Directives File Syntax" on page 442
- "Setting Linker Options with the OPTION Directive" on page 442
- "Setting Defaults with the DEFAULTS Directive" on page 443
- "Defining a Memory Map with the MEMORY Directive" on page 445
- "Defining a Section Map with the SECTIONS Directive" on page 447
- "Modifying your Section Map for Speed or Size" on page 463
- "Customizing the Run-Time Environment Program Sections" on page 464

When you create a project with the **Project Wizard**, it generates default linker directives files for various program layouts for your board. If you want to write your own linker directives file, we recommend that you copy one of these default

files and modify it. For information about these default files, see “Working with Linker Directives Files” on page 74.

Use caution when removing program sections from the default section map, because the Green Hills run-time environment might use them (see “Customizing the Run-Time Environment Program Sections” on page 464).

Note

If you specify multiple linker directives files, **elxr** processes those files in the order you specify them. If you define a memory map and section map in separate files, you must specify the file containing the memory map first.

Linker Directives File Syntax

This section covers general linker directives file syntax and expressions. For information about syntax for a specific directive, see the documentation for that directive.

- Expressions — include all standard C assignment operators with their normal precedence, as well as operators that have side effects. The following sections indicate where you can use expressions in each directive.
- Comments — begin with the number symbol (`#`) or two forward slashes (`//`). If you use the **Linker→Preprocess Linker Directives** option, you must use forward slashes when writing comments.
- The dot symbol (`.`) — evaluates to the linker's current position in laying out memory blocks or sections. You can change the value of the dot symbol using expressions.

Setting Linker Options with the OPTION Directive

You can include the linker options listed in “Linker-Specific Options” on page 437 in a linker directives file using the following syntax:

```
OPTION ("option…")
```

You cannot use expressions in the `OPTION` directive.

Note

This directive is provided for backwards compatibility; we recommend that you specify linker options using the driver.

Example 9.1. Writing an OPTION Directive

This example defines an `OPTION` directive that specifies the **-extractweak** and **-append** options:

```
OPTION ("-extractweak -append")
```

Note

You can also use the `OPTION` directive to specify which files you want to link, using the following syntax:

```
OPTION ("file")
```

Setting Defaults with the DEFAULTS Directive

Defaults are strings that you can use in place of absolute values when defining section and memory maps in a linker directives file. Specify defaults with the following syntax:

```
DEFAULTS {name=value…}
```

To set the default `foo` to the value `0x2000`, add the following line:

```
DEFAULTS {foo=0x2000}
```

You can specify multiple defaults by putting each default on its own line:

```
DEFAULTS {
   foo = 0x2000
   bar = 0x4000
}
```

You can use expressions in the `DEFAULTS` directive that include defaults set earlier in the directive:

```
DEFAULTS {
   foo = 0x2000
```

```
// Set bar to 0x4000
   bar = foo + 0x2000
}
```

When you specify an amount of memory, you can suffix the number of bytes with `M` to designate megabytes and `K` to designate kilobytes. You can also prefix the number with `0x` to specify the amount in hexadecimal.

Note

If you specify a default with the `DEFAULTS` directive, you can change its value at link time by passing the **-C** option (see “Define Linker Constant” on page 228).

Example 9.2. Writing a DEFAULTS Directive

This example specifies a `DEFAULTS` directive that defines two defaults:

```
DEFAULTS {
// Set heap_reserve to 1048576
   heap_reserve   = 1M
// Set stack_reserve to 524288
   stack_reserve  = 512K
}
```

Defining a Memory Map with the MEMORY Directive

A memory map names and defines memory blocks on your target. Specify each of the following elements to define a memory block:

```
MEMORY
{
memory_block: ORIGIN=origin_expression, LENGTH=length_expression
...
}
```

where:

memory_block Specifies the name of a memory block.
origin_expression Specifies the starting address of the memory block.
length_expression Specifies the length of the memory block.

Expressions in the `MEMORY` directive can include:

- defaults set in the `DEFAULTS` directive.
- the dot symbol (`.`). In this context, the dot symbol evaluates to the first address available after the end of the previous memory block.

If you pass a memory map to the linker, only the memory blocks named and defined in it can be referenced in an associated section map.

Example 9.3. Writing a MEMORY Directive

In this example, the `MEMORY` directive defines a memory map with two blocks:

```
DEFAULTS {
   heap_reserve   = 1M
   stack_reserve  = 512K
}

MEMORY
{
   fast_memory  : ORIGIN = 0x80000000, LENGTH = 10M
   slow_memory  : ORIGIN = 0xbfc00000, LENGTH = 10M
}
```

This memory map defines the memory blocks `fast_memory`, which begins at `0x80000000`, and `slow_memory`, which begins at `0xbfc0000`. They are both 10 megabytes in size.

Reserved Memory Blocks

Sometimes the default Green Hills linker directives files contain reserved memory blocks. Typically, this is done to avoid using memory that preconfigured software on your target, such as a monitor, already uses.

Reserved memory blocks are just like any other memory blocks, except that you should not allocate sections to them.

```
MEMORY {

// 10MB of dram starting at 0x80000000

   dram_rsvd1   : ORIGIN = 0x80000000, LENGTH = 32K
   dram_memory  : ORIGIN = .,          LENGTH = 10M -32K
   dram_rsvd2   : ORIGIN = .,          LENGTH = 0

// 10MB of flash starting at 0xbfc00000

   flash_rsvd1  : ORIGIN = 0xbfc00000, LENGTH = 32K
   flash_memory : ORIGIN = .,          LENGTH = 10M-32K
   flash_rsvd2  : ORIGIN = .,          LENGTH = 0

}
```

Defining a Section Map with the SECTIONS Directive

A section map maps each section in each object file to a program section. The section map also maps program sections to memory blocks. Format a section map as follows:

```
SECTIONS
{
secname [start_expression] [attributes] : [{ contents }] [> memory_block,… | >.]
...
}
```

where:

secname Specifies name of the section.
start_expression Specifies the starting address of the section. If you specify both *start_expression* and *memory_block*, *start_expression* takes precedence.
attributes Specifies section attributes. You can use expressions in attribute parameters. For a list of section attributes, see “Using Section Attributes” on page 456.
{ *contents* } Specifies the contents of the section. It contains *section inclusion commands* that dictate which sections from which object files to place into the section, and assignments that create or modify objects. You can use an unlimited number of commands and assignments.
> *memory_block*,… | >. Allocates the section to the memory block named *memory_block*, defined in the memory map. If you specify both *start_expression* and *memory_block*, *start_expression* takes precedence.

Expressions in the `SECTIONS` directive can include:

- defaults defined in the `DEFAULTS` directive.
- memory blocks defined in the `MEMORY` directive.
- the dot symbol (`.`). In this context, the dot symbol evaluates to the linker's current position in laying out objects and sections.
- ELF symbols defined in the object files passed to the linker.
- the functions listed in “SECTIONS Directive Functions” on page 454.

Specifying the Location of a Section

The following list describes where the linker allocates a section in memory depending on how you specify *start_expression* and *memory_block*:

- If you specify *start_expression*, the linker allocates the section starting at the address *start_expression*, whether or not you specify *memory_block*.
- If you specify just *memory_block*, the linker allocates the section starting at the first available address in *memory_block*. If the section does not fit, the linker returns an error.
- If you do not specify *start_expression*, but you specify the dot symbol for *memory_block*, the linker allocates the section directly after the previous section, in the same memory block. If the section does not fit, the linker returns an error.
- If you do not specify *start_expression*, but you specify a comma-separated list of memory blocks for *memory_block*, the linker allocates the section to the first specified memory block with enough remaining space to hold it.
- If you do not specify either *start_expression* or *memory_block*, the linker allocates the section directly after the previous section, in the same memory block. If the section does not fit, the linker allocates the section to the first memory block with enough remaining space to hold it.

No matter whether the linker determines the start address from *start_expression* or *memory_block*, it might increase the address to satisfy the alignment requirements of its contents. For example, say you have the following directive:

```
SECTIONS {
   .text    0x10001 :
}
```

The final executable's `.text` section might be located at `0x10004` instead of `0x10001` if the `.text` section of any object file the linker places in this section requires 4-byte alignment.

Example 9.4. Writing a SECTIONS Directive

In this example, the `SECTIONS` directive defines a section map that tells the linker where to place the `.text`, `.data`, `.bss`, and custom `.mydata` sections in memory on your target:

```
DEFAULTS {
   heap_reserve   = 1M
```

```
   stack_reserve  = 512K
}

MEMORY
{
   fast_memory  : ORIGIN = 0x80000000, LENGTH = 10M
   other_memory : ORIGIN = 0xbfc00000, LENGTH = 8M
   slow_memory  : ORIGIN = 0xf2c00000, LENGTH = 10M
}

SECTIONS
{
// Place .text into fast_memory. Fail if it does not fit.
   .text                  : > fast_memory
// Place .data into fast_memory after .text, or into other_memory if .data
// cannot fit into fast_memory, or into slow_memory if .data cannot fit
// into other_memory. Otherwise, fail.
   .data                  :
// Place .bss after .data. Fail if it does not fit in the same memory block.
   .bss                   : >.
// Place .mysection starting at the first available address in fast_memory.
// If it does not fit, place it into other_memory instead.
   .mysection             : > fast_memory, other_memory
}
```

Specifying Multiple Memory Blocks for a Section

If you specify a single memory block for a section, and the section does not fit in that memory block, the linker returns an error. If you specify multiple memory blocks, the linker places the section in the first specified memory block with enough remaining space to hold it (see Example 9.4. Writing a SECTIONS Directive on page 448).

On targets that support Small Data Area (SDA) optimization, be careful when using a comma-separated list of memory blocks with SDA sections. For example, it is dangerous to write:

```
SECTIONS {
   ...
   .sbss     : > fast_but_small_memory, slow_but_large_memory
   .sdata    : > fast_but_small_memory, slow_but_large_memory
   ...
}
```

because `.sdata` and `.sbss` must be contiguous in memory. When using this section map, the linker might put `.sbss` in the `fast_but_small_memory` block and `.sdata` in the `slow_but_large_memory` block (or vice versa, if `.sbss` does not fit in the `fast_but_small_memory` block but `.sdata` does). Therefore, you

should not use this syntax on SDA sections. See “Special Data Area Optimizations” on page 87 for more information.

Writing Section Inclusion Commands

Section inclusion commands are part of the `{ contents }` parameter in a line in the section map. By default, the linker places each section from each object file into the program section with the same name. If there is no program section with the same name, the linker creates a new program section at the end of the current section map and emits a warning (to silence or promote these warnings, see the **-append** and **-no_append** linker options).

If you want to change this behavior, use section inclusion commands to dictate which sections from which object files the linker should place into each section of the program. The linker executes section inclusion commands in the order that it encounters them in the linker directives file. They use the syntax *filename*(*section_name*):

```
// Place the .data section from foo.o into .mydata
   .mydata             :{ foo.o(.data) }
// Then, place .data sections from all remaining object files into .data
   .data               :
```

Separate section inclusion commands with spaces, and write them in the order that you want the linker to execute them:

```
// Place the .data section from foo.o and bar.o into .mydata
// The foo.o .data section goes first
   .mydata             :{ foo.o(.data) bar.o(.data) }
```

If you want to place sections from a specific object file in a specific library into a program section, include the library name using the form *library_name*(*filename*(*section_name*)):

```
// Place the .data section from foo.o contained within lib.a into .mydata
   .mydata             :{ lib.a(foo.o(.data)) }
```

Note

Do not include the full path to the library; use just the library name. Library names are case-sensitive.

Using Wildcards in Section Inclusion Commands

Use an asterisk ("*") in place of *filename* in a section inclusion command to specify all object files passed to the linker:

```
// Place .data sections from all object files into .data
   .data              :{ *(.data) }
```

Because the program section and the object file sections have the same name, you do not need to specify a `{ contents }` argument in this case, and the following syntax is sufficient:

```
// Place .data sections from all object files into .data
   .data              :
```

If the program section and object file sections have different names, you must specify a `{ contents }` argument:

```
// Place .data sections from all object files into .mydata
   .mydata            :{ *(.data) }
```

You can also use wildcards inside *filename* and *section_name*. An asterisk ("*") matches multiple characters and a question mark ("?") matches any single character. When you use a wildcard in this fashion, surround the section inclusion command in double quotes (").

```
// Place .text sections from all object files matching "code_*.o" into .mytext
   .mytext            :{ "code_*.o(.text)" }
```

If you have multiple sections in your object files that have a similar naming convention, you can use wildcards in a section inclusion command to assign them all to the same section in the output executable:

```
// Place all sections matching ".text_*" into .mytext
   .mytext            :{ "*(.text_*)" }
```

You can also use wildcards when specifying a library:

```
// Place .data sections from object files contained within lib.a into .mydata
   .mydata            :{ lib.a(*(.data)) }
// Place .data sections from all other object files into .data
   .data              :
```

Using the COMMON Section in Section Inclusion Commands

The .bss section of a fully linked executable contains all the data from the .bss section of each included object file. It also contains uninitialized global data from the COMMON section. To instruct the linker to place COMMON data in a different section from .bss data, use the COMMON keyword in a section inclusion command:

```
// Place all COMMON data into .mybss
   .mybss              :{ *(COMMON) }
// Place all .bss data into .bss
   .bss                :
```

To allocate only the COMMON data from the **main.o** object file to .mybss, enter the following:

```
   .mybss              :{ main.o(COMMON) }
```

You can also do this with the .sbss section and SMALLCOMMON, and the .zbss section and ZEROCOMMON.

Using Assignments

The syntax for an assignment is *symbol = expression*;. You can set a symbol to an absolute address or an address relative to a section.

- To set a symbol to an absolute address, write the assignment on its own line in the SECTIONS directive. The linker evaluates *expression* to an address, and sets the symbol to that address. The dot symbol (.) evaluates to the absolute address at the end of the last defined section.
- To set a symbol to an address relative to a section, write the assignment inside the { *contents* } parameter for that section. The linker evaluates *expression* to an offset *n*, and sets the symbol to an address *n* bytes from the beginning of the section. The dot symbol evaluates to the current offset from the beginning of the section.

You can assign addresses to symbols whether or not they exist:

- If you assign an address to a non-existent symbol, the linker creates that symbol.
- If you assign an address to an existing symbol, the linker reassigns that symbol without moving any data associated with that symbol. For example, if you

assign an address to `main`, the linker causes relocations to `main` to point to the new address, but it does not move the actual code for `main` to that address.

To add padding at the current position in a section, increment dot. If you decrease the value of dot, the linker issues an error. The following example sets the symbol `foo` to the beginning of `.mysection` and adds 4 bytes of padding between `foo` and `bar`:

```
.mysection : { foo = .; . += 4; bar = .; }
```

Without this padding, the linker would assign the same address to both `foo` and `bar`.

Example 9.5. Working with Absolute and Relative Assignments

This example shows you how to set symbols to absolute addresses and addresses relative to a section.

```
SECTIONS {
   // Set alias to the same absolute address as printf
   alias = printf;

   // Assignments end with semicolons, but section inclusion commands do not.
   .text : { alpha = 0; *(.text) beta = .; *(.text2) bad = printf; } > dram_memory

   // Set gamma to address 0
   gamma = 0;

   // Set delta to the end of the .text section
   delta = .;
}
```

`alias`, `gamma`, and `delta` are assigned absolute addresses, because they are not defined inside the contents of any section. `alpha`, `beta`, and `bad` are assigned addresses relative to the beginning of `.text`, because they are defined inside its `{` *`contents`* `}` parameter.

The linker assigns addresses to the symbols in `.text` as follows:

- `alpha` is located at offset `0` from the start of `.text`.
- `beta` is located in `.text` after the `.text` input sections, at the same address as the first object in the `.text2` input sections.
- `bad` is located at an offset that is `printf` bytes from the start of `.text`. It is not located at the same address as `printf`, because the assignment is relative

to the section. For example, if `.text` starts at `0x4000` and `printf` is located at `0x4020`, then `bad` is located at `.text+0x4020`, or `0x8020`.

SECTIONS Directive Functions

The linker recognizes the following functions during expression evaluation in the `SECTIONS` directive. Their names are case-insensitive.

`absolute(`*`expr`*`)` Given a section-relative offset value, `absolute` returns the absolute address by adding the address of the containing section to *expr*. It is an error to use `absolute` outside of the braces that specify a section's contents.
`addr(`*`sectname`*`)` Returns the memory address of the section or memory block *sectname*.

`align(`*`value`*`)`
Returns the current position (`.`) aligned to a *value*-byte boundary. *value* must be a power of two. This is equivalent to:
`(. + `*`value`*` - 1) & ~(`*`value`*` - 1)`

`alignof(`*`sectname`*`)` Returns the alignment of the section *sectname*.
`endaddr(`*`sectname`*`)` Returns the memory address of the current end of the section *sectname*, or the end of the memory block *sectname*.
`error("`*`string`*`")` Generates a linker error, displaying *string*, the current section's name and address, and the current section offset.
`final(`*`finalexpression`*` [,`*`earlyexpression=0`*`])` Evaluates the given expression only during the final layout pass. The linker evaluates the optional second argument during all preceding passes. All symbols are expected to be resolved prior to the final layout pass, but addresses are subject to change between passes.
`isdefined(`*`symbol`*`)` Returns `1` if *symbol* exists and is defined, `0` otherwise.
`isweak(`*`symbol`*`)` Returns `1` if *symbol* is a weak global symbol, `0` otherwise.

`memaddr(`*`sectname`*`)` See `addr`.
`memendaddr(`*`sectname`*`)` See `endaddr`.

`min(`*`value1`*`,`*`value2`*`)`
`max(`*`value1`*`,`*`value2`*`)`
Returns the minimum or maximum, respectively, of the two values supplied.

`pack_or_align(`*`value`*`)`
Returns the current position (`.`) aligned such that the section cannot span a page boundary of size *value*. Generally, you only use this expression as the *start_expression* for a section map. It is equivalent to:
`((. % value)+sizeof(this_section)>value?align(value):.)`

`provide`(*name* = *expr*) If the symbol *`name`* is not yet defined, then this is equivalent to *`name = expr`*. Otherwise, it is equivalent to *`expr`*.
`sizeof(`*`sectname`*`)` Returns the current size of the section *sectname*, or the size of the memory block *sectname*.

Example 9.6. Incrementing the Address of an Existing Symbol

Depending on your target and the optimizations you enable, the linker might perform program layout multiple times. When necessary, use the `final()` function to ensure that the linker evaluates an expression during the final layout only.

For example, say that if the symbol `func` exists, you want to increment its address by one. The following assignment might cause the linker to increment the address of `func` multiple times:

```
// Incorrect - might increment multiple times
   .text : { isdefined(func) ? (func += 1) : 0; }
```

Instead, use the `final()` function to ensure that the linker increments the address of `func` once:

```
// Correct - only increments during the final layout
   .text : { final(isdefined(func) ? (func += 1):0); }
```

Using Section Attributes

Use the following attributes to configure section behavior:

`ABS` Sets a flag in the output file that indicates that this section has an absolute address and should not be moved. Program loaders and other utilities that manipulate the output image should not include this section in any movement related to position independent code or position independent data.
`ALIGN(`*n*`)` Aligns the start address of the section to the byte boundary given by *n*.

`AT(`*expr*`)`
`LOAD(`*expr*`)`
These attributes specify an expression *expr* that indicates the address where the section will be loaded in memory. This is only relevant if the address the section is linked to is different than the address where the section should be loaded.
Note that `AT`, unlike `LOAD`, must be placed after the `:` in the section map. This is mainly used for enhanced compatibility with GNU linker directives files.

`CLEAR`

`NOCLEAR`

Sets or removes the `CLEAR` attribute of the section. If `CLEAR` is set, the linker makes an entry for the section in the run-time clear table. Startup code often uses this table to initialize memory blocks to zero (see "Run-Time Clear and Copy Tables" on page 476).

`CLEAR` is specified implicitly on `.bss`, `.zbss`, and `.sbss` sections; all other sections default to `NOCLEAR`. If you direct the linker to include `COMMON`, `ZEROCOMMON`, or `SMALLCOMMON` data in any other section, you should specify the `CLEAR` attribute for that section. For suggested uses, see Example 9.8. Using CLEAR and NOCLEAR on page 460.

`FILL(`*value*`)`

Fills the section with the byte pattern *value*. The section must have the `CLEAR` attribute set either explicitly, or by default. If the `CLEAR` attribute is not set, the linker ignores the `FILL` attribute.

`MAX_ENDADDRESS(`*address*`)`

Causes the linker to generate an error if the section extends beyond *address* during final layout.

`MAX_SIZE(`*size*`)`

Causes the linker to generate an error if the section exceeds *size* bytes in length.

`MIN_ENDADDRESS(`*address*`)`

Instructs the linker to pad with zeros, if necessary, to ensure that the section extends to at least *address*.

`MIN_SIZE(`*size*`)`

Instructs the linker to pad with zeros, if necessary, to ensure that the section is at least *size* bytes in length.

`NOCHECKSUM`

Instructs the linker to output the section without a checksum. This overrides the effect of the **-checksum** option for an individual section.

`NOLOAD`

Instructs the linker to output the section as `SHT_NOBITS`. The linker allocates the section into the target image, but discards its contents.

`PAD(`*value*`)`

Places *value* bytes of padding at the beginning of the section. This is equivalent to specifying padding at the beginning of the section contents.

For example, the following two definitions are equivalent:

```
.stack pad(0x10000) : {}
.stack              : { . += 0x10000; }
```

ROM(*sect_name*) CROM(*sect_name*) ROM_NOCOPY(*sect_name*) These attributes allocate the contents of a section named *sect_name* to ROM at link time, while reserving space for the data to be copied to RAM. The ROM and CROM options instruct the linker to put an entry in the real-time ROM copy table and compressed ROM copy table, respectively, so that the Green Hills startup code copies the section from ROM to RAM. CROM compresses its copy by as much as 40% to 50%, and thus can greatly reduce the amount of ROM required for your application. For more information, see “Copying a Section from ROM to RAM at Startup” on page 461. ROM_NOCOPY behaves like the ROM section attribute except that it does not instruct the linker to put an entry in the real-time ROM copy table. As a result, the Green Hills startup code will not copy the section from ROM to RAM. Use this option when you want to use your own startup code for copying a section from ROM to RAM. Uncompressed ROM sections that may be referenced by startup code before being copied should maintain the same order and relative offsets for the ROM copies as their RAM equivalents.
SHFLAGS(*expr*) Sets the sh_flags field of the section's ELF header to the given expression *expr*. For example, to specify that a section should be given the SHF_ALLOC and SHF_WRITE flags, use: SHFLAGS(0x3) For more information, see “Section Attribute Flags” on page 874.

Example 9.7. Setting the Size of the Stack and Heap

This example demonstrates how to set the size and alignment of the `.stack` and `.heap` sections using section attributes:

```
DEFAULTS {
   heap_reserve   = 1M
   stack_reserve  = 512K
}

MEMORY
{
   fast_memory  : ORIGIN = 0x80000000, LENGTH = 10M
   slow_memory  : ORIGIN = 0xbfc00000, LENGTH = 10M
}

SECTIONS
{
   .text                                    : > fast_memory
   .data                                    : > .
   .bss                                     : > .
   .heap       ALIGN(8) PAD(heap_reserve)   : > .
   .stack      ALIGN(8) PAD(stack_reserve)  : > .
}
```

The `.heap` section has two attributes, `ALIGN` and `PAD`. `ALIGN(8)` directs the linker to place `.heap` at an address that is a multiple of 8, regardless of the size of the sections before it. `PAD(heap_reserve)` uses the `heap_reserve` default, and effectively sets the size of the heap to 1 MB.

The `.stack` section has the same two attributes as the heap.

All of the sections following `.text` use the dot (`.`) in place of a memory block, so the linker places them consecutively in `fast_memory` after `.text`. If all of the sections do not fit into `fast_memory`, the linker returns an error.

Example 9.8. Using CLEAR and NOCLEAR

This example shows you how to use the `CLEAR` and `NOCLEAR` attributes to control whether or not the linker puts an entry for a section into the run-time clear table. For more information about the table, and how startup code uses it to initialize sections to zero, see "Run-Time Clear and Copy Tables" on page 476.

The following line specifies the `.bss` section:

```
.bss                            :
```

By default, the linker makes an entry for `.bss` in the run-time clear table. When the program starts, it will initialize all data in this section to zero.

This line describes a section named `.mybss`:

```
.mybss                  CLEAR :{*(COMMON)}
```

`{*(COMMON)}` directs the linker to put COMMON data in this section (see "Using the COMMON Section in Section Inclusion Commands" on page 452). `CLEAR` directs the linker to make an entry in the clear table for `.mybss`. When the program starts, it will initialize data in this section to zero.

You can also use `CLEAR` for the heap:

```
.heap CLEAR PAD(0x100000) :
```

By default, the linker does not make an entry in the run-time clear table for sections other than `.bss`, `.zbss`, and `.sbss` . `CLEAR` overrides this behavior, so the linker makes an entry for `.heap`. `PAD(0x100000)` adds 1 MB of padding to the beginning of `.heap`, effectively setting the size of the heap to 1 MB. When the program starts, it will allocate 1 MB of RAM for this section, and initialize it to zero.

Copying a Section from ROM to RAM at Startup

Use the `ROM` and `CROM` attributes to control whether or not the linker puts an entry for a section into one of the run-time copy tables. Startup code uses these tables to determine which sections to copy to RAM at startup. For more information about these tables, see "Run-Time Clear and Copy Tables" on page 476.

When defining a new section with the `ROM` or `CROM` attributes:

- The name of the new section must be `.ROM.`*`sect`*, where *sect* is the name of the existing section. *sect* must not include any periods (`.`).
- The `ROM` or `CROM` attribute must also specify *sect*.

Do not specify section contents or any other attributes for the new section; the new section inherits the attributes and contents of the existing section, and the existing section becomes a placeholder in RAM. You cannot create multiple `ROM` or `CROM` copies of the same section.

For example, to place the `.data` section in ROM with instructions to have it copied to RAM at startup, include the following lines in your section map:

```
   .data                      : > RAM_memory_block
   ...
// Create a copy of .data in ROM and
// create an entry in the run-time ROM copy table
   .ROM.data      ROM(.data) : > ROM_memory_block
```

Startup code copies the ROM image of `.ROM.data` into the `.data` section, located in RAM.

To place the `.text` section in compressed ROM with instructions to have it copied to RAM at startup, include the following lines in your section map:

```
   .text                      : > RAM_memory_block
   ...
// Create a copy of .text, compress it in ROM, and
// create an entry in the run-time compressed ROM copy table
   .CROM.text    CROM(.text) : > ROM_memory_block
```

Startup code decompresses the compressed ROM image `.CROM.text` and copies the decompressed image into the `.text` section, located in RAM.

If the placement of a section that is not a ROM copy or compressed ROM copy is dependent on the size of a CROM section, you might get a link-time error. Furthermore, relocations should not reference any symbols, including `__ghsbegin_`*`section`* or `__ghsend_`*`section`*, whose locations depend on the size of a CROM section.

You can disable the effects of the CROM attribute by passing the option **-no_crom** on the command line (see "Linker-Specific Options" on page 437). Sections marked as `CROM` are still copied to ROM but are not compressed.

Example 9.9. Placing CROM Sections

The following section map generates errors because the placement of `.myrodata` depends on the size of `.CROM.mydata` and `.CROM.mytext`:

```
.mydata                          : >RAM
.mytext                          :
...
.CROM.mydata   CROM(.mydata)     : >ROM
.CROM.mytext   CROM(.mytext)     :
.myrodata                        :
```

To correct this example, rearrange the sections so that only `CROM.mytext` depends on the size of `.CROM.mydata`:

```
.mydata                        : >RAM
.mytext                        :
...
.myrodata                      : >ROM
.CROM.mydata   CROM(.mydata)   :
.CROM.mytext   CROM(.mytext)   :
```

Modifying your Section Map for Speed or Size

The **Project Wizard** (see the documentation about creating a project in the *MULTI: Managing Projects and Configuring the IDE* book) provides a variety of default linker directives files with section maps that either maximize the speed or minimize the size of your program by varying the number of sections copied to RAM at startup. You can control this feature by manually editing your section map.

Maximizing Execution Speed

Because access times for RAM are better than for ROM on most hardware, your program will generally execute faster if all of its sections are copied from ROM to RAM at startup. The Green Hills startup code copies from ROM to RAM those sections identified with the `ROM` attribute (see "Copying a Section from ROM to RAM at Startup" on page 461).

If you want to use custom code instead of the Green Hills startup code to copy sections from ROM to RAM, see "Customizing the Run-Time Environment Program Sections" on page 464 and "Run-Time Clear and Copy Tables" on page 476.

Minimizing RAM Usage

To minimize RAM usage, you must ensure that as few program sections as possible are copied to RAM at startup.

If a section does not change at run time, you do not need to copy it to RAM. Although you generally must copy `.data` to RAM, the Green Hills V850 and RH850 compiler can determine whether certain data items do not change at run time, and so can be retained in ROM. In order to take advantage of this capability, create a section called `.rodata` . The linker places user-defined `const` variables into the `.rodata` section. When you declare a variable with the keyword `const`, the value becomes a constant, ensuring that the linker places it in the `.rodata` section.

The `.rodata` section should be placed in the link map adjacent to the program `.text` section, causing this data to be put into ROM along with the program text.

If you use the SDA optimization (see “Special Data Area Optimizations” on page 87), some data that would normally go into the `.rodata` section might be placed in the `.rosdata` section.

If you use the ZDA optimization (see “Special Data Area Optimizations” on page 87), some data that would normally go into the `.rodata` section might be placed in the `.rozdata` section.

If you use the TDA optimization (see “V850 Tiny Data Area (TDA) Optimization” on page 96), some data that would normally go into the `.rodata` section might be placed in the `.tdata` section.

Customizing the Run-Time Environment Program Sections

The following material describes the special program sections that are created for and maintained by the Green Hills run-time environment system (the libraries **libsys.a** and **libstartup.a** and the module **crt0.o**). These sections appear in all the default linker directives files provided with your distribution, and their contents are generated automatically, so you should not explicitly add to them.

Note

Any attempt to explicitly place text or data into the following sections is likely to produce unexpected results. When creating custom-named sections, you must take care not to use any of the names of these special sections. For more detailed descriptions of the functions that use these linker directives and special sections to set up the run-time environment, see “Customizing the Run-Time Environment Libraries and Object Modules” on page 813. You can also look at the comments in the customizable source files, which are located in the **src/libsys** and **src/libstartup** directories.

.fixaddr and .fixtype

The compiler creates `.fixaddr` and `.fixtype`.

These two sections contain information that enables the Green Hills startup code to relocate PIC/PID initializers of data variables. The compiler generates data in the `.fixaddr` and `.fixtype` sections, if needed, when using PIC and/or PID. The

default Green Hills run-time libraries might already have information in these sections, because many of these libraries are always built with PIC and PID.

These two sections contain read-only data and can be placed in ROM. Failure to include these sections in the linker directives file might cause the linker to add them to the end of the section list, which might cause the following warning:

```
[elxr] warning: section .fixaddr from libsys.a(ind_crt1.o)
isn't included by the section map;
  appending after last section.
  add to section map or use -append to append without warning
```

If your program has dynamic memory that expands past the end of the last specified section in a section map, it might be fatal to append sections to the end of the section map, because those sections might be overwritten. Therefore, you should include these sections in section maps.

.heap

The linker creates the `.heap` section with the information from your section map. It specifies the size and location of the run-time heap. This section is required when using the Green Hills run-time libraries for dynamic memory allocation. A reference to the special linker symbol `__ghsbegin_heap`, which denotes the beginning of the `.heap` section, is located in the **ind_heap.c** module of the **libsys.a** library.

If you do not include a `.heap` section in your linker directives file, and your program pulls in `ind_heap.o`, you will get one or more linker errors, such as the following:

```
[elxr] (error) unresolved symbols: 1
 '__ghsbegin_heap' from libsys.a(ind_heap.o)
```

If you are not using the Green Hills run-time libraries, you can omit the `.heap` section from your linker directives file.

The Green Hills run-time libraries ensure that dynamic memory allocation does not overrun the specified `.heap` area. The libraries do so by returning an error code from `sbrk()` that causes `malloc()` and other memory allocation routines to return a `NULL` pointer. Your code should always check for erroneous return values when calling dynamic memory allocation routines. You can customize the Green Hills `sbrk()` routine, located in **ind_heap.c**, to avoid using the `.heap` section.

.rodata

The compiler creates the `.rodata` section. Some Green Hills compilers place read-only data (such as data declared with the C `const` specifier and string constants) into a read-only section called `.rodata`. By convention, you place `.rodata` with other sections that can be placed in ROM. Failure to include a `.rodata` section in the section map may cause the linker to append it to the end of the section map. If you compile your program with either **-pic** or **-pid**, it does not use this section.

ROM Sections

When the linker encounters a section directive that has the `ROM` attribute, it creates a read-only copy of the section. For debugging purposes, a section that contains the `ROM` attribute should be named `.ROM.`*`sect`*, where *.sect* is the name of the section that is copied from ROM into RAM (see "Copying a Section from ROM to RAM at Startup" on page 461). The Green Hills startup code, **ind_crt0.c** (see "ind_crt0.c" on page 817), copies the data from the `.ROM.`*`sect`* section to the *.sect* section when copying initialized data from ROM to RAM.

If you want to use your own custom code to copy a section from ROM to RAM at program startup, use the `ROM_NOCOPY` attribute when defining the ROM section. When you use this attribute, the linker will not create an entry in the copy table, and the Green Hills startup code will not copy the section from ROM to RAM. This attribute behaves similarly to the `ROM` and `CROM` attributes.

If the custom code uses the copy table, use the `ROM` attribute. Then, disable the Green Hills code that copies sections from ROM to RAM by modifying or deleting code in **ind_crt0.c**.

If you do not need to copy sections from ROM to RAM, do not include `.ROM.`*`sect`* sections or use the `ROM`, `CROM`, or `ROM_NOCOPY` attributes in your linker directives file.

.sdabase

For completeness, all section maps should include `.sdabase`, regardless of whether or not the target supports the SDA optimization.

If SDA is implemented, the `.sdabase` section is required. In this case, place the `.sdabase` section just prior to the start of the sections that make up the SDA. The Green Hills debug servers initialize the SDA base register with the address of `.sdabase` section (a dummy, zero-sized section). The Green Hills startup code initializes the SDA base register by referencing the predefined symbol `__ghsbegin_sdabase` to obtain the location of the `.sdabase` section.

.secinfo

`.secinfo` is a special section output by the Green Hills linker that contains information about the section layout of programs. The startup code, **ind_crt0.c**, uses this information to determine which sections must be cleared (for example, `.bss` sections) and which sections must be copied (for example, ROM sections) at program startup. This section is read-only and thus can be placed in ROM. Failure to include this section in the link map causes the linker to append it to the end of the section map. The following three section flags control the information placed in the `.secinfo` section:

- `a` — allocated for downloading
- `w` — writable
- `x` — executable

For example:

```
.section ".foo", "aw"
.section ".bar", "ax"
```

Consult the extensive comments in the **ind_crt0.c** file for further documentation on the automatic copy/clear feature.

The linker allows a program to obtain the locations, sizes, and types of its sections at run time. If the symbol `__secinfo` is referenced but not defined in a program, the linker fills in additional section information in the `.secinfo` section and sets the `__secinfo` symbol to point to it. The **indsecinfo.h** header file in the **src/libsys** directory of your distribution has a declaration for this structure and a sample program that shows its use.

.stack

.stack specifies your intended stack area and size, although there is no general mechanism for ensuring that the stack will stay within the limits specified by the .stack section. Green Hills startup code (when running in stand-alone mode) and debug servers initialize the stack pointer of a process based on this .stack section. When building programs to run under an operating system that does not allow user-specification of the stack area, an empty .stack section can still be included in the section map to resolve references in the startup code. The reference to the .stack area in the Green Hills startup code (**crt0.o**) is used only when programs are in stand-alone mode (that is, when they are not downloaded through the MULTI Debugger). Use of the .stack section in the startup code can also be customized by editing and rebuilding the **crt0.800** assembly module. The Debugger also allows the initial stack pointer to be specified differently with each download of a program via the special _INIT_SP variable; use of _INIT_SP supersedes use of a .stack section to locate the initial stack pointer.

.syscall

.syscall contains the run-time library code for Green Hills emulation of system calls. This section must be located in RAM if you use the Green Hills run-time libraries for system call emulation. When system call emulation is not being handled in this way, you can modify **ind_dots.800** to remove its dependency on the .syscall section (see "ind_call.800, ind_dots.800" on page 819).

Symbol Definitions

The linker recognizes several symbols that, if referenced but not defined in the source code, are given particular addresses corresponding to the final image.

Beginning, End, and Size of Section Symbols

When the linker performs final symbol resolution for a fully linked output file, it recognizes that certain undefined symbols refer to memory addresses in the final section map. The names of these symbols are __ghsbegin*section*, __ghsend*section*, and __ghssize*section*, where *section* is the name of each section in the output file, with any dot (.) characters changed to underscores ("_").

For example, for a section named `.text`, the symbols `__ghsbegin_text` and `__ghsend_text` resolve to the absolute addresses of the beginning and end of the section, while `__ghssize_text` refers to its size.

For a section map containing these lines:

```
SECTIONS
{
    .bss1  0x400000:
    .bss:
}
```

and this portion of **main.c** program code:

```
extern char __ghsbegin_bss1[], __ghssize_bss1[];

void foo() {
    memset(__ghsbegin_bss1, 0, (size_t)__ghssize_bss1);
    __ghsbegin_bss[0] = 0xff;
}
```

if the size of section `.bss1` is `0x100`, then the linker resolves the following labels:

`__ghsbegin_bss1` to `0x400000`

`__ghsend_bss1` to `0x400100`

`__ghssize_bss1` to `0x100`

`__ghsbegin_bss` to `0x400100`

End of Function Symbols

When the linker performs final symbol resolution for a fully linked output file, it recognizes that symbols named `__ghs_eofn_`*`function`*, where *function* is a function name, refer to the memory addresses of the end of functions in the final executable. For example, for function `foo`, the symbol `__ghs_eofn_foo` resolves to the virtual address of the end of that function.

For example:

```
extern void __ghs_eofn_foo(void);
int foo(int x) { return x+1; }
int main() {
    printf("Size of foo is %d\n",
```

```
        (int)__ghs_eofn_foo - (int)foo);
    return 0;
}
```

If the function `foo` is at address `0x100000` and has size `0x20` bytes, then the linker resolves `__ghs_eofn_foo` to be `0x100020`.

Linker Generated Tables

When the linker performs final symbol resolution for a fully linked output file, it recognizes undefined symbols named `__ghstable_`*`tablename`* as tables. When it encounters one of these symbols, the linker resolves the symbol to a table that it allocates to the read-only `.rodata` section. The linker fills the table with the addresses of all defined symbols named `__ghsentry_`*`tablename_entryname`*, and then terminates it with a NULL pointer. The order of the entries within the table is unpredictable. If the name of an entry symbol matches multiple tables that the linker is filling, the entry is added to the table with the longest matching name.

Note

`__ghstable_`*`tablename`* is not supported with Position Independent Code (PIC) or Position Independent Data (PID).

In the following example, `__ghstable_resetvectors` is an array of pointers to functions. The array contains the addresses of `__ghsentry_resetvectors_foo()` and `__ghsentry_resetvectors_bar()`. The function `reset_all()` has the effect of calling both functions.

```
extern void (* const __ghstable_resetvectors[])(void);

int foo = 5;
void __ghsentry_resetvectors_foo(void)
{
    foo = 0;
}

double bar = 2.0;
void __ghsentry_resetvectors_bar(void)
{
    bar = 0.0;
}
```

```
void reset_all()
{
    int i;
    for (i = 0; __ghstable_resetvectors[i] != 0; ++i) {
        (__ghstable_resetvectors[i])();
    }
}
```

Forcing The Linker to Pull In a Module From a Library

Normally, the linker pulls in an object file from a library only when that file provides a definition for an unresolved symbol that is not weak. You can use the **-extractall** and **-extractweak** options to change this behavior globally. However, there might be cases when you want the linker to pull in particular object files from a library when that library is added on the command line, but you do not want the linker to pull in all of the object files from that library.

The linker searches all libraries added on the command line for symbols beginning with `__ghsalwaysimport` or `__ghsautoimport`. If an object file in a library defines a symbol whose name begins with `__ghsalwaysimport`, the linker will always pull in that object file. If an object file defines a symbol whose name begins with `__ghsautoimport`, the linker will pull in that object file unless the symbol has already been defined by another object file. The distinction between `__ghsalwaysimport` and `__ghsautoimport` may be useful when dealing with common symbols (see "Allocation of Uninitialized Global Variables" in "C/C++ Data Allocation" on page 200 for more information about the **--commons** option). That is, if multiple modules define a common symbol named `__ghsalwaysimport_foo`, the linker will pull them all in, but if multiple modules define a common symbol named `__ghsautoimport_foo`, the linker will only pull in the first such module it encounters.

Deleting Unused Functions

The **-delete** option instructs the linker to remove functions that are not referenced in the final executable. The linker iterates to find functions that do not have relocations pointing to them and eliminates them. If a symbol is referenced by a relocation in the first pass, it can still be eliminated in a subsequent pass if the referencing symbol is eliminated. Note that the **-delete** option can increase the time and memory required for linking.

Note

If you are hand-coding assembly and intend to use the **-delete** option, you must properly identify assembly labels as functions or objects with the proper size.

The linker supports **-delete** only for object files produced by the Green Hills assembler. Other assemblers might produce object files that do not preserve information required for **-delete** to function properly.

Tip

In cases where a symbol should never be deleted, use the **-u** driver option, **-keep=*funcname*** linker option, or the `.need` assembler directive.

The `.need` assembler directive introduces an explicit dependency between two symbols (usually functions) so that the linker keeps one function if another function is used. For example, use this directive when you have an assembly function that does not return, but instead falls through into the next function in the file.

The linker never deletes symbols preserved by these methods, even if the symbols appear unused. It is useful to preserve symbols such as interrupt table entry points, non-standard entry-points, or other portions of code not reached by normal control transfer within the program.

If your code is written in C++, the compiler mangles the names of some symbols, and the names passed to these options must match the mangled names. To find the mangled names, pass the options **--no_link_filter -lnk=-v** and the linker will list them in its output (see “Demangling of Names in Linker Messages” on page 284).

Compiler-generated profiling instrumentation or debugging information might create references to unused functions, preventing the linker from deleting them. See **-ignore_debug_references** in “Link-Time Ignore Debug References” on page 226 for more information.

Advanced Linker Features

Code Factoring

Code factoring is a link-time optimization that reduces overall program size by removing redundant segments of code from object files. It removes these segments by using subroutine calls and tail merges. This optimization is useful for reducing program size, but may decrease program speed by increasing the number of branches. The standard libraries provided with the Green Hills tools for V850 and RH850 are all prepared for code factoring.

To enable code factoring:

Set the **Linker→Linker Optimizations→Code Factoring** option to **On** (**-codefactor**).

Code factoring does not work with libraries by themselves, but does affect the size on an executable that links with the library.

Note

Code factoring might cause DWARF debugging information to be out of sync with the program in memory.

Example 9.10. Basic Code Factoring

To factor code from source files **foo.c** and **bar.c**, use:

```
ccv850 -codefactor -o hello foo.c bar.c
```

This command generates two object files, **foo.o** and **bar.o**. They are prepared for code factoring, and then the driver passes them to the **elxr** linker, which performs the optimization and links them into an executable, **hello**.

Example 9.11. Controlling the Scope of Code Factoring

The linker performs code factoring on a per-section basis. For example, if you have the following in your section map:

```
SECTIONS {
  .footext : { foo1.o(.text) foo2.o(.text) }
  .bartext : { bar1.o(.text) bar2.o(.text) }
}
```

- The linker factors code from the `.text` sections of both `foo1.o` and `foo2.o`, and places the optimized code into the `.footext` section.
- The linker factors code from the `.text` sections of both `bar1.o` and `bar2.o`, and places the optimized code into the `.bartext` section.

This behavior can be useful when you want to prevent code factoring from creating dependencies between sections that must remain independent.

Example 9.12. Code Factoring and Incremental Linking

Code factoring runs during the final stage of linking, when the linker produces an executable file. If you incrementally link your files before producing an executable, pass the `-codefactor` option at each stage to ensure that the linker preserves the necessary relocation and symbol information. For example, to generate a relocatable object file **mod.o** from object files **foo.c** and **bar.c**, enter the following:

```
ccv850 -codefactor -relobj -o mod.o foo.c bar.c
```

This command generates the relocatable object file **mod.o**, and preserves all symbol and relocation information. To subsequently generate a code factored executable, enter the following:

```
ccv850 -codefactor mod.o -o hello
```

This command produces the code-factored executable, **hello**.

Example 9.13. Partial Code Factoring

It is possible to apply code factoring to only some of the modules that are linked into the final executable. If **foo.c**, **bar.c**, and **baz.c** are to be linked together to produce the executable **hello**, but only **bar.c** and **baz.c** are to be code factored, first generate an object file from **foo.c**, as follows:

```
ccv850 -c foo.c
```

This command produces an object file named **foo.o**, which is excluded from code factoring in any subsequent link.

Next, generate object files from **bar.c** and **baz.c**, using the `-codefactor` option, as follows:

```
ccv850 -codefactor -c bar.c baz.c
```

This command produces the object files **bar.o** and **baz.o**, which are prepared for subsequent code factoring.

Finally, generate an executable, as follows:

```
ccv850 -codefactor foo.o bar.o baz.o -o hello
```

This command produces an executable, **hello**, in which code factoring is applied only to **bar.o** and **baz.o**.

Note

Please note the following:

- Hand-written assembly code files and mixed assembly and C/C++ source files (that is, C or C++ source files with `#pragma asm`, `asm` macros, etc.) cannot be code factored by default. For optimal results, do not mix both assembly and C/C++ code within the same source file. If code factoring of mixed assembly and C/C++ source files is necessary, see `#pragma ghs safeasm` and `#pragma ghs optasm` in "Green Hills Extension Pragma Directives" on page 753.
- When using the MULTI Debugger, you cannot set breakpoints on lines of source code that have been altered or removed by code factoring. These lines of code display a red breakdot instead of a green breakdot. Switch to interlaced source and assembly mode, and then set breakpoints at the assembly level.
- To control code factoring in the linker, the compiler generates a number of `.ghsnote` assembler directives (which reside in the non-allocated notes section `.ghsinfo`, and have no impact on final code size). You should not modify these directives or the contents of this section.

Run-Time Clear and Copy Tables

These three tables are contained within the `.secinfo` section.

- The *clear* table is bounded by the symbols `__ghsbinfo_clear` and `__ghseinfo_clear`.
- The *copy* table is bounded by the symbols `__ghsbinfo_copy` and `__ghseinfo_copy`.
- The *compressed copy* table is bounded by the symbols `__ghsbinfo_comcopy` and `__ghseinfo_comcopy`.

These tables each contain zero or more records detailing the action to be taken at startup.

By default, the Green Hills C run-time library **libstartup.a** automatically clears `.bss`, and copies `ROM` and decompressed `CROM` sections to RAM based on this table. The actual implementation of the above three routines for your CRT code can be found in **src/libstartup/ind_crt0.c** and **src/libstartup/ind_uzip.c** in your Green Hills product and differs slightly for PIC/PID support on some targets.

Chapter 10

The ax Librarian

Contents

The librarian combines object files created by the assembler or linker into a library (or archive file). By convention, libraries have the **.a** extension. At link time, the linker can search through libraries for object files, and pull them in to provide definitions for undefined symbols.

Creating a Library from the Compiler Driver

To create a library using the compiler driver, use the following syntax:

ccv850 *filename ...* -archive -o *libname*.a

where each *filename* is a source file or object file that you want to include in the library, and ***libname*.a** is the name of the library you want the librarian to create. The compiler driver compiles and assembles the source files, and then runs the librarian to create a library from the resulting object files.

If ***libname*.a** already exists, the librarian inserts the new object files into the existing library. If the existing library contains an object file that has the same name as one of the new object files, the librarian overwrites the existing object file with the new one.

Use the **–update_archive** option to force the librarian to delete all leftover object files from the library. The builder uses this option when creating and updating libraries. This option is ignored with the option **-merge_archive**.

To create a *merged library* – a library that merges together several existing libraries and/or object files – use the following syntax:

ccv850 *filename ... input_library.a ...* -merge_archive -o *output_library*.a

-merge_archive is similar to **-archive**, with the following differences:

- If ***output_library*.a** exists, it is replaced.
- The entire contents of each specified ***input_library*.a** is added to the library (libraries specified with **-l*file*** are ignored). The contents of each input debug library (**.dba**) are added to the output debug library.
- Any input archives are passed to the prelinker after `--`. Templates that are instantiated in the archive may be used, but may not be rebuilt.

-merge_archive is implemented by passing the **C** and **M** command modifiers to the archiver (see "Librarian Command Modifiers" on page 481).

If you need a 64-bit format archive, pass the **-large_archive** option.

Example 10.1. How to Create a Library

To create a library **libfoo.a** from the source file **foo.c**, enter:

```
ccv850 foo.c -archive -o libfoo.a
```

To create a library **libhello.a** from the object files **hello.o** and **world.o**, enter:

```
ccv850 hello.o world.o -archive -o libhello.a
```

Modifying Libraries

You can modify libraries (**.a** files) by invoking the **ax** librarian directly, as follows: **ax** *-command* [*modifier*]... [*-option*]... *libraryfile* ... [*memberfile*]...

where:

- *-command* is one of the librarian commands
- each *modifier* is a command modifier
- each *-option* is a librarian option
- ***libraryfile*** is the name of the file you want to modify, including the extension.
- each ***memberfile*** is the name of the file (usually an object file) in the library you want to perform the command on, including the extension.

The following tables list valid command letters, modifiers, and options.

Librarian Commands

This table lists each librarian command, including the modifiers that you can use with it. The librarian command modifiers are explained in the subsequent table, "Librarian Command Modifiers" on page 481. Enter the command and modifiers as one string without spaces.

-d [e] [S] [v] *libraryfile memberfile ...* Deletes each specified *memberfile* from the specified *libraryfile*.
-p [e] *libraryfile memberfile ...* Prints each specified *memberfile* from the specified *libraryfile* to standard output (if the members are object files, then the output might not be human-readable).
-r [c] [C] [e] [S] [n] [m] [M] [v] [u] *libraryfile memberfile ...* Replaces each specified *memberfile* in the specified *libraryfile*. The new version of *memberfile* overwrites the existing version. If *memberfile* does not already exist in *libraryfile*, the librarian adds it. If *libraryfile* does not exist, the librarian creates it. We recommend that you create libraries with the compiler driver and not with the librarian, because only the compiler driver generates the **.dba** files necessary for subsequent debugging.
-t [e] [s] [v] *libraryfile ...* Lists the names of the files contained in each specified *libraryfile*. Use the **v** option to list names, file sizes, and the dates that the members were last modified.
-x [e] [v] [o] *libraryfile memberfile ...* Extracts each specified *memberfile* from the specified *libraryfile* into the current directory. The librarian does not alter *libraryfile*. If you only specify *libraryfile*, the librarian extracts all member files.

Librarian Command Modifiers

These modifiers can be appended to the librarian command letters to give you greater control over operations on library files. See "Librarian Commands" on page 479 to determine which modifiers are available with each command letter.

c Suppresses warnings when creating a library that did not exist.
C Replaces the existing library.
e Prefixes messages with `ERROR` or `WARNING`. Equivalent to the compiler driver option **-prefixed_msgs**.
n **m** **M** These options control how the librarian handles input files that are libraries. They are mutually exclusive. • **n** — [default] The librarian treats all input files as complete units and does not consider that the input file might be a collection of files. Each input library is placed in the output library as one complete unit. • **m** — The librarian places each member of each input library into the output library as a separate file. • **M** — Similar to **m**, but the names of the files are mangled by prepending the base name of the archive to each file. The full path and file extension are not used in this mangling scheme. This modifier reduces the likelihood that the archive will contain two files with the same name.
o Keeps original timestamps when extracting files from a library.
s Regenerates the table of contents; use with the **-t** command.
S Prevents the creation of a table of contents.
u Replaces files only if the timestamp is newer (use with **-r**).

v

Prints verbose messages when executing a command.

Librarian Options

The following options are recognized on the command line before the name of the library:

-display_toc Used with the **t** command letter to display the table of contents and the offset of each member file within the archive.
-pecoff When generating the table of contents, this option also generates a Microsoft Windows compatible table of contents.
-stderr=*errfile* Redirects error output of the librarian to the named file.
-tmp=*dir* Specifies the temporary directory.
-auto_large_archive The archiver detects when to output a 64-bit archive and prints a warning when it does. If the archiver takes a 64-bit archive as input but detects that it does not need a 64-bit output file, it prints a warning and does not use the 64-bit format.
-quiet_auto_large_archive The archiver detects when to output a 64-bit archive. If the archiver takes a 64-bit archive as input but detects that it does not need a 64-bit output file, it does not use the 64-bit format. The archiver does not print any warnings related to 64-bit archives.
-large_archive The archiver always uses 64-bit archives. The archiver does not print any warnings related to 64-bit archives.
-no_large_archive The archiver never uses 64-bit archives. The archiver does not print any warnings related to 64-bit archives, but it does print an error if it needs to truncate any values because the archive format does not provide enough space.

Examples

To add a new object file **foo.o** to an existing library **lib.a**, enter:

```
ax -r lib.a foo.o
```

To delete object files **foo.o** and **bar.o** from library **lib.a**, enter:

```
ax -d lib.a foo.o bar.o
```

To extract object file **foo.o** from library **lib.a** and rename it to **newfoo.o**, enter:

```
ax -p lib.a foo.o > newfoo.o
```

To print the table of contents of library **lib.a** in verbose mode, enter:

```
ax -tv lib.a
```

Creating and Updating a Table of Contents

The librarian creates a table of contents, by default, whenever it creates or modifies a library. This improves the linker's performance, because the linker does not have to create the table of contents every time it reads a library at link time. The librarian stores the table of contents in a hidden member file named / (slash), which is always the first file in the library.

To subsequently create or update a table of contents without altering the library, use the **-t** command with the **s** option. For example, if you want to create a table of contents for **lib.a**, enter:

```
ax -ts lib.a
```

The Green Hills 64-Bit Archive Format

The **ax** archiver provides an extended archive format that allows archives to exceed 2 GB in size. The archiver attempts to detect that the extended format is needed based on its input files. If the archiver detects that the extended format is necessary, it prints the following message:

```
ax:warning: Total size of archive exceeds 2GB limit, switching to large
archive format. Pass -large_archive to suppress this message, or
-no_large_archive to force normal archive output.
```

If the archiver downgrades input files from the 64-bit archive format, it prints the following message:

```
ax: warning: Input archive is large-format, but large format is not necessary.
Switching to normal-format archive. Use -large_archive to force large output,
or -no_large_archive to suppress this message.
```

For information about controlling 64-bit archive format selection and warnings, see "Output Archives in 64-Bit Format" on page 287.

Note

Because the large archive format is specific to Green Hills Tools, archives in this format are not compatible with third-party linkers or other third-party utilities.

Chapter 11

Utility Programs

Contents

The MULTI IDE includes many useful command line utility programs, including functional replacements for the standard Linux/Solaris utilities **dump**, **file**, **hide**, **nm**, **size**, **strip**, and **what**. All utility programs work with files generated by any Green Hills development tools.

- **gaddr2line**: Similar to the Linux/Solaris **addr2line** program. Maps addresses to filenames and line numbers.
- **gasmlist**: Generates interlaced assembly and source output (object files only).
- **gbin2c**: Converts binary files into C array definitions.
- **gcores**: Combines multiple images into a multi-core archive
- **gbincmp**: Compares two binary files (single object files or executables only).
- **gbuild**: A command line interface to the MULTI Builder.
- **gcolor**: Takes text input and colors it using ANSI escape sequences.
- **gcompare**: Compares space or time performance.
- **gdump**: Similar to the Linux/Solaris **dump** program. Dumps or disassembles a file.
- **gfile**: Similar to the Linux/Solaris **file** program. Describes the file type.
- **gfunsize**: Returns the code size of a function.
- **ghide**: Similar to the Linux/Solaris **hide** program. Hides global symbols in an object file.
- **gmemfile**: Converts an executable file to a binary image suitable for loading.
- **gnm**: Similar to the Linux/Solaris **nm** program. Displays file information.
- **gpatch**: Installs and creates simple patches. Use this utility to install patches provided by Green Hills support.
- **gpjmodify**: Performs command line editing of **.gpj** project files.
- **gsize**: Similar to the Linux/Solaris **size** program. Displays section sizes.
- **gsrec**: Converts executable files or a multi-core archive into a Motorola S-Record, Intel hexadecimal, or Tektronix hexadecimal format file.
- **gstack**: (for executable files) Computes stack size for each task.
- **gstrip**: Similar to the Linux/Solaris **strip** program. Removes symbol or debugging information from an executable.
- **gversion**: (for executable files) Returns version date and time information.

- **gwhat**: Similar to the Linux/Solaris **what** program. Reports or updates version information.
- **mevundump**: Converts (and merges, if multiple input files given) an input event log of any format to **mevgui**'s event log format.
- **ccorti** and **osa_orti.so**: Provides operating systems awareness (OSA) for ORTI-enabled operating systems such as many AUTOSAR and OSEK operating systems.
- **protrans**: Gathers profiling data for use with MULTI.

The gaddr2line Utility Program

The **gaddr2line** utility program displays the source filename and line number for instruction addresses.

The syntax for this utility is as follows:

gaddr2line [*options*] *addresses*

where *addresses* is a whitespace-separated list of instruction addresses and *options* are listed in the following table:

-f Display function names along with filename and line number.
-e *filename* Specifies the executable filename. The default is **a.out**.
-r Display procedure relative line numbers instead of file relative line numbers.
-s Display just the base filename, instead of the full path and filename.

The **gaddr2line** utility looks up each specified address in the debug information for the program and prints the source filename and line number that correspond to the address. The information is displayed as `filename:line number`. If there is no debug information, or the address does not correspond to a valid source line, then `??:0` is displayed. When the **-f** option is passed, the function name is displayed on the line preceding the source line information, and `??` is displayed for unknown functions. An example is shown below:

```
gaddr2line -e hello.out -f  0x10110 0x10114 0x99999999
main
hello.c:2
main
hello.c:4
??
??:0
```

The gasmlist Utility Program

The **gasmlist** utility program displays interlaced assembly and source for the specified object files.

The syntax for this utility is as follows:

gasmlist [*options*] *files*

where *files* is a whitespace-separated list of files and *options* are listed in the following table:

--all **--all_with_asm** Controls which files have source lines printed: • **--all** prints source lines for all files • **--all_with_asm** prints source lines for all files that produce assembly or, if **--src_then_asm** is specified, for all files. The default is to restrict output to the object file's base source file (so only source lines from the base source file are printed).
--file_num **--proc_num** Restricts identification of output lines to one of the following: • file line numbers • procedure line numbers The default is to print both procedure and file line numbers.
--file_offset Calculates offsets from the start of the present object file. The default is to calculate them from the start of the present section.
--help Prints usage information.
--hide_all_labels Suppresses the printing of detailed label information.

--hide_mangled

Suppresses the printing of mangled versions of labels. The demangled versions are always printed.

-I *directory*

Specifies an additional *directory* to search for source files.

The following directories are searched for source files in this order:

- the relative or absolute path specified in the object file
- directories specified using **-I *directory***
- the user's path

The current directory is not searched unless it is specified in one of these categories.

--max_header_lines *n*

Specifies the maximum number *n* of header source lines to be printed.

The default value for *n* is 4. Setting *n* to 0 instructs **gasmlist** to print all associated header file lines.

--no_prefix_file

Suppresses the printing of the filename at the beginning of each source line.

--omit_headings

Suppresses the printing of headings to separate files and sections.

--proc_num_on_left

When both file and procedure line numbers are enabled, prints the procedure numbers on the left. The default is to print them on the right.

--root_remap *rt dir*

Re-maps the original path *rt* to the new path *dir*. For example to map **a/b/c** to **d/c**, pass:

```
--root_remap a/b d
```

--show_func_labels

Prints line offsets with the label they are offset from. For instance, `+ 0x4` would instead be printed `label + 0x4`.

--src_then_asm

Orients the output primarily towards source code. The default is to emphasize assembly.

The gcores Utility Program

The **gcores** utility combines one or more core images and an optional shared section into a single archive. This single file archive provides linked images and metadata suitable for loading onto and debugging a multi-core target. It can be manipulated with the GHS archiver (**ax**) and the **gdump** utility. Individual images can be identified in the **gdump** output by looking for `multi-core(image)` where image is the name of the linked image.

The syntax for this utility is:

gcores [*options* -core *core1_specs* -core *core2_specs* -share *shared_spec* -common *common_options*] where the options, core specifications and shared specifications are described below.

Options available only in the command line:

-h Print usage
-p Show progress. This is the default.
-q Run in quiet mode. Do not show any progress.
-core { *core_or_shared_spec* } Specify a linked core image to include in the multi-core archive. Within the braces, a relocatable image, objects to include in the shared module and linker options may be specified. At least one core must be specified.
-share { *core_or_shared_spec* } Specify a shared module for the multi-core archive. This is an optional argument, and only one shared specification is allowed. Within the braces, a relocatable image, objects to include in the shared module and linker options may be specified.

Options available in the command line and in a MultiCoreArchive **.gpj** project file:

-common See the Builder documentation for “Gcores Common Link Options” on page 246.
-driver See the Builder documentation for “GCores Driver” on page 246.

-keeptempfiles See the Builder documentation for “Keep GCores Temporary Files” on page 247.
-cpu See the Builder documentation for “GCores Cpu” on page 247.
-o See the Builder documentation for “Gcores Output Filename” on page 245.
-shared_imports See the Builder documentation for “Gcores Shared Modules Import Symbols From Cores” on page 245.
-cross_core_imports See the Builder documentation for “GCores Cores Import Symbols From Other Cores” on page 246.
-exported_absolutes_only See the Builder documentation for “Gcores Exported Absolutes Only” on page 247.
-srec[=*filename*] See the Builder documentation for “GCores Generate Additional Output” on page 248.
-hex[=*filename*] See the Builder documentation for “GCores Generate Additional Output” on page 248.

Note

Users should take care when constructing multi-core archives to ensure that project components in a core or shared module only reference library routines and compiler functionality (that may depend on library routines) that is available to code executing in that module. For example, if all C libraries are per-core, then features that depend on these libraries such as run-time error checking may cause the shared module to fail linking with unresolved symbol errors. Specifically, the symbols that exist in those libraries will by default not be available to the shared module since they are not themselves in the shared module.

Note

When linking a program, the linker will match unresolved symbols in the core-specific programs to exported symbols in the shared module last, after exhausting attempts to resolve the symbols elsewhere.

Example 11.1. Using gcores command line

```
gcores -shared_imports -cross_core_imports -exported_absolutes_only
-o HyperthreadsDemo.mca
-share { -o shared.so ./shared.rel -extra_file=shared.ld }
-core { -o core0 core0.rel -extra_file=core00.ld  }
-core { -o core1 core1.rel -extra_file=core11.ld  }
```

A final image **HyperthreadsDemo.mca** is generated. It has a shared module and two cores. Each core and the shared module are built out of a relocatable image, and specify their linker directives file by providing a linker directives file (**.ld**) with the **-extra_file** option in the specification.

The optional options **-shared_imports**, **-cross_core_imports** and **-exported_absolutes_only** are provided.

Example 11.2. Using gcores from MULTI

gcores can be called from a command line, as explained in the previous paragraph, or automatically from MULTI as part of a multi-core build.

To achieve that, a **.gpj** of the type MultiCoreArchive can be created.

A MultiCoreArchive project file can include options that will be provided to gcores.

- A SharedMemory project file defines a fully linked program that will be shared across all cores.

- A Core project file defines a fully linked program that represents a Core or a Thread.

Driver options can be provided in the SharedMemory or Core projects (or in the top most project if these options are wanted to be common). These options will be used in the compile and initial linker stage (in order to produce the relocatable object that will be passed automatically as input to **gcores**). If a link option is needed in the final linking stage, where the relocatable objects are resolved, this has to be provided as a **-common { option1 [option2] ... [optionN] }** in the MultiCoreArchive . The SharedMemory project is optional, and only one can be provided, but at least one Core project should be included.

Each Core or SharedMemory project should be provided its own linker configuration file for proper layout.

MultiCoreArchive example

The following example shows the contents of a MultiCoreArchive project file that defines one SharedMemory and two Cores.

MultiCoreArchive **multicore.gpj**

```
#!gbuild
[MultiCoreArchive]
 -shared_imports
 -cross_core_imports
 -exported_absolutes_only
shared.gpj  [SharedMemory]
hardware_thread0.gpj  [Core]
hardware_thread1.gpj  [Core]
```

SharedMemory project file **shared.gpj**

```
#!gbuild
[SharedMemory]
        -o shared.so
shared.c
shared.ld
```

Core 0 project file **hardware_thread0.gpj**

```
#!gbuild
[Core]
 -o hardware_thread0
hardware_thread0.c
hardware_thread0.ld
```

Core 1 project file **hardware_thread1.gpj**

```
#!gbuild
[Core]
 -o hardware_thread1
hardware_thread1.c
hardware_thread1.ld
```

The gbin2c Utility Program

The **gbin2c** utility converts a group of binary files into a set of array definitions in a source file suitable for inclusion in C or C++ projects. You can control the byte by byte conversion of each binary file into a char array by passing command line arguments.

The syntax for this utility is: **gbin2c** [*global_options*] [-input] *file1* [*file1_local_options*] ...

Command line options for **gbin2c** are divided into three categories.

- *Global options* — Apply to each binary file that you pass to **gbin2c** for conversion. You can specify global options anywhere on the command line.
- *Module inclusion options* — Specify the binary files that you want to convert.
- *Local options* — Apply only to the binary file specified by the nearest preceding module inclusion option.

Global Options

-allnoconst

Removes **const** from a data array type for all modules and their references in the module list.

-compat

Runs **gbin2c** in **gbin2src** compatibility mode.

Implies:

- the **-list** global option
- the **-static** local option for each binary file.

If an additional options file is not specified on the command line (with **-i**), this option also instructs **gbin2c** to read standard input for additional options in lieu of an additional options file.

-crc

Outputs a CRC value for each converted binary file, using the polynomial `0x10211021`. Sample source code for verifying checksums is included in the **src/libstartup/cksum.c** file of your installation.

The declaration of the CRC variable for the array named *name* is: `const unsigned int name_crc;`

-cxx

Replaces `const` with `extern const`. Use this option if you will be including the output in a C++ project.

-ghsentry *tablename*

Outputs an entry for the linker generated table *tablename* for each converted module. Use this option as an alternative to `-list`. For more information, see "Linker Generated Tables" on page 470.

-help

Displays basic usage information.

-i *file*

Reads *file* as a list of additional options.

The contents of *file* are parsed as additional options. Extra whitespace and newline characters are ignored. Options listed in the file have the same effect as options on the command line with the following exceptions:

- The options file is not preprocessed by the user's shell so no special treatment, such as wildcard substitution, is performed.
- The meaning of the **-o** option in the additional options file context differs from its meaning in the command line context. In the options file, **-o *name*** means "refer to the current module as *name* in the module list, and set the array name for the current module to ***name*_data**". In essence, the **-o** option is an alias for the **-output** option when it occurs on the command line and the **-module_name** and **-array_name** options when it occurs in the options file.

Note that only one additional options file can be specified using the **-i** option.

-list

Outputs a list of converted modules.

The default definition for the type of each list element is:

```
struct static_module_info {
 const char * const name;
 const char * const data;
 const struct static_module_info * const next;
};
```

The default declaration of the module list is:

```
const struct static_module_info __static_modules[];
```

-list_name *name*

Sets the name of the module list to *name*.

-list_type *name*

Sets the name of the module list element struct type to *name*.

-nullterm

Terminates each data array with a null character.

-o *file*

-output *file*

Writes output to *file*. See **-i**, above, for the **-o** option's alternate meaning in an options file.

If neither of these options is passed, then output is written to `stdout`.

-size

Outputs the length of each array written.

The declaration of the size variable for the array named *name* is:

```
const unsigned int name_size;
```

-size_array

Outputs an array containing the lengths of all data arrays written when **-list** is specified

-size_field

Adds a `size` field to the module list `struct` type.

-string

Formats output in string literal form instead of comma-separated hex values.

Module Inclusion Options

[-input] *file*

Adds *file* to the list of binary files to convert.

An array is defined for each converted binary file. The default declaration for the array generated from file *file* (stripped of all path information, and with any non-alphanumeric character replaced with underscores) is:

```
const char file_data[];
```

The nature of the array can be controlled via the *local options*.

Note: Non-alphanumeric special characters in the name of *file* are replaced with underscores.

Local Options

These options modify settings for the most recently included module:

-alignment *n*   Sets the alignment of an array for the current module.
-array_name *name*   Sets the array name for the current module to *name*.   By default the array name for a file is the base name of the file with all non-alphanumeric characters replaced with underscores.
-limit *n*   Outputs a maximum of *n* bytes of the current module.
-module_name *name*   Refers to the current module as *name* in the module list.
-noconst   Removes `const` from the type of the array for the current module.
-signed   **-unsigned**   Adds `signed` or `unsigned` to the type of the array for the current module.
-skip *n*   Skips the first *n* bytes of the current module.
-static   Adds `static` to the type of the array for the current module.
-volatile   Adds `volatile` to the type of array for the current module.

Example 11.3. Using gbin2c

```
gbin2c mod.o
```

The contents of **mod.o** are converted to a `const char` array named `mod_o_data` and printed on `stdout`. The output looks similar to the following:

```
/**
 * This file was automatically generated by gbin2c v1.10
 * Green Hills Software [ http://www.ghs.com ]
 * Using command line: gbin2c mod.o
 * Date generated: Fri Jun  17 13:17:13 2011
```

```
 **/

/* mod.o: */

const char mod_o_data[] = {
    0x7F, 0x45, etc.
};
```

Example 11.4. Using the -crc and -size Options

```
gbin2c -crc -size mod.o
```

In addition to the output from example 1, size and CRC information for **mod.o** are displayed:

```
/**
 * This file was automatically generated by gbin2c v1.10
 * Green Hills Software [ http://www.ghs.com ]
 * Using command line: gbin2c -crc -size mod.o
 * Date generated: Fri Jun  17 13:17:13 2011
 * Note: CRC of input file and this array can be found at end of file
 **/

/* mod.o: */

const char mod_o_data[] = {
    0x7F, 0x45, etc.
};

const unsigned int mod_o_data_size = 2620;

const unsigned int mod_o_data_crc = 0xD91D5642;
/* CRC of input file = 0xD91D5642 */
/* Note that file CRC may not match array CRC if -skip or -limit were used */
```

Example 11.5. Using the static Keyword

```
gbin2c -crc -size mod.o -static -skip 64 -limit 4
```

In this example, the `static` keyword is added to the type of `mod_o_data` and only bytes 64-67 (starting counting at 0) of **mod.o** are output:

```
/**
 * This file was automatically generated by gbin2c v1.10
 * Green Hills Software [ http://www.ghs.com ]
 * Using command line: gbin2c -crc -size mod.o -static -skip 64 -limit 4
 * Date generated: Fri Jun  17 13:17:13 2011
 * Note: CRC of input file and this array can be found at end of file
 **/

/* mod.o:  skipped 64 bytes and limited to 4 bytes */
```

```
static const char mod_o_data[] = {
    0x65, 0x2F, 0x73, 0x73
};

const unsigned int mod_o_data_size = 4;

const unsigned int mod_o_data_crc = 0x358E11C6;
/* CRC of input file = 0xD91D5642 */
/* Note that file CRC may not match array CRC if -skip or -limit were used */
```

Example 11.6. Converting Multiple Files

```
gbin2c -crc mod.o -static foo -signed -size -array_name bar
```

In this example, both **mod.o** and **foo** are converted. Notice that the **-crc** and **-size** global options apply to both files while **-static** and **-signed** only apply to **mod.o** and **foo** respectively. This example also demonstrates the use of the **-array_name** option and its effect on the size and CRC variable names.

```
/**
 * This file was automatically generated by gbin2c v1.10
 * Green Hills Software [ http://www.ghs.com ]
 * Using command line:
 *      gbin2c -crc mod.o -static foo -signed -size -array_name bar
 * Date generated: Fri Jun  17 13:17:13 2011
 * Note: CRC of input file and this array can be found at end of file
 **/

/* foo: */

const signed char bar[] = {
    0x66, 0x64, 0x61, 0x73, 0x0A
};

const unsigned int bar_size = 5;

const unsigned int bar_crc = 0x11A9EBE1;
/* CRC of input file = 0x11A9EBE1 */
/* Note that file CRC may not match array CRC if -skip or -limit were used */


/* mod.o: */

static const char mod_o_data[] = {
    0x7F, 0x45, etc.
};

const unsigned int mod_o_data_size = 2620;

const unsigned int mod_o_data_crc = 0xD91D5642;
/* CRC of input file = 0xD91D5642 */
/* Note that file CRC may not match array CRC if -skip or -limit were used */
```

Example 11.7. Outputting the Module List

```
gbin2c  mod.o foo -module_name bar -list
```

In this example the module list is output and the module name for **foo** is changed to `bar`.

```
/**
 * This file was automatically generated by gbin2c v1.10
 * Green Hills Software [ http://www.ghs.com ]
 * Using command line: gbin2c mod.o foo -module_name bar -list
 * Date generated: Fri Jun  17 13:17:13 2011
 **/

struct static_module_info {
    const char * const name;
    const char * const data;
    const struct static_module_info * const next;
};

/* foo: */

const char foo_data[] = {
    0x66, 0x64, 0x61, 0x73, 0x0A
};

/* mod.o: */

const char mod_o_data[] = {
    0x7F, 0x45, etc.
};

const struct static_module_info __static_modules[] = {
    {
        "bar",
        (const char *)&foo_data,
        &__static_modules[1]
    },
    {
        "mod.o",
        (const char *)&mod_o_data,
        0
    }
};
```

Example 11.8. Including an Additional Options File

```
gbin2c -i modules -list -crc
```

In this example, the **-i** option is used to include an additional options file called **modules**, which contains the following text:

```
mod.o
foo -o bar
```

The **-o** option in **modules** specifies that the module name for **foo** should be `bar` and that its array name should be `bar_data`.

```
/**
 * This file was automatically generated by gbin2c v1.10
 * Green Hills Software [ http://www.ghs.com ]
 * Using command line: gbin2c -i modules -list -crc
 * Additional options from file: mod.o foo -o bar
 * Date generated: Fri Jun  17 13:17:13 2011
 * Note: CRC of input file and this array can be found at end of file
 **/

struct static_module_info {
    const char * const name;
    const char * const data;
    const struct static_module_info * const next;
};


/* foo: */

const char bar_data[] = {
    0x66, 0x64, 0x61, 0x73, 0x0A
};

const unsigned int bar_data_crc = 0x11A9EBE1;
/* CRC of input file = 0x11A9EBE1 */
/* Note that file CRC may not match array CRC if -skip or -limit were used */


/* mod.o: */

const char mod_o_data[] = {
    0x7F, 0x45, etc.
};

const unsigned int mod_o_data_crc = 0xD91D5642;
/* CRC of input file = 0xD91D5642 */
/* Note that file CRC may not match array CRC if -skip or -limit were used */


const struct static_module_info __static_modules[] = {
    {
        "bar",
        (const char *)&bar_data,
        &__static_modules[1]
    },
    {
        "mod.o",
        (const char *)&mod_o_data,
        0
```

```
        }
};
```

The gbincmp Utility Program

The **gbincmp** utility program compares two binary input files. The two input files can be ELF format object files or executable files.

If the sections in the two files are identical, **gbincmp** exits with no output. If they differ, it prints the first difference, unless you specify the **-continue** option.

The **gbincmp** utility program can compare any of the following:

- Contents of sections loaded on the target (target sections)
- Contents of sections not loaded on the target (host sections)
- Section headers
- Relocation tables
- String tables
- Symbol tables
- DWARF debug information sections

The syntax for this utility is as follows:

gbincmp [*options*] *file1 file2*

where *file1* and *file2* are the names of the two files to be compared.

The **gbincmp** *options* are:

-all Compares all sections.
-continue Instructs **gbincmp** not to stop at the first difference.
-debug **-nodebug** Enables or disables comparison of the `.debug` (DWARF) and `.debug_info` (DWARF2) sections.
-help Displays information about all options.

-host **-nohost** Enables or disables comparison of the contents of host sections.
-line **-noline** Enables or disables comparison of the line number information sections `.line` (DWARF) and `.debug_line` (DWARF2).
-normalize Ignores differences in encoded timestamps.
-prefixed_msgs **-no_prefixed_msgs** Enables or disables the insertion of the words `WARNING` and `ERROR` before every warning and error message.
-reloc **-noreloc** Enables or disables comparison of relocation information.
-section=*sections* **-nosection=*sections*** Enables or disables comparison of the sections named in the comma-separated *sections* list. For example, `-section=text,data` (ELF only).
-string **-nostring** Enables or disables comparison of string table information.
-symbol **-nosymbol** Enables or disables comparison of symbol table information.
-target **-notarget** Enables or disables comparison of the contents of target sections.
-v Verbose output. Shows what files and sections are compared.

The gbuild Utility Program

The **gbuild** utility program provides a command line interface to build projects created with the MULTI Project Manager (see the documentation about the MULTI Project Manager in the *MULTI: Managing Projects and Configuring the IDE* book). This utility can be used to integrate building your project into a script.

The syntax for this utility is as follows:

gbuild [*option*]... [*file*]...

where the optional *file* parameter can be:

- one or more project files (**.gpj**)
- one or more source or object files
- one or more executables

If you do not specify *file*, **gbuild** builds the file specified by the **-top** option. If you do not specify **-top**, **gbuild** tries to build **default.gpj** in the current directory.

The *options* for **gbuild** are:

-# Displays status messages and commands but does not perform any actions. This option is a shortcut for **-info -commands**.
-all Rebuilds all files, even if they are up to date.
-allinfo Displays the status messages that you would encounter if you rebuilt all files. This option does not actually build your files or perform the actions described in the messages. This option is a shortcut for **-all -info**.
-clean Deletes all previous output files without building.
-cleanfirst Deletes all existing output files and then initiates a build.
-commands Displays commands before they are executed.

-D*macro_name*[=*value*]

Defines a macro *macro_name* as *value* for use in project (**.gpj**) files and customization (**.bod**) files. See “Green Hills Project File Syntax” on page 906 for more information.

This option does not define a preprocessor symbol.

-find *filename*

Displays the first build path to the file *filename*. The build path specifies a route through the build structure to reach *filename*.

-findall *filename*

Displays all build paths to the file *filename*. The build paths specify routes through the build structure to reach *filename*.

-H

Displays includes (see “Display includes Preprocessor Directives Listing” on page 163).

--help

-help

-h

Displays usage information for basic **gbuild** *options*.

-Help

Displays usage information for all **gbuild** *options*.

-ignore

Continues building, even if compile or link errors occur.

-ignore_gbuild_dir_change

Do not rebuild a file simply because the current tools installation is different than the one used to build the existing version of the file.

-ignore_parse_errors

Continues building even if errors occur while parsing project files. Some critical errors cannot be ignored and will still halt the build.

-info

Displays status messages but does not perform the actions described in the messages.

-leave_output

Deletes only intermediate output files. This option is only available with **-clean** or **-cleanfirst**.

-link

Forces linking. The default behavior is to re-link the executable only if any of its components have changed.

-list

Lists all filenames. This option does not perform any build actions.

-list_depends

Lists known dependencies. You must build your project before passing this option. This option does not perform any build actions.

-list_options

Lists options specified in the **.gpj** file. This option does not perform any build actions.

-lockout

Uses a lock file to prevent simultaneous builds of the same program or library.

-lockoutretry

Waits to obtain a lock file. This option implies **-lockout**.

-nested_commands

Displays full commands, including internal commands, before they are executed. If you specify this option with **-info**, the output does not include internal commands.

-nice

Executes the build with low priority.

-nochecklibs

Disables Implicit Dependency Analysis. For more information, see “Implicit Dependency Analysis” on page 516.

-nolink

Generates object files but does not link them.

-noparallel

Specifies that **gbuild** cannot run parallel processes for the build. The build will be single threaded.

-noprogress

Suppresses progress messages.

-noreasons

Restricts the information given in progress messages.

For example, the message:

```
Compiling hello.c because hello.obj does not exist
```

is reduced to:

```
Compiling hello.c
```

-parallel[=*n*]

Specifies the number of processes that **gbuild** can run in parallel. The default value for *n* depends on the number of cores in your host system.

`-parallel=0` is equivalent to `-noparallel`.

-parallel_level=*level*

Controls how aggressively **gbuild** parallelizes the build. Valid values for *level* are:

- `low` — Parallelizes compiling source files within a single project (**.gpj**) file when safe to do so. This setting is equivalent to **-parallel** behavior in MULTI 5 and earlier.
- `medium` — [default when **-nochecklibs** is enabled] Parallelizes processing files within a single linked output when safe to do so. This option is similar to `low`, but parallelizes across Subprojects.
- `high` — [default] Parallelizes processing files whenever safe to do so, including linking tasks; however, if you have disabled Implicit Dependency Analysis with **-nochecklibs**, **gbuild** may not have enough information about link-time dependencies to parallelize linking tasks, and defaults to `medium`.

  If you use **-nochecklibs** with **-parallel_level=high**, you may want to explicitly specify dependencies using the **:depends** builder option (see "Dependencies Relative to Source Directory" on page 255).

If you specify this option after **-noparallel**, **gbuild** runs with the default number of processes instead of running a single-threaded build.

-path *list_of_project_files file*

Specifies a build path of project (**.gpj**) files to follow when building the file *file*. The build path specifies a route through the build structure to reach *file*. When using **-path**, you cannot specify any file arguments to **gbuild** other than the one specified for **-path**. This option is helpful when one file is utilized in multiple paths in your build tree, and you want to specify which path to use for building. For an example, see Example 11.13. Building One of Multiple Projects with the Same Name on page 515.

-preprocess

Only performs preprocessing on a project, generating **.i** files from applicable source.

-quiet

-silent

Prevents the utility from outputting any warning or error messages.

-skip=*target*

Skips the specified build target *target* when building. Here, *target* is the output name associated with a build file.

-statistics Displays comprehensive timing information.
-stderr=*filename* Redirects stderr output (status and error messages) to *filename*.
-strict Aborts **gbuild** if **gbuild** is unable to open a **.gpj** file in your build tree or if you specify an unrecognized option on the command line. Without this option, these problems only generate warning messages.
-time Displays basic timing information.
-top *filename*.gpj Specifies the Top Project file. By default, **gbuild** uses **default.gpj** in the current directory. This option allows you to specify a path to a project file in another directory.
-unix Displays directory slashes as /.
-verbose Prints additional information during the build.
-within *intermediate.gpj* Restricts searching for the specified *file* arguments to children of *intermediate.gpj*. This option is an alternative for **-path**; it allows you to specify multiple files, while restricting the search for those files to a sub-tree of your project. This option is helpful when there are multiple files with identical names in multiple sub-trees. For an example, see Example 11.13. Building One of Multiple Projects with the Same Name on page 515.

Using gbuild

The following examples take place in a directory, **MyProjects/Project1**, which contains a Top Project file called **default.gpj**, and a number of levels of subprojects.

Example 11.9. Building the Entire Project

To build the entire project, enter the following:

```
gbuild
```

If you do not pass any arguments, **gbuild** attempts to process **default.gpj** in the current directory.

Example 11.10. Building Part of Your Project

To build a particular executable, you must specify either its project file or the name of the executable itself.

For example, to build **print_utility.gpj**, a project file that generates an executable called **print_utility**, enter the following:

```
gbuild print_utility.gpj
```

or:

```
gbuild print_utility
```

Example 11.11. Building Individual Object Files

To build individual object files, you must specify either the initial source files or the object files that they will be compiled into.

For example, to build the source files **foo.c**, **bar.s**, and **baz.cxx** into object files, enter the following:

```
gbuild foo.c bar.s baz.cxx
```

or:

```
gbuild foo.o bar.o baz.o
```

Example 11.12. Finding Project Files

As your project grows in size, it might become difficult to find individual files, particularly if you are using several levels of subprojects. To search for the project file **print_utility.gpj**, enter the following:

```
gbuild -find print_utility.gpj
```

This command prints the position of **print_utility.gpj** in the project tree and the path to it from the current directory. The output resembles the following:

```
default.gpj utilities.gpj utilities/print_utility/print_utility.gpj
```

In this case, **print_utility.gpj** is a subproject of **utilities.gpj**, which is a subproject of **default.gpj**. The actual file can be found at **utilities/print_utility/print_utility.gpj**.

Example 11.13. Building One of Multiple Projects with the Same Name

If your project contains identically named sub-projects, use the **-path** or **-within** options to specify which project to build. For example, if your project structure is:

```
default.gpj
    target_one.gpj
        foo.gpj
        lib.gpj
    target_two.gpj
        foo.gpj
        lib.gpj
```

- To build the **lib.gpj** project in **target_two.gpj**, use the following command:

  ```
  gbuild -path default.gpj target_two.gpj lib.gpj
  ```

  The **-path** option only allows you to specify one item to build.
- To build both the **lib.gpj** and **foo.gpj** projects in **target_two.gpj**, use the following command:

  ```
  gbuild -within target_two.gpj foo.gpj lib.gpj
  ```

Example 11.14. Running gbuild in a Project Subdirectory

When building a portion of your project, **gbuild** looks for a **default.gpj** in the current directory to ensure that any build options set in the upper portions of the project structure are inherited appropriately. If your Top Project is not named **default.gpj** or does not exist in the current directory, you must specify its location with the **-top** option.

For example, if you have navigated to **utilities/print_utility/**, and intend to build **print_utility.gpj**, you should enter:

```
gbuild -top ../../default.gpj print_utility.gpj
```

Implicit Dependency Analysis

A program is explicitly dependent on all of its children in the project tree (subprojects, libraries, source files, etc.) All other dependencies are considered implicit dependencies. Implicit Dependency Analysis (IDA) attempts to locate two specific kinds of implicit dependencies:

1. Additional libraries referenced by **Prebuilt Library** objects within a project.
2. Additional libraries or objects pulled in by options (for example, **-l**, **-kernel**, **-linker.args**), or declared by using **:depends**.

The first type of IDA is performed as soon as the builder encounters a **Prebuilt Library** object. The second type of IDA is performed after all children of a program have been built, and before the link phase begins.

gbuild attempts to locate the build item that produces additional dependencies found via IDA, and builds the dependencies automatically if they are not current. Dependencies that **gbuild** cannot build (for instance, because they are system libraries) are ignored.

When searching for dependencies, **gbuild** looks for children of the following items:

- Ancestors of the Program **.gpj** being analyzed.
- Project and Subproject **.gpj** files that are children of any of the program's ancestors.
- Project and Subproject **.gpj** files that are not descendants of a buildable item.

IDA does not search for dependencies within projects that cannot be reached from the current top-level build file.

An implicit dependency can be defined after an item that references it. As long as it meets the preceding requirements, IDA will still find and build the implicit dependency. **gbuild** does not perform IDA during the clean phase of a build. Thus, implicit dependencies are not cleaned. You must clean the dependencies separately, or clean a parent project.

IDA is enabled by default, and can be disabled by adding **-nochecklibs** to the default build options in your Top Project. For more information about setting default build options, see the documentation for the **defaultBuildOptions** group 1 directive in

"Top Project Directives" on page 909. Disabling IDA may degrade parallel build performance. For more information, see the documentation for **-parallel_level=** in "The gbuild Utility Program" on page 509.

The gcolor Utility Program

The **gcolor** program takes text input, colors it using ANSI escape sequences, and outputs it to `stdout`.

The syntax for this utility is as follows:

gcolor [*options*] [*filenames*]...

where *filenames* is an optional list of one or more whitespace-delimited input filenames. If no files are specified, `stdin` is used.

The *options* for **gcolor** are:

-f Forces coloring even when output is redirected.
-help Shows the list of **gcolor** options.
-r *file* Reads the coloring rules from *file*. If this option is not passed, the default rules contained within **defaults/colors/build.rul** are used.

The **gcolor** utility reads in a set of rules from a rules file to determine how the input should be colored. If a line from the input cannot be identified as conforming to any rule, then no coloring is performed.

There are five types that can be defined using rules that are recognized by **gcolor**. Each one can output a specific foreground, background, and attribute.

Rule	Default Coloring
`info`	Foreground = Bold, Default
`warning`	Foreground = Bold, Magenta
`error`	Foreground = Bold, Red
`before`	Foreground = Normal, Default

Rule	Default Coloring
`after`	Foreground = Bold, Default

Note

The types `before` and `after` are used for coloring output from a `diff`, for example when comparing two files.

Color Configuration Files

The color configuration for each of the types can be controlled by creating a **colors.ini** file and saving it in your user configuration directory.

The syntax of a color configuration file is as follows:

```
[type]
attribute = value
foreground = value
background = value
```

where *type* and *value* are listed in the following tables:

Types
`info`
`warning`
`error`
`before`
`after`

Colors	
Value	**Background or Foreground**
-1	Default
0	Black
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta
6	Cyan
7	White

Attributes	
Value	**Attribute**
-1	Default
0	None
1	Bold
4	Underscore
5	Blink
7	Reverse
8	Concealed

Comment lines must be prefixed with a '`#`' character as the first non-whitespace character.

For example, to make `info` display in normal white text on a blue background, you would use the following:

```
[info]
attribute = 0
foreground = 7
background = 4
```

Note

If you do not specify coloring for a type, that type is colored in the default manner. If you do not specify *values* for `attribute`, `foreground`, or `background`, the default values are used.

Rules Files

The default set of rules is located in **defaults/colors/build.rul**. These correspond to the output from the builder and any of the programs that it invokes (the compiler, assembler, linker, etc.).

Alternatively, you can specify your own set of rules to use with the **-r *file*** option. You must create a rules file (***file*.rul**) and save it in the following location:

- Site-wide rules should be saved in the **config/colors** directory.
- User-specific rules should be saved in your .**ghs/colors** directory.

Writing a Rules File

Rules are described using regular expressions. To add a rule to a rules file, first you must give the type of the rule, followed by a colon, and then the desired regular expression. The regular expression begins with the first non-whitespace character following the colon.

For example, if you wanted to define directories in a standard Linux/Solaris-style directory listing as `info`, you can use a simple rule such as:

```
info: /$
```

There are two special rule types: `none` and `previous`.The `none` type causes matching text to be printed without color. This is useful if you want to prevent the

coloring of specific lines that would otherwise match against a later rule. For example, the rules:

```
none: ^foobar
info: ^foo
```

would color lines beginning with `foo`, except those beginning with `foobar`.

The `previous` type causes matching text to be colored the same as the previous line of text. This is useful to ensure that continuation lines are colored the same as their parents.

Comment lines must be prefixed with a '`#`' character as the first non-whitespace character.

Example 11.15. Coloring a File

To color file **foo**, using your own set of rules contained within **myrules.rul**:

```
gcolor -r myrules.rul foo
```

To color builder output using the standard rules, enter (for **sh** shells):

```
gbuild default.gpj 2>&1 | gcolor
```

Note that the syntax for **csh** shells is different. If you are using a **csh** shell, enter:

```
gbuild default.gpj |& gcolor
```

Note

Regular expression support is provided by the PCRE library package, which is open source software, written by Philip Hazel, and copyrighted by the University of Cambridge, England. It can be downloaded from the University of Cambridge FTP server [ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/].

The gcompare Utility Program

The **gcompare** utility program compares the code size of matching functions in two input files and produces a report. Input files can be ASCII text files, object files, object file libraries, or executable files. If an input file is a text file, it consists of lines in the following format:

```
name1  number1
name2  number2
...
```

Each name/number pair is on its own line, separated by blanks or tabs.

If either input file is an object file, an object library file, or an executable file, **gcompare** automatically runs **gfunsize -gcompare** (or another command specified by the **-x** option) to produce an input text file to compare.

You can mix all input file types with no restrictions, including files for different target CPUs.

The **gcompare** utility reads both input files. For any name that does not exist in both files, **gcompare** prints a warning, unless **-w** is specified. For each name that exists in both files, **gcompare** compares the old and new numbers. If the new number is worse (larger is worse unless **-i** is specified), **gcompare** writes a line to the output report file containing the name, new number, old number, and percentage by which the new number is worse.

For example, if the size of the function `main` has increased from 28 bytes in **a.out** to 32 bytes in `b.out`, then `gcompare a.out b.out` will produce:

```
    main          32      28  -14%
```

The syntax for this utility is as follows:

gcompare [*options*] *oldfile newfile*

where *oldfile* is the baseline input file and *newfile* is the input file for which deviations (improvements or degradations) are reported.

The **gcompare** *options* are:

-1

Produces a report with comparisons sorted from worst to best. All comparisons (including those in which *newfile* is no larger than *oldfile*) are shown. In addition, a summary is provided at the top, showing the aggregate size comparison between *newfile* and *oldfile*.

-a

Prints comparisons even when items exist in only one file.

-help

Displays information about all options.

-i

With **-i**, larger is better. Without **-i**, smaller is better.

-ignore *name*

Suppresses comparisons of symbol *name*.

-L

Formats the report in 132-column (landscape) mode. The default is 80-column (portrait) mode.

-min=*n*

Suppresses comparisons where the difference between the compared items is less than *n* bytes.

-r

Produces a 2-column report with comparisons sorted both from worst to best and from best to worst.

-v

Verbose mode, similar to **-1**, but with additional diagnostics.

-w

Suppresses warnings.

-width=*n*

Sets width (*n*) to any value greater than 32.

-x *cmd*

Specifies the command to execute on an input file recognized as an object, object library, or executable file. The default **-x** command is:

```
-x "gfunsize -gcompare -all"
```

-z

Does not show cases where files are the same.

If the **-r**, **-1**, or **-v** options are used, all comparisons are shown, regardless of the result of the comparison. With **-r**, two reports are shown side by side in two columns. The left column is sorted from best to worst, and the right column is sorted from worst to best.

Sample **-r** output:

```
   linpack.OS.o                        |    linpack.O.o
vs linpack.O.o                         | vs linpack.OS.o
BETTER by:           3208   3476 7.71% | WORSE by:            3476   3208 8.35%
---------------------------------------|---------------------------------------
epslon                 20     20    0% | dgefa                 484    356  -36%
main                  912    928    2% | matgen                304    252  -21%
dmxpy                1188   1224    3% | dgesl                 272    248  -10%
idamax                108    112    4% | dscal                  52     48   -8%
daxpy                  76     80    5% | daxpy                  80     76   -5%
dscal                  48     52    8% | idamax                112    108   -4%
dgesl                 248    272    9% | dmxpy                1224   1188   -3%
matgen                252    304   17% | main                  928    912   -2%
dgefa                 356    484   26% | epslon                 20     20    0%
```

This output can require many columns. Use the **-L** option for a 132-column format. On Linux/Solaris, use the **lpr -L** command to print in landscape mode.

The gdump Utility Program

The **gdump** utility program performs a similar function to the Linux/Solaris **dump** program. It produces information about various types of files.

The syntax for this utility is as follows:

gdump [*options*] *filename*

where *filename* is the name of the input file, and *options* are listed in one of the following tables:

Debugging File Options

You can use the **gdump** utility to print symbolic debugging information from a **.dbo**, **.dba**, **.dlo**, **.dla**, or **.dnm** file. The options for **.dnm** files are different than those for the other debugging files.

-c Performs internal consistency checks.
-C Performs internal consistency checks, but ignores warnings.

-d*info*

Enables the display of particular information, where *info* is a letter from the lists below.

The following values for *info* are available for all debugging file formats:

- `a` — the attributes
- `f` — the file table
- `h` — the file header
- `t` — the `typedef` table

The following values for *info* are available for all debugging file formats except **.dnm**:

- `c` — static call information
- `d` — the `#define` table
- `g` — actual call information
- `i` — include reference information
- `m` — frames
- `o` — code trees
- `p` — the procedure table
- `s` — the symbol table
- `u` — the auxiliary table
- `x` — cross reference information

The following values for *info* are available for the **.dnm** file format only:

- `b` — linkage stubs
- `d` — **.dlo** files
- `e` — the global `#define` table
- `i` — the trace regions table
- `k` — link labels
- `l` — libraries
- `s` — sections
- `u` — **.dnm** updates

-help

Displays information about all options.

-v

Performs internal consistency checks only. Does not display symbols.

-V Same as **-v**, but ignores warnings.

ELF File Options

The **gdump** *options* for ELF format files are as follows:

-asm Produces text sections as pure assembly language (see also the **-ytext** option).
-dwarf Produces DWARF information only.
-full Produces everything except section contents (see also the **-ysec** option).
-help Displays information about all options.
-integrity_checksum Applies the checksum calculation method used by INTEGRITY when the **-print_checksum** or **-verify_checksum** options are specified.
-load Displays ELF header summary.
-map Displays section summary.
-N Only displays information as indicated by the **-y** options.
-print_checksum Prints the checksum for each appropriate section. If **-verify_checksum** is also specified, checksums are assumed to exist and **-print_checksum** prints them for those sections where the checksum is found to be correct. If the **-verify_checksum** option is not specified, checksums are assumed not to exist in the section, so they are calculated using all bytes in the section.
-raw If the **-ysec** option is specified, this produces text sections in hexadecimal format instead of disassembly.
-section=*name* Displays the contents of the specified section only.

-sx

-nx

Enables or disables attempted shorter C++ demangling.

-sym

Uses symbol names instead of numbers in relocation output.

-v1

-v2

Specifies display of DWARF Version 1 or 2. The DWARF version in use is detected automatically by **gdump**, and the appropriate option set automatically.

-verify_checksum

Instructs **gdump** that all non-empty, allocated sections have a 4-byte checksum generated by the Green Hills linker. Compares the content of each section against the existing checksum and, if they do not match, displays both.

-y*info*

-n*info*

Enables or disables the display of particular information, where *info* is one of:

- `d`: DWARF debugging information
- `dynamic`: dynamic linkage information
- `f`: DWARF call frame information
- `g`: global offset table
- `h`: ELF header information
- `l`: DWARF line number information
- `m`: DWARF macro information
- `r`: relocation information
- `p`: procedure linkage table
- `s`: symbol table information
- `sec`: section contents
- `sh`: section header information
- `str`: string table information
- `text`: text section contents

COFF File Options

The **gdump** *options* for COFF files are as follows:

-ascii Displays section contents in ASCII.
-aux Displays auxiliary entries (in hex) with symbol entries.
-bigendian **-littleendian** Forces either a big or little endian target.
-brief Displays symbols only.
-c Displays string table.
-cpu=*name* Sets CPU type (*name*).
-d *num* [+d *num2*] Displays section *num* or sections from *num* to *num2*.
-f Displays file header.
-full Displays everything.
-g Displays global symbols.
-h Displays section headers.
-helpcoff Displays all options relating to COFF files.
-l Displays line number information.
-map Displays section summary.

-nx Does not attempt to demangle C++ names.
-o Displays optional header.
-r Displays relocation information.
-raw Produces section contents in raw form.
-s Produces section contents.
-sx Attempts shorter C++ demangling.
-sym Uses symbol names instead of numbers in relocation output.
-t *num* [+t *num2*] Displays symbol *num* or symbols from *num* to *num2*.
-ysec **-ytext** Enables the display of section contents and/or text section contents.

BSD a.out File Options

The **gdump** *options* for BSD a.out format files are as follows:

-c Displays section contents.
-h Displays file header.
-helpbsd Displays information about all BSD-specific options.
-r Displays relocation entries.

-s

Displays symbol table.

The gfile Utility Program

The **gfile** utility program performs a similar function to the Linux/Solaris **file** program. It displays the file type of each filename argument, and can also display additional information. For example, if a machine supports both big endian and little endian data ordering, then for an object file, object file library, or executable file, **gfile** displays the machine type and byte order. For unrecognized object files, **gfile** prints `unknown machine type`. In general, **gfile** handles BSD a.out, ELF, and some COFF formats.

The syntax for this utility is as follows:

gfile *filenames ...*

where *filenames* is a list of one or more filenames separated by whitespace.

If the current host system is a SPARC workstation running the Solaris 2.*x* operating system (which uses the ELF format for executable files), you will see the following behavior:

```
gfile /bin/od
/bin/od:        SPARC big endian executable (ELF)
```

The gfunsize Utility Program

The **gfunsize** utility program displays the code size of one or more named functions in an object file, library file, or executable. The code size of ELF functions is usually part of the ELF symbol information.

The syntax for this utility is as follows:

gfunsize [*options*] *filename*

where *filename* is the file to examine, and *options* are listed in the following table:

-addr Displays the address of each function.
-file Displays the filename before each function.
-func=*name* Displays the code sizes of the specified function or functions.
-gcompare Displays the output in a format suitable for input to the **gcompare** utility program.
-help Displays information about all options.
-hex Displays function code sizes and addresses in hexadecimal.
-nounderscores Removes leading underscores from function names.
The following options are mutually exclusive: • **-sect=*name*** Displays functions in section *name*. • **-sectnum=*n*** Displays functions in section number *n*.
-w Suppresses warnings.

The ghide Utility Program

The **ghide** utility program modifies the symbol table of an existing object file by converting all global symbols to local symbols, except for a specified list of global symbols that remain global. **ghide** does not affect `COMMON` variables. The object file can be either relocatable or not relocatable, but it must contain a symbol table; otherwise **ghide** has no effect.

The syntax for this utility is as follows:

ghide [*options*] *retain_list object_file*

where *retain_list* is the name of an input file containing symbol names, separated by whitespace or newline characters, that are to be retained (not hidden), and *object_file* is the name of the existing object file to be modified by **ghide**.

The **ghide** *options* are:

-help

Displays information about **ghide**.

Using ghide

Suppose an embedded application system consists of a kernel and several application tasks, all developed independently. Some global functions in the kernel are for kernel use only. Other global kernel functions can be called from the application tasks.

First, the kernel is linked into a single file using the linker's **-r** option to retain relocation information. Then, **ghide** is run on the kernel using a retain list of the functions that should remain visible to the application tasks. Finally, the application tasks are linked with the kernel file, producing a complete executable file.

The use of **ghide** ensures that only the desired global symbols in the kernel are visible to the application tasks. This also prevents duplicate symbol errors in the linker if any application might have a global symbol with the same name as an internal kernel-only function.

The gmemfile Utility Program

The **gmemfile** utility program creates a binary memory image of a fully linked executable. This memory image is suitable for downloading to target memory. Use the `-memory` driver option to automatically invoke `gmemfile`. For more information about `-memory`, see "Generate Additional Output" on page 220.

The syntax for this utility is as follows:

gmemfile *executable* [*options*]

where *executable* is the name of the executable from which to generate the memory image, and *options* are listed in the following table:

-byte *b* [of] *n*

By default, **gmemfile** outputs all data bytes to the memory image. This option looks at data in the input file in chunks the size of the bus width (*n*) and writes only the specified data bytes (*b*) within each data chunk in the input data to the output file. It can be used to separate odd or even data bytes or words for burning PROMs when the size of the PROMs differs from the size of the bus. It is also used to reverse the order of the data bytes for a shared bus environment.

The width, *n*, represents the width of the data bus. The maximum width allowed is 32 that represents a 256 bit bus. The bytes, *b*, within the specified bus width *n* are numbered 1 through *n* where 1 is the first byte and *n* is the last byte.

The bytes, *b*, can be specified in one of the following ways:

- as a single byte (for example, `-byte 1 of 4`). In this example, for each 4 byte chunk of input data, only byte 1 is written to the output file.
- as a range of bytes (for example, `-byte 1-4 of 8`). In this example, for each 8 byte chunk of data in the input file, bytes 1, 2, 3 and 4 are written to the output file.
- as a list of bytes (for example, `-byte 4,3,2,1 of 4`). In this example, for each 4 byte chunk of data, the bytes are reversed and written to the output file.
- as a combination of a list and a range (for example, `-byte 1-4,6,9-12 of 16`).

-end *address*

Specifies the highest address of data to be placed in the memory image. If *address* is higher than the highest address of the executable, the bytes between them are padded with zeros unless **-fill*x*** is passed. You can specify a section name instead of an address, and the highest address of that section is used.

-fill*x* *start end value*

Fills holes before, after, or between sections of the executable in the specified range *start* to *end* with the specified *value*. The byte order of the input file (big or little endian) is applied to the fill value.

-fill1 fills holes with a single byte of data.

-fill2 fills holes with two bytes of data, aligning the fill bytes on a 2-byte boundary.

-fill4 fills holes with four bytes of data, aligning the fill bytes on a 4-byte boundary.

If the *start* and/or *end* parameters are outside the address range of the data in the executable, the fill value is applied to the gaps at the beginning and/or end of the memory image. Section names can be specified instead of addresses for the *start* and *end* parameters. The start address of a section is used for the *start* parameter. The end address of a section is used for the *end* parameter.

When multiple **-fill** options cover the same area, **-fill4**, **-fill2**, **-fill1** options are processed in that order. The **-start** and **-end** options take precedence over the **-fill** options and can be used to control the bounds of the memory image.

-help

Displays a summary of the command line syntax and options for using **gmemfile**.

-hole *size*

Overrides the default checking for holes between sections.

By default, a warning message is displayed if there is a hole larger than 4096 bytes; no error is generated for any size hole.

This option causes an error to be emitted if there is a hole larger than *size* bytes.

-import *start filename*

Copies the binary data from the specified *filename* to the specified *start* address in the memory image. Use the **-import** option to bring in input from a previous run of **gmemfile** or from some other utility that produces a memory image.

If you specify a section name for the *start* parameter, the binary data is copied to the next byte after the last address in the specified section.

-map *filename*

Creates a detailed map file. The map file contains the command input, a list of all sections from the input executable file, a list of **-import** data, a memory map of the output memory file, and a summary of the memory image.

-o *filename*

Specifies the name of the memory image file to be output. The default is the name of the executable with the extension, if any, replaced by `.bin`. If there is no input file, the default name of the output file is **a.bin**.

-size *value* Sets the size of the memory image to *value* bytes. This is an alternative to using the **-end** option. An error is displayed if both **-size** and **-end** appear in the command input. By default, all data from the executable is written to the memory image.
-skip *sname* Excludes data in the specified section *sname* of the executable from the memory image.
-start *address* Specifies the *address* in the executable at which to begin copying data to the memory image. A section name can be specified instead of an address, and then the start address is the first address in the section. If the start address is lower than the lowest address in the executable, the bytes between the start address and the executable are padded with zeros or with a fill value if one was specified with **-fill1**, **-fill2**, or **-fill4**.
-w Does not display warnings.

Data Splitting

In some hardware implementations, the width of the data bus differs from the width of the PROMs. Depending on how the bus and PROMs are connected, it might be necessary to store low bytes, or words, in one PROM and high bytes, or words, in another PROM. The **gmemfile** utility supports such *data splitting* through the **-byte** option (see Example 11.21. Separating Low and High Bytes of Data into Separate PROMs on page 538 and Example 11.22. Separating the Low Half and High Half of Each 4-byte Interval on page 539).

The following examples use for illustration a program file **prog1** that contains the following sections:

Name	Start Address	End Address	Size
`.text`	`0x1000`	`0x10ff`	`0x100`
`.rodata`	`0x1100`	`0x12ff`	`0x200`
`.rdata`	`0x1300`	`0x15ff`	`0x300`
`.data`	`0x2000`	`0x22ff`	`0x300`
`.bss`	`0x2300`	`0x24ff`	`0x200`

The `.text` section contains all of the code in the program that is executed from ROM. The `.rodata` section contains read-only data that is accessed in ROM. The

`.rdata` section contains initialized data that is stored in ROM and copied to `.data` in RAM at program startup. The program references initialized data at the addresses in section `.data` in RAM. The `.bss` section contains uninitialized data. It is set to zero at program startup.

Example 11.16. Using gmemfile

```
gmemfile prog1.elf
```

In this example, because no options have been specified, the output file is called **prog1.bin**, the first byte in the memory image is the first byte of section `.text`, and its last byte is the last byte of `.rdata` (which is the last section that contains initialized data).

Example 11.17. Using the -start and -o Options

```
gmemfile -start 0 prog1.elf -o prog1.mem
```

In this example, the option `-start 0` sets the starting address at which **gmemfile** begins reading the executable to 0. Because the first section of the executable is `.text`, which begins at `0x1000`, the first `0x1000` bytes of the memory image are set to zero. The utility continues reading and outputting the data from sections `.text`, `.rodata`, and `.rdata`. The last byte in the memory image is the last byte of section `.rdata`, which is at file offset `0x15ff` in the memory image and at address `0x15ff` in the executable.

The option `-o prog1.mem` sets the name of the memory image to **prog1.mem**.

Example 11.18. Using the -fill Option

```
gmemfile prog1.elf -fill2 0 .text 0x3e7f -o prog1.mem
```

In this example, the `-fill2` option specifies that any holes between 0 and the end of section `.text` be filled with the value `0x3e7f`. The initialized data in `.text` is not overwritten by `-fill2`. The memory image produced in this example has the same start (`0`) and end (`0x15ff`) addresses as that in the previous example (the `-start 0` option is not needed, since `-fill2` starts at zero).

Example 11.19. Using the -start and -end Options

```
gmemfile prog1.elf -start 0 -end 0x1fff -o prog1.mem
```

In this example, the PROM is 0x2000 bytes wide and the PROM burner requires that all bytes in the PROM be filled. The **prog1** executable has a hole in front of its first section (from 0x0 to 0xfff) and after the end of its initialized data (from 0x1600 to 0x1fff).

The -start option sets the start address of the memory image to zero and -end sets its end address to 0x1fff. Because no -fill options have been used, the holes 0x0 to 0xfff and 0x1600 to 0x1fff are filled with zeros.

Example 11.20. Using the -start and -size Options

```
gmemfile prog1.elf -start 0 -size 0x2000 -o prog1.mem
```

In this example, the memory image produced is identical to that of the previous example. Instead of specifying an end address with -end, a size is specified with -size 0x2000.

Example 11.21. Separating Low and High Bytes of Data into Separate PROMs

```
gmemfile -byte 1 of 2 prog1.elf -start .text -size 0x100 -o prog1.low.mem
gmemfile -byte 2 of 2 prog1.elf -start .text -size 0x100 -o prog1.high.mem
```

In this example it is necessary to separate low and high bytes of data into separate PROMs, each of which is 0x100 bytes wide. The **gmemfile** utility is run twice, using the -byte option to select alternate bytes for each of the output files.

In the first command line, the option -byte 1 of 2 outputs the first byte of data in each two byte interval. The -start option sets **gmemfile** to reading the executable at 0x1000 (the first byte of .text). The -size option sets the size of the memory image to 0x100 bytes.

The first byte of data in the memory image is the first byte of data in section .text. The second byte of data in the memory image is the third byte of data in section .text. The last byte of data in the memory image at file offset 0xff is from the .rodata section in the executable at address 0x11fe (0x200 bytes above 0x1000).

If the bytes in the .text section of the executable consisted of:

```
01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10...
```

The following bytes would be output to the first memory image file:

```
01 03 05 07 09 0b 0d 0f...
```

The second command line takes the second byte in each 2-byte interval starting at the address of `.text` (`0x1000`). The first byte of data in the memory image is the second byte of data in section `.text`. The second byte of data in the memory image is the fourth byte of data in section `.text`. The last byte of data in the memory image is the byte in section `.rodata` at address `0x11ff`.

If the bytes in the `.text` section of the executable consisted of:

```
01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10...
```

The following bytes would be in the second memory image file:

```
02 04 06 08 0a 0c 0e 10...
```

Example 11.22. Separating the Low Half and High Half of Each 4-byte Interval

```
gmemfile -byte 1,2 of 4 prog1.elf -start .text -size 0x100 -o prog1.low.mem
gmemfile -byte 3,4 of 4 prog1.elf -start .text -size 0x100 -o prog1.high.mem
```

This example is similar to the previous example, except that the low half and high half of each 4-byte interval is separated. The first command takes the low 2-byte half of each 4-byte interval of data in the executable starting at the address of `.text` (`x1000`) until there are `0x100` bytes of data in the memory image.

If the bytes in the `.text` section of the executable consisted of:

```
01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10...
```

The following bytes would be in the first memory image file:

```
01 02 05 06 09 0a 0d 0e...
```

The second command takes the high 2-byte half of each 4-byte interval of data in the executable starting at the address of `.text` (`x1000`) until there are `0x100` bytes of data in the memory image.

If the bytes in the `.text` section of the executable consisted of:

```
01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10...
```

The following bytes would be in the second memory image file:

```
03 04 07 08 0b 0c 0f 10...
```

Example 11.23. Changing Data in the .rdata Section

This example shows you how to change the endianness of the data in the `.rdata` section of **prog1.elf**.

```
gmemfile prog1.elf -start .rdata -end .rdata -byte 4,3,2,1 of 4 -o reverse.bin
gmemfile prog1.elf -skip .rdata -import .rodata reverse.bin -o prog1.bin
```

The first command loads just the `.rdata` section of **prog1.elf** into memory. The `-byte 4,3,2,1 of 4` option reverses the byte order of the data. Then, **gmemfile** writes the result to the memory image file **reverse.bin**.

The second command loads **prog1.elf** into memory, but the `-skip .rdata` option discards any data in the `.rdata` section. The `-import .rodata reverse.bin` option loads **reverse.bin** after the end of the `.rodata` section (the beginning of the `.rdata` section). Finally **gmemfile** writes the result to **prog1.bin**.

Note

Because the `-import` option instructs **gmemfile** to place the data in `reverse.bin` immediately after the `.rodata` section, this example works only if the end of the `.rodata` section is one byte before the beginning of the `.rdata` section.

The gnm Utility Program

The **gnm** utility program performs a similar function to the Linux/Solaris **nm** program. It displays the symbol table of an object file, object library file, or executable file created by Green Hills development tools.

The syntax for this utility is as follows:

gnm [*options*] *filenames ...*

where *filenames* is one or more whitespace-delimited input files, and *options* are listed in the general options table and the appropriate object format options table:

General Options

The general **gnm** options are as follows:

-help Displays general options and ELF-specific options.
-helpbsd Displays BSD-specific options.
-helpcoff Displays COFF-specific options.
-helpelf Displays ELF-specific options.
-output *file* Writes the output from **gnm** to *file*.
-V Prints the **gnm** version.

ELF File Options

The **gnm** options for ELF format files are as follows:

-a Prints special symbols that are normally suppressed.

-D Prints symbols from the dynamic symbol table.
-defined_only Prints only defined symbols.
-dg Prints defined symbols and global symbols. It does not print debug symbols.
-g Prints only global symbols.
-h Does not print headers.
-l Prints an asterisk (`*`) after symbol type for WEAK symbols. This is for **-p** mode only.
-n Sorts output by symbol name.
-no_debug Does not print debug symbols such as `..bof`, `..eof`, and `..lin`
-no_dotdot Does not print any symbols beginning with "`..`".
-no_line Does not print line debug symbols. For example, `..lin`.
-o Prints value and size of symbols in octal.
-p Uses 3-column output format.
-prefixed_msgs **-no_prefixed_msgs** Enables or disables the printing of `WARNING` and `ERROR` before messages.
-r Prefixes filename to each line of output.
-R Prefixes archive name and filename to each line of output.

-S In 3–column output mode, **-S** prints section names instead of one-letter codes.
-u Prints undefined symbols only.
-v Sorts output by symbol value.
-X Does not print local compiler labels (`.Ln`).
-x Prints value and size of symbols in hexadecimal.

BSD a.out File Options

The **gnm** options for BSD a.out format files are as follows:

-a Prints all symbols, including debugging information. Not available with the **-s** option.
-g Prints only external symbols.
-n Sorts symbols by value.
-o Prefixes lines with object or archive element name.
-p Does not sort symbols.
-r Sorts in reverse order.
-s Sorts external symbols by size.
-u Prints only undefined symbols.

COFF File Options

-a Prints special symbols that are normally suppressed.
-A Uses System V output format.
-E Treats input files as Alpha ECOFF format.
-g Prints only global symbols.
-h Does not print headers.
-p Uses 3-column output format.
-r Prefixes filename to each line of output.
-s Prints section names.
-t Truncates standard call suffixes (Win32).
-T Truncates names to 20 characters.
-u Prints only undefined symbols.

Output Formats

By default, **gnm** produces output similar to the following excerpt (indices 3 through 11 are omitted for brevity):

```
[Index]   Value      Size    Type  Bind  Other Shndx   Name

[1]     |         0|       0|FILE |LOCL |0    |ABS    |hi.c
[2]     |         0|       0|SECT |LOCL |0    |.text  |.text
[12]    |         0|      52|FUNC |GLOB |0    |.text  |main
[13]    |         0|       0|NOTY |GLOB |0    |UNDEF  |printf
```

Column	Meaning
`Index`	Position of symbol in the symbol table.
`Value`	The value or address of the symbol.
`Size`	Size of the symbol (for example, a 4-byte integer symbol has a size of 4).
`Type`	One of the following: • `NOTY`: Symbol without type. • `FILE`: Filename symbol. • `SECT`: Section name symbol. • `OBJT`: Data symbol. • `FUNC`: Code symbol.
`Bind`	One of the following: • `LOCL`: Local symbol (for example, C/C++ static). • `GLOB`: Global symbol. • `WEAK`: Weak global symbol (if undefined, the symbol's value resolves to zero).
`Other`	Reserved field, generally zero.
`Shndx`	The name of the section where the symbol is defined. For example: • `.text`: Code defined in the `.text` section. • `.data`: Data defined in the `.data` section. • `ABS`: No section (for example, a filename symbol). • `COMMON`: Common variable whose section is not yet determined.
`Name`	Symbol name.

The **-p** option enables an alternative, easily parsable, 3-column format, which provides backward compatibility with tools that can read only this format:

```
0000000000 F hi.c
0000000000 S .text
0000000000 T main
0000000000 U printf
```

The first column is the value or address of the symbol. The second column is its type (as shown in the following table), and the third is the symbol name. Note that this format cannot fully describe all sections and storage classes.

Type	Meaning
A	External absolute
a	Local absolute
B	External zeroed data
b	Local zeroed data
C	Common variable (same as B except not yet assigned to a section)
D	External initialized data
d	Local initialized data
F	Filename
I	Local thread local-storage
L	External thread-local storage
N	Informational symbol
S	Section name
T	External text
t	Local text
U	External undefined

The gpatch Utility Program

The **gpatch** command line utility program allows you to install and create simple patches. It is most commonly used to install a patch provided by Green Hills support staff.

To install a patch into the current working directory, ensure the MULTI IDE or Compiler installation directory is in your path and run:

gpatch [-install_patch] *patch_filename*

The following table describes command line options supported by **gpatch**.

-help Displays documentation for all **gpatch** command line options.
-install_patch *patch_filename* Installs the patch *patch_filename*. By default, the patch is installed into the current working directory, but you may also pass **-target_dir** (below) to specify another location.

-key *installation_key*   Decrypts the patch using *`installation_key`*.
-list   Lists the files contained in the archive. This does not install the patch.
-nouninstall   Does not create an uninstallation archive.
-target_dir *directory*   Specifies the directory where the patch will be installed. If you do not pass this option, the patch is installed into your current working directory.

For a complete list of the command line options available for use with **gpatch**, run **gpatch -help**. Example uses of the **gpatch** utility program follow.

To list all files in **patch_1234.iff** without installing the patch:

```
> gpatch -list -install_patch patch_1234.iff
```

To install patch **2028.iff** into **C:\ghs\multi_*xxx***:

```
> gpatch -install_patch 2028.iff -target_dir C:\ghs\multi_xxx
```

The gpjmodify Utility Program

The **gpjmodify** utility program allows you to automate editing of project **.gpj** files. General editing should be performed through the Project Manager (see the documentation about the MULTI Project Manager in the *MULTI: Managing Projects and Configuring the IDE* book), or via a text editor.

Note

You can edit only one file at a time using **gpjmodify**.

Editing Existing .gpj Files

The syntax for editing existing **.gpj** files is as follows:

gpjmodify *file*.gpj [[*sub*.gpj…]|[*@path-to-sub*]] [*options*] *action*

where:

- ***file*.gpj** is a Top Project file. This is either the file to be edited or the ultimate parent of ***sub*.gpj**.
- The optional ***sub*.gpj…** is an immediate child subproject of ***file*.gpj** or a chain of subprojects (the last of which will be edited). Alternately, you can specify **@path-to-sub.gpj**, an absolute path to a subproject at any point in the project tree.
- *options* are listed in the following table:

-f Forces the edit, even if errors would result.
-help Displays help information.
-o *output*.gpj Writes the edited file to ***output*.gpj**. By default, edits are written to the existing file.
-quiet Suppresses warning messages.
-toolsdir *directory* Specifies an alternative MULTI Compiler installation *directory*.

- *action* listed in the following table:

-add_after *child* [-filetype *type*] *files* [-sourcedir *dir_paths*] Inserts *files* in the project's list of children, directly after *child*.
-add_before *child* [-filetype *type*] *files* [-sourcedir *dir_paths*] Inserts *files* in the project's list of children, directly before *child*.
-add_customization_file *filename* Adds a customization file to the Top Project.
-add_to_back [-filetype *type*] *files* [-sourcedir *dir_paths*] Inserts *files* at the end of the project's list of children.
-add_to_front [-filetype *type*] *files* [-sourcedir *dir_paths*] Inserts *files* at the beginning of the project's list of children.
-change_macro *name=value* Changes the value of the existing macro *name* to *value*.
-make_tree_canonical Canonicalizes filenames used in the project. This option applies to the entire project tree, not just the specified project file.
-modify *old_option new_option* Replaces all instances of *old_option* in the specified project with *new_option*.
-remove *files* Deletes the specified *files* from the project's list of children.
-remove_customization_file *filename* Removes the specified customization file from the Top Project.
-replace_with *existing_child* [-filetype *type*] *new_child* Replaces the references to the file *existing_child* with references to *new_child*.
-set *options* Inserts a list of build *options*.
-set_target *target_file* Specifies the project's target file. This is only saved if you are modifying a Top Project file.
-unset *options* Deletes a list of build *options*.

Pass the **-filetype *type*** parameter only if the files you are adding have non-standard extensions that **gpjmodify** is unable to recognize. For a list of accepted values, see "File Types" on page 915. If *type* contains a space, you must enclose it in quotes.

The **-sourcedir *dir_paths*** option specifies paths in which to search for the specified *files*. **gpjmodify** always uses the first file that it finds.

For any of the actions that require a list of *files* or *options*, each item in the list must be separated with a space. If one of the items in the list contains a space, you must enclose that item in quotes.

Creating a New .gpj File

The syntax for creating a new **.gpj** file is as follows:

gpjmodify *file*.gpj -create *type* -target *target_file* [*options*]

where ***file*.gpj** is the name of the file to be created, *type* is the type of project file to create (usually `Project`, `Subproject`, `Program`, or `Library`), and *target_file* specifies the project's target. Valid *options* are listed in the following table:

-quiet Suppresses warning messages.
-toolsdir *directory* Specifies an alternative MULTI Compiler installation *directory*.
-top Specifies that the new file will be a Top Project project file.

Generating a Makefile

Normally, if you need to build your project from the command line, you should use the **gbuild** utility program, which provides a command line interface to build MULTI projects (see "The gbuild Utility Program" on page 509). However, if you need to integrate Green Hills tools with an existing makefile structure, the **gpjmodify** utility can generate a rudimentary makefile from a project (**.gpj**) file. Use this makefile as a guide when writing makefiles for your project.

To generate a makefile with **gpjmodify**, enter the following command: **gpjmodify** *file*.gpj -generate_makefile [*options*]

where ***file*.gpj** is the **.gpj** file to be converted, and valid *options* are listed in the following table:

-quiet Suppresses warning messages.
-toolsdir *directory* Specifies an alternative MULTI Compiler installation *directory*.

Tip

If you also need to know the commands **gbuild** executes when building your project, enter the following command:

```
gbuild -info -commands file.gpj
```

The gsize Utility Program

The **gsize** utility program performs a similar function to the Linux/Solaris **size** program. It analyzes object files, object library files, or executable files and, for each file, displays the size of each section in bytes. If more than one file is named, or if an object library is named, **gsize** prints the name of the file with the section name and totals for each section.

The syntax for this utility is as follows:

gsize [*options*] *filename*

where *filename* is the name of the input object file, object file library, or executable file.

The *options* for **gsize** are as follows.

-all Causes all sections to be displayed, including empty sections and non-allocated sections.
-commons Displays a block of information describing each `COMMON` symbol in each output file (see "Text and Data Placement" on page 76). This block is suppressed if you pass the **-table**, **-nodetails**, or **-nobss** options. This option implies **-count_commons**.
-count_commons Includes the size of common variables in the total size of `.bss`, the size of zero-common variables in the total size of `.zbss`, and the size of small-common variables in the total size of `.sbss`. This option has no effect if you pass the **-bss** option.
-details — [default] **-nodetails** Displays or suppresses detailed information about each section in each file.
-help Displays information about all options.
-nobss Suppresses `bss` sections. This option makes **-commons** and **-count_commons** have no effect.
-nodata Suppresses `data` sections.

-notext Suppresses `text` sections.
-table Displays output in table format. This option implies **-details** and **-nototals**.
-text Only displays `text` sections, suppressing `data` and `bss` sections. This is the same as **-nodata** and **-nobss**.
-totals — [default] **-nototals** Displays or suppresses the summary information. If you pass **-totals**, summary information is output only if **gsize** is processing multiple files or common symbols.
-zero Displays zero-length sections.

The gsrec Utility Program

The **gsrec** utility program converts executable files or a multi-core archive into Motorola S-Record, Intel hexadecimal, or Tektronix hexadecimal format. These file formats contain ASCII representations of binary data. Many simulators, In-Circuit Emulators (ICEs), PROM programmers, and debuggers use one of them as a program download format. Use the `-hex` and `-srec` driver options to automatically invoke `gsrec` and generate a Intel hexadecimal or S-Record file after linking a program. For more information about these driver options, see "Generate Additional Output" on page 220.

The syntax for this utility is as follows:

gsrec [*options*] *filenames*

where *filename* is the name of an input executable file or multi-core archive to be converted to one of the output formats. Motorola S-Records is the default output format.

The **gsrec** *options* are:

-allow_overlap Allows sections to overlap. By default with more than one executable file or a multi-core archive a fatal error will be given if any non-empty sections overlap.
-allow_relocated_sections Allows sections with a non-empty matching `.rel` or `.rela` section to be converted to hex.
-auto Determines byte order from the file header. This is the default.
-B **-L** Specifies a big or little endian input file.
-bytes *n* Sets the maximum count of data bytes per record (minimum 4). For S-Record output, the maximum and default are 28. For Intel hexadecimal output, the maximum and default are 32.
-e *addr* Sets entry point in the termination record to given address.

-end *address*

Specifies the highest *address* of data to be output.

-eol *char-seq*

Specifies the character sequence used to terminate each S-Record, where *char-seq* is one of:

- `cr`: a `\r` character.
- `crlf`: a `\r\n` combination. This is the default in Windows.
- `lf`: a `\n` character. This is the default in Linux/Solaris.

-fill*x n1 n2 v*

Fills memory from address *n1* to address *n2* with the *x*-byte value *v* (where *x* is 1, 2, or 4). This option is not supported with multiple input files or a multi-core archive.

-help

Displays information about all options.

-hex86

-hex386

-tekhex

Specifies output of Intel HEX86, HEX386, or extended Tektronix hexadecimal format.

The Intel HEX86 hexadecimal format can handle memory addresses only in the range of `0x00000000` to `0x0010ffef`. When this address range is exceeded, **gsrec** displays the message: `an address is too big`.

The Intel HEX386 format supports the full 32-bit address space. A linear address record containing the upper 16-bits of the address was added to the HEX86 format to form the HEX386 format.

The Tektronix extended tekhex format supports the full 32-bit address space. The **-tekhex** option is deprecated and may be removed in future versions of **MULTI**.

-interval *n*[:*m*]

Places only those data bytes from the input file that occur within the specified interval in the output file. Values for *n* are 1 through 8. The default is 1, indicating that every byte will be output. The *m* value specifies the number of output bytes for each interval. For example, `4:2` outputs 2 consecutive bytes out of every 4.

This option requires that you set the **-start** and **-romaddr** options.

-name *string*

Uses the first 10 characters of *string* as the module name in the `S0` initial record.

-noS0

Suppresses production of an `S0` initial record.

-noS5

Suppresses production of an `S5` block count record.

-noSLA

Do not output the Start Linear Address Record in HEX386 mode. Instead, the start address is placed in the end record.

-o *filename*

Specifies the output filename. By default, output is sent to standard output.

-pad*x n1 n2 v*

Pads *x*-byte holes between hexadecimal data records from address *n1* to *n2* with value *v*. *x* may be `1`, `2`, or `4`. This option is not supported with multiple input files or a multi-core archive.

-page_boundary

-no_page_boundary

-page_boundary ensures that no record crosses a 64 kbyte boundary, and only applies to **-hex86** or **-hex386**. The default behavior is **-no_page_boundary**.

-prefixed_msgs

-no_prefixed_msgs

Enables or disables the printing of `WARNING` and `ERROR` before messages.

-raw

Treats input files as raw data to be placed at the address specified by **-romaddr**. This option is not supported with a multi-core archive.

-romaddr *addr*

Sets start address in ROM to place data. The default is the same address as in the input file.

-S*type*

Specifies the type of S-Record to be produced, where *type* is one of the following:

- `1`: `S1` data records (16-bit addresses)
- `2`: `S2` data records (24-bit addresses)
- `3` (default): `S3` data records (32-bit addresses)
- `5`: an `S5` record with a count of data records (32-bit count)
- `5old`: an old format `S5` record (16-bit count)
- `7`: an `S7` end record (32-bit entry point)
- `8`: an `S8` end record (24-bit entry point)
- `9`: an `S9` end record (16-bit entry point)

The default is to produce `S3` data records, an `S5` record, and an `S7` end record.

-skip *s*   Does not output data for section *s*.
-skip_unknown_file   Silently ignores files which are neither ELF nor COFF.
-sort   Sorts output records in ascending order. If multiple executable files or an archive containing multiple files is given, records will only be sorted within each file. If input files contain overlapping addresses, records will be output for both input files.
-start *address*   Specifies the lowest *address* of data to be output.
-w   Does not issue a warning when the input is a relocatable COFF object file.
-warn_overlap   Issues warnings for input sections that overlap.

S-Record Output Format

An S-Record file contains ASCII text that can be displayed or edited. There are eight types of S-Records, numbered `S0` to `S9`, including the following:

- An `S0` header record that identifies the program.
- Data records that represent the binary data in the program.
- An `S5` data count record that contains the count of data records in the S-Record file.
- A termination record that contains the address to begin program execution.

Not all S-Record reader programs behave the same way. Some programs require certain types of data records. Some target environments do not accept `S5` data count records, and some require certain types of termination records. The **gsrec** utility provides options that handle many of these cases.

Every S-Record begins with the letter S followed by a digit (0 through 9) representing the format of the line. The next two characters are a hexadecimal representation of the number of bytes in the remainder of the line, which always consists of the fields referred to as address, data, and checksum. The address field

may contain 2, 3, or 4 bytes depending on the format. The length of the data field varies. The checksum field always contains 1 byte.

For example, the following `S3` record has a length of 33 (`0x21`) bytes.

```
S3 21 00000000 00E280FF04001FE8084807400638403600000636000B20580FF0000 B5
```

The 4-byte address is `00000000`, the data field contains 28 bytes, and the checksum is one byte.

Data and Termination Records

A data record contains the address where data is loaded, followed by the data itself. The three types of data records, `S1`, `S2`, and `S3`, differ in load address size, as indicated in the previous table.

By default, an `S5` data count record is emitted. If your S-Record loader does not handle `S5` records, use the option **-noS5** to avoid outputting an `S5` data count record.

By default, the `S5` data record contains a 4-byte address. Some old S-Record readers expect an `S5` record to contain a 2-byte address. In this case, use the **-S5old** option.

A termination record contains the entry point (the address where program execution begins). There are three types of termination records, `S7`, `S8`, and `S9`, as indicated in the previous table.

Some S-Record readers accept data records up to a specified length. Use the option **-bytes *size*** to set the maximum data record length.

If you forget to specify the entry point address when linking, you can use **-e** *address* to set the entry point to the given *address*.

Data Splitting

In some hardware implementations, the width of the bus differs from the width of the PROMs. Depending on how the bus and PROM's are connected, it might be necessary to store even bytes in one PROM and odd bytes in another PROM. *Data splitting* is a technique of dividing the data into even and odd bytes, or in other ways.

gsrec can perform data splitting. The **-start** option specifies the starting address of the data in the object input file to output to the S-Record output file. The **-end** option specifies the last address of data in the object input file to output. The **-interval** option specifies the distance between bytes in the input file to output. A value of 2 for **-interval** outputs every other byte. Sometimes it is necessary to relocate the data to address zero in the S-Record output file for programming PROMs. The **-romaddr** option specifies the start address of the data bytes in the S-Record output file. The last two succeeding examples separate even and odd bytes into two different S-Record files.

The following examples use a program file **prog1** that contains the following sections:

```
1: .text,   address: 0x1000, size: 0x100
2: .data,   address: 0x2000, size: 0x200
3: .bss,    address: 0x3000, size: 0x200 (No Load Section)
4: .data2,   address: 0x4000, size: 0x200
```

gsrec Examples

Example 11.24. Using gsrec

```
gsrec prog1
```

The **gsrec** program reads the data in the input file **prog1** and writes the S-Records to the standard output. Because the contents of section **.bss** are not loaded, no S-Records are output for its data.

Example 11.25. Using the -Stype Options

```
gsrec -S1 -S9 prog1 -o   prog1.run
```

The **gsrec** program reads the data in the input file **prog1** and writes the S-Records to the specified output file **prog1.run**. Data are output as `S1` records that have a 16-bit address space. The start address is output as an `S9` record that has a 16-bit address space.

Example 11.26. Using the -start, -end, and -o Options

```
gsrec -start 0x1000 -end 0x1080 prog1 -o prog1.run
```

The **gsrec** program reads the data in the input file **prog1** starting at address `0x1000` and ending at address `0x1080` and writes the resulting S-Records to the output file **prog1.run**.

Example 11.27. Using the -romaddr Option

```
gsrec -start 0x1000 -end 0x1080 -romaddr 0 prog1 -o prog1.run
```

The **gsrec** program reads the data in the input file **prog1** starting at address `0x1000` and ending at address `0x1080`, relocates the data to start at address zero, and writes the resulting S-Records to the output file **prog1.run**.

Example 11.28. Using the -interval Option for Even Data Bytes

```
gsrec -start 0x1000 -end 0x1080 -romaddr 0 -interval 2 prog1
```

The **gsrec** program reads the data bytes at even addresses in the input file **prog1** starting at address `0x1000` and ending at address `0x1080`, relocates the data to start at address zero, and writes the resulting S-Records to the standard output.

Example 11.29. Using the -interval Option for Odd Data Bytes

```
gsrec -start 0x1001 -end 0x107F -romaddr 0 -interval 2 prog1
```

The **gsrec** program reads the data bytes at odd addresses in the input file **prog1** starting at address `0x1001` and ending at address `0x107F`, relocates the data to start at address zero, and writes the resulting S-Records to the standard output.

Example 11.30. Using the -interval Option for 2 Out of Every 4 Data Bytes

```
gsrec -start 0x1002 -end 0x107F -romaddr 0 -interval 4:2 prog1
```

The **gsrec** program reads 2 out of every 4 data bytes in the input file **prog1** starting at address `0x1002` and ending at address `0x107F`, relocates the data to start at address zero, and writes the resulting S-Records to the standard output.

The gstack Utility Program

The **gstack** utility program analyzes a program to report the maximum stack size each task might need during execution and the call chain that produces it. Bugs caused by exceeding the available stack can be very difficult to track down, and they are often responsible for intermittent crashes, errant behaviors, and security

leaks. By using **gstack** on a regular basis, you can be sure that your program will never exceed the available stack.

The syntax for the **gstack** utility is:

gstack [*options*] *program*

For a list of options, see "gstack Options" on page 575.

Note

GNU also has a utility named **gstack**. When invoking **gstack**, make sure to check that you are using the program in your Green Hills installation.

Preparing Your Program for gstack Analysis

gstack analyzes the call graph of a program and determines the maximum amount of stack it needs. While **gstack** does most of its analysis directly from existing code, it may require you to provide additional information order to produce correct results. There are also ways to modify your code to make it easier for **gstack** to analyze, particularly when it uses function pointers or recursion.

To prepare your program for use with **gstack**:

1. Make sure all files in your project are built with the debug level set to anything other than **--no_debug** (see "Debugging Level" on page 156).
2. Determine the program entry points that you want to analyze. Run **gstack** on your program, passing the entry point functions as arguments, and examine the report that it generates. For example:

   ```
   > gstack -entry_point main a.out
   ```

   For each function that you specify as an entry point, **gstack** reports the maximum amount of stack that can be used by a call chain starting with that function. Stack used by a program before the entry point is not included.
3. If the **gstack** report contains a `Missing Information` section, you need to specify additional information to **gstack**. Address each of the following types of missing information in the list as follows:

- `MC` (missing call) — An assembly function for which no call information has been provided. See "Annotating Assembly Functions for gstack" on page 562.
- `MS` (missing stack) — An assembly function for which no stack information has been provided. See "Annotating Assembly Functions for gstack" on page 562.
- `IC` (incomplete call) — A C function that requires additional call information. See "Annotating C Functions for gstack" on page 563.
- `IS` (incomplete stack) — A C function that requires additional stack information. See "Annotating C Functions for gstack" on page 563.

At this time, ignore items specified as `MI` (maximum instances); they will be addressed in step 5.

4. Break up recursive clusters wherever possible. See "Working With Recursive Clusters" on page 565.
5. At this point, the remaining items in the `Missing Information` section should be marked only with `MI` (maximum instances). These are functions that **gstack** has determined may appear more than once in a given call stack, but for which an upper bound on the number of times they may appear has not been specified and cannot be computed. To address these items, see "Annotating Recursive Functions" on page 570.
6. If your program uses interrupts or similar features where the program's execution may be interrupted and continued from some other function using the same stack, this information must be provided with the appropriate command line options. See "Specifying Interrupts" on page 572.
7. Use the **-stack_limit** option to specify an upper bound on the stack usage (in bytes) for each specified entry point. **gstack** issues an error if it determines that the stack limit for any entry point can be exceeded during the program's execution.

Annotating Assembly Functions for gstack

When a function is defined in assembly, either in an assembly source file or within a `pragma asm` block in a C file, you must provide **gstack** with information about the stack usage of the function and any calls it makes. If **gstack** detects an assembly function for which this information is missing, it lists the function in the `Missing Information` section of its report with `MC` (missing call) and/or `MS` (missing stack).

If the function is defined in an assembly file, you must provide the missing information with assembler directives that immediately follow the definition of the function. If the function is defined in a C file, you must declare the function outside the `pragma asm` block, and provide the missing information with pragma directives immediately before the declaration. **gstack**-specific assembler directives in C source files are ignored.

To specify call information for an assembly function, use one of the following for each distinct function that the assembly function calls:

- Assembly file — `.scall` *`func, callee`*
- C file — `#pragma ghs stactic_call` *`callee`*
- Command line — **-a *func*:*callee1*,*callee2*,...**

If the function does not make any calls, use:

- Assembly file — `.scall` *`func`*`, __leaf__`
- C file — `#pragma ghs static_call 0`
- Command line — **-a *func*:__leaf__**

To specify stack information for an assembly function, use one of the following:

- Assembly file — `.maxstack` *`func, size`*
- C file — `#pragma ghs max_stack (`*`size`*`)`
- Command line — **-f *func=size***

where *size* is the maximum amount of stack that is allocated by the function at any point during its execution. The `.maxstack` directive is ignored unless at least one `.scall` directive is specified for the same function.

Annotating C Functions for gstack

There are certain constructs for which information needed by **gstack** is not statically computed. For example, **gstack** is unaware of:

- function calls within `asm` statements
- stack usage of `asm` statements

- stack usage of variable length arrays (VLAs)
- stack usage of calls to `alloca()`

If it detects any of these constructs within a C function, it lists the function in the `Missing Information` section of its report with `IC` (incomplete call) and/or `IS` (incomplete stack).

When using pragma directives to provide this information to **gstack**, specify the directive immediately before the start of the function.

To specify call information for a C function, use one of the following:

- C file — `#pragma ghs static_call` *`callee`*
- Command line — **-a *func*:*callee1*,*callee2*,...**

If no calls are made from within the function's `asm` statements, use:

- C file — `#pragma ghs static_call 0`
- Command line — **-a *func*:__leaf__**

To specify stack information, use one of the following:

- C file — `#pragma ghs extra_stack (`*`size`*`)`
- Command line — **-x *func*=*size***

where *size* is a C constant that represents the total additional stack usage required by the `asm` statements, VLAs, and `alloca()` calls used within the function.

If a function for which you have provided all necessary information inlines a callee for which you have not provided enough call or stack information, **gstack** may list the function as missing call or stack information. We recommend that you make sure the necessary information is provided for all of a function's callees before providing the information for the function itself.

Warning

gstack results are undefined if the stack pointer is different upon entry and exit from an `asm` statement.

Working With Recursive Clusters

When **gstack** analyzes your program, it computes the program's *recursive clusters*, sets of functions for which any function can reach any other function in the set, either through a direct call or series of calls.

When computing the program's maximum stack usage, **gstack** assumes that a call chain containing any function that belongs to a recursive cluster will incur the total possible stack usage of that cluster. That value is computed by multiplying the individual stack usage of each of the cluster's functions by the function's maximum instances value (the maximum number of times it can appear in a single call stack) and summing the result over all of the functions that make up the cluster.

This approach ensures that **gstack** always computes a correct upper bound for the cluster's stack usage, but it also means that the value will often be very conservative, particularly for large or complex clusters. We strongly recommend that you modify your code to eliminate or simplify recursive clusters as much as possible to get the best results.

To eliminate and simplify clusters:

1. Use strong function pointer types to remove spurious recursions that appear in the call graph due to the ambiguity of function pointers (see “Using Strong Function Pointers” on page 566).
2. Identify and remove recursions that cannot occur at run time. For example, you should remove the call from `funcTwo()` to `funcOne()` below:

```
int funcOne(int x);

int funcTwo(int x){
    if(x > 0)              // This condition cannot be met
        return funcOne(x); // so funcOne() never gets called
    else
        return 0;
}

int funcOne(int x){
    x--;
    return funcTwo(x);
}

int main(void){
    int x = funcOne(0);
    return x;
}
```

3. Re-factor code to separate distinct recursive cycles into smaller clusters (see "Re-Factoring Code to Simplify Recursive Clusters" on page 569).

In addition to improving the quality of **gstack** results, breaking up recursive clusters makes it easier to determine accurate maximum instances values in cases where it would otherwise be difficult or impossible to do so. Providing accurate maximum instances information is a necessary step in preparing programs that use recursion for **gstack** (see "Annotating Recursive Functions" on page 570).

Using Strong Function Pointers

Calls made through `unknown` function pointers – pointers that do not have strong function pointer types – are represented in the call graph using references to a special `<<address_taken>>` node. This node has its own outgoing references that point to any function whose address is ever taken (except if it is only taken to assign it to a strong function pointer).

This allows **gstack** to correctly compute the maximum stack usage for programs that use function pointers, since there is a path in the call graph from each caller to any function that may be reached by its calls. However, since many of these paths reflect calls that cannot actually happen, use of unknown function pointers can cause **gstack** results to be very conservative and its call graph to contain spurious or overly complex recursive clusters. For example:

```
int funcTwo(void)
{
   return 0;
}

int (*fptrTwo)(void) = funcTwo;

int funcOne(void)
{
    return fptrTwo();
}

int (*fptrOne)(void) = funcOne;

int main()
{
  int x = fptrOne();
```

```
  return x;
}
```

Both `main()` and `funcOne()` call through unknown function pointers, so they each have a reference to the `<<address_taken>>` node. Because both `funcOne()` and `funcTwo()` have their addresses assigned to unknown function pointers, the `<<address_taken>>` node has its own references that point to them:

```
main
. <<address_taken>>
. . funcTwo
. . funcOne
. . . <<address_taken>>            [RPT]
```

This call graph allows **gstack** to correctly compute an upper bound on the stack usage of the program, because it contains paths from `main()` to `funcOne()` and from `funcOne()` to `funcTwo()`. However, it also contains a path from `funcOne()` to itself, which could cause the computed stack usage to be conservative, and which results in the following erroneous recursive cluster:

```
<<address_taken>>
. funcOne
. . <<address_taken>>            [RPT]
```

You can address this problem by using strong function pointer types, which are created by the `strong_fptr` attribute and provide a means of tracking the possible destinations of distinct function pointer calls separately from one another (for detailed information about this attribute, see `strong_fptr` on page 682).

When a function makes a call through a strong function pointer, the call graph contains a reference from the function to a node that represents the strong function pointer type. That node has its own outgoing references that point to any function whose address is assigned to a pointer of that type. The previous example can be updated to use strong function pointer types as follows:

```
typedef __attribute__((strong_fptr)) int (*FPTR_ONE)(void);
typedef __attribute__((strong_fptr)) int (*FPTR_TWO)(void);

int funcTwo(void)
{
   return 0;
}
```

```
FPTR_TWO fptrTwo = funcTwo;

int funcOne(void)
{
    return fptrTwo();
}

FPTR_ONE fptrOne = funcOne;

int main()
{
  int x = fptrOne();
  return x;
}
```

gstack now reports the call graph as follows:

```
main
. (FPTR_ONE *)
. . funcOne
. . . (FPTR_TWO *)
. . . . funcTwo
```

The graph still contains the paths necessary to correctly compute the stack usage, but the path from `funcOne()` to itself and the corresponding recursive cluster are eliminated.

Replacing unknown function pointers with strong function pointers is an important and powerful means of improving the accuracy of **gstack** results for your program. However, when updating your code you should take care to use separate strong function pointer types wherever possible, since sharing the same type between multiple pointers can result in the exact same problems associated with unknown function pointers. Note that if the previous change had used the same strong type, `FPTR_ONE`, for both pointers, the resulting call graph looks similar, but there is an `FPTR_ONE` node in place of the `<<address_taken>>` node:

```
main
. (FPTR_ONE *)
. . funcTwo
. . funcOne
. . . (FPTR_ONE *)            [RPT]
```

The graph also contains the recursive cluster:

```
(FPTR_ONE *)
. funcOne
. . (FPTR_ONE *)            [RPT]
```

By default, if **gstack** detects a call through an unknown function pointer, it lists the name of the function that makes the call in a section of its report titled `Problems During Execution` (see "The Problems During Execution Report Section" on page 577). To disable this behavior, pass the **--ignore_unknown_func_ptr_warnings** option.

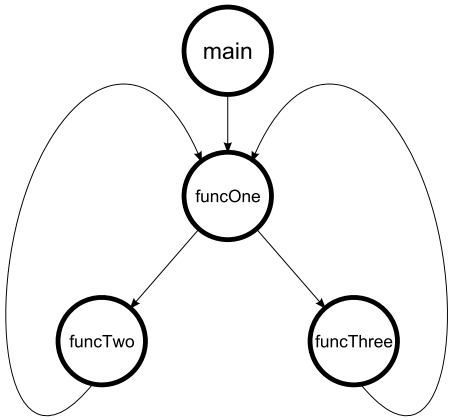
Re-Factoring Code to Simplify Recursive Clusters

In some situations, your code may include recursive clusters that can be simplified by re-factoring. The following example code contains a single recursive cluster consisting of `funcOne()`, `funcTwo()`, and `funcThree()`:

Code

```
int funcOne(int x);

int funcThree(int x)
{
  return 1 + funcOne(x-1);
}

int funcTwo(int x)
{
  return 1 + funcOne(x+1);
}

int funcOne(int x)
{
  if (x > 0)
    return funcThree(x);
  else if (x < 0)
    return funcTwo(x);
  else
    return 0;
}

int main(void) {
  int x = funcOne(5) + funcOne(-5);
  return x;
}
```

Diagram of Cluster



gstack Output

```
Recursive Cluster #0 (3 nodes):
funcOne
. funcThree
. . funcOne              [RPT]
. funcTwo
. . funcOne              [RPT]
```

Note

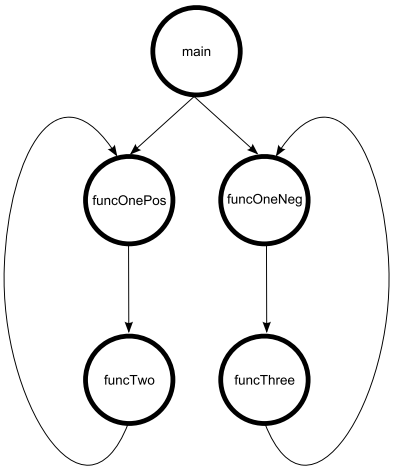
`[RPT]` indicates that the node is repeated. Its callees are not displayed again.

This code actually contains two separate recursive loops: `funcOne()`→`funcTwo()`→`funcOne()`, and `funcOne()`→`funcThree()`→`funcOne()`. You could rewrite it as follows to break the cluster up into two smaller clusters, each consisting of two nodes:

Code

```
int funcOnePos(int x);
int funcOneNeg(int x);

int funcThree(int x)
{
  return 1 + funcOnePos(x-1);
}

int funcTwo(int x)
{
  return 1 + funcOneNeg(x+1);
}

int funcOnePos(int x)
{
  if (x > 0)
    return funcThree(x);
  else
    return 0;
}

int funcOneNeg(int x)
{
  if (x < 0)
    return funcTwo(x);
  else
    return 0;
}

int main(void) {
  int x = funcOnePos(5) + funcOneNeg(-5);
  return x;
}
```

Diagram of Cluster



gstack Output

```
Recursive Cluster #0 (2 nodes):
funcOnePos
. funcThree
. . funcOnePos              [RPT]

Recursive Cluster #1 (2 nodes):
funcOneNeg
. funcTwo
. . funcOneNeg              [RPT]
```

Annotating Recursive Functions

In order to correctly compute the maximum stack usage for a program, **gstack** needs to know the maximum number of instances of each recursive function that can

appear on the stack at a given time. If it detects a recursive function for which a maximum instances value has not been specified and cannot be computed, **gstack** lists the function in the `Missing Information` section of its report with `MI`. To provide this information to **gstack**, add the following directive immediately before the definition of the function:

```
#pragma ghs max_instances(n)
```

or use the following command line option:

```
-max_instances func=n
```

Because **gstack** tries to infer a function's maximum instances value based on the values of its callers and callees, it may not be necessary to specify a value for every function in a particular recursive cluster. However, the values **gstack** computes may be conservative. Providing **gstack** with additional values could yield more accurate results. For example:

```
int funcOne(int x);

int funcThree(int x)
{
    return 3 + funcOne(x-1);
}

int funcTwo(int x)
{
    return 2 + funcOne(x-1);
}

int funcOne(int x)
{
    if (x <= 0)
        return 0;
    else if (x & 1)
        return funcTwo(x);
    else
        return funcThree(x);
}

int main (void){
    int x = funcOne(10);
    return x;
}
```

Specifying a maximum instances value of 11 for `funcOne()` would cause **gstack** to compute a maximum instance values of 11 for both `funcTwo()` and `funcThree()`. However, because neither of these functions exist on the stack more

than 5 times, specifying individual maximum instances values for each of them would lead to a lower computed maximum stack usage for the cluster.

Note

If a function is inlined its value is no longer used to compute the values for other nodes in the cluster, so building your program with different sets of options may require you to annotate more functions.

gstack computes the maximum stack usage for a recursive cluster by multiplying each function's individual stack usage by its maximum instances value, and summing the result over all the functions in the cluster. Therefore, large or complex recursive clusters can cause **gstack** results to become extremely conservative, and it is strongly recommended that you modify your code to simplify them as much as possible (see “Working With Recursive Clusters” on page 565).

Specifying Interrupts

If your program uses any feature that causes execution to be interrupted, such as interrupts or signals, and the handler uses the same stack as the main program, you need to provide this information to **gstack** to get accurate results.

Specify each of your program's interrupt handlers and associated priority levels as follows:

```
-interrupt_level func:levels
```

where *func* is the name of the handler and *levels* is a comma-separated list of priority levels at which the interrupts serviced by the handler can fire. Each entry in the list is a single level or a range of levels (denoted by a hyphen). For example, the following option specifies that `myFunc` is handler that services interrupts that can fire at priority levels 1 through 5, level 7, and levels 10 through 15:

```
-interrupt_level myFunc:1-5,7,10-15
```

If your program uses an interrupt dispatch function that calls off to the actual handlers, that function should be specified with the **-interrupt_entry_point** option. The maximum stack usage of the dispatch function is added to the total stack usage for the worst interrupt handler for each distinct priority level when computing the worst possible interrupt stack.

After specifying interrupt handlers, specify the maximum interrupt nesting depth, which is the number of interrupts that can fire on top of each other without their associated handlers being popped off of the stack. Specify the depth as follows:

`-interrupt_depth` *`depth`*

If you do not specify this option, **gstack** issues a warning and uses the number of distinct priority levels for which at least one handler is specified. To use this behavior and ignore the warning, use:

`-interrupt_depth` *`max`*

When computing the combination of nested interrupts that would use the most stack, **gstack** assumes that only one interrupt triggers for each priority level. Therefore, the worst possible interrupt stack consists of the *depth* distinct priority levels whose associated handlers have the largest total stack usages. The total amount of stack used by this combination of handlers is added to the maximum non-interrupt stack usage of each entry point when deciding whether it can exceed its allocated stack limit.

Use the **-i** option to tell **gstack** to include the `Interrupt Functions` section in its report, which lists all interrupt functions along with their stack usages and associated priority levels.

gstack Report Format

By default, **gstack** outputs a report that contains the results of its analysis and prints a message pointing to the location of the report. If you specify **-quiet**, **gstack** does not print a message unless it detects a problem that requires attention. If you pass the **-skip_report** option, no message is printed and no report is generated unless a problem is detected. If an error occurs before report generation begins, no report is generated.

Name	Appears When	Contains
`Executive Summary`	always	a high-level overview of the report
`STACK LIMIT EXCEEDED`	at least one entry point exceeds its provided stack limit	the name, worst stack usage, and specified stack limit for each entry point that exceeds its limit

Name	Appears When	Contains
`Problems During Execution`	**gstack** detects problems that could cause its results to be incorrect or conservative	an explanation of each problem
`Missing Information`	**gstack** detects missing information	details about what information is missing
`User Supplied Information`	**gstack** detects user-supplied information	details about what information was supplied by the user
`Unbounded Recursive Clusters`	**gstack** detects unbounded recursive clusters	a list of unbounded recursive clusters
`Bounded Recursive Clusters`	**gstack** detects bounded recursive clusters	a list of bounded recursive clusters
`Worst Call Stacks from Entry Points`	at least one entry point is specified	the worst stack usage and the call stack that yields it for each entry point
`Worst Interrupt Stacks for Entry Points`	at least one entry point has an interrupt stack with non-zero stack usage	the worst interrupt stack for each entry point that has one and the worst call stacks from the contributing interrupt handlers
`Interrupt Functions`	**-i** is specified	a listing of all interrupt functions, their stack sizes, and the priorities at which they can fire
`Max Stack Usages For All Nodes`	**-j** is specified	all nodes and their maximum stack usages
`Call Graph`	**-g** is specified	the call graph
`Nodes With No Callers`	**-u** is specified	all nodes that have no callers
`Callers from Procs`	**-c** is specified	information about callers of specific nodes
`Command Line`	always	the command line used to invoke **gstack**

gstack Options

-a *caller*:*callee1*[,*callee2*,...]	Adds connections to the call graph. `-a caller:callee1,callee2` adds calls from `caller` to `callee1` and `callee2` to the call graph.
-c *func*	Displays all callers of *func*.
-d *callee*[:*caller*,...]	If you specify one function, this option deletes all calls to that function. For example, `-d callee` deletes all calls to `callee`. If you also specify a list of callers, this option deletes all calls from those callers to the specified callee. For example, `-d callee:caller1,caller2` deletes all calls from `caller1` and `caller2` to `callee`.
-f *func*=*size*	Specifies function maximum stack usage *size*.
-g	Displays the call graph.
-help	Displays information about all options.
-entry_point *func*, **-e *func***	Specifies a function *func* to add to the set of entry points for the program. **gstack** displays information for each of these functions, including the call stack and the combination of nested interrupts that would use the most stack.
-entry_point_options *entry opt1* [*opt2...*] -+, **-e_opts *entry opt1* [*opt2...*] -+**	Specifies a list of options that apply to a single entry point. The list must begin with the name of the entry point and end with `-+`. The following options are allowed: **-stack_limit**, **-interrupt_depth**, and **-interrupt_level**. **-stack_limit** and **-interrupt_depth** options specified inside the list override those specified outside the list. **-interrupt_level** options specified inside the list do not override those specified outside the list.
@*file*, **-opt_file=*file***	Specifies a file containing additional options. For more information, see “Using a Driver Options File” on page 17.
--ignore_mismatched_prototype_warnings	Suppresses warnings about possible mismatches between the prototypes and definitions of certain library functions that take function pointer arguments, which can cause **gstack** results to be incorrect.

--ignore_unknown_func_ptr_warnings	Suppresses warnings about calls through unknown function pointers, which can cause **gstack**'s results to be conservative.
--ignore_wrap_warnings	Suppresses warnings about detected uses of the **-wrap** option, which can cause **gstack**'s results to be conservative.
-interrupt_entry_point ***func***	Specifies the dispatch function that calls the actual interrupt handlers. If this option is specified multiple times, only the last value for *func* is used.
-interrupt_level *func*:*m*-*n*[,*o*,*p*-*q*...]	Specifies functions that are interrupts, and the interrupt levels at which they can occur. You must specify at least one level or range of levels after *func*.
-interrupt_depth *depth*	Specifies the maximum depth of the interrupt stack.
-interrupt_funcs, **-i**	Displays all interrupt functions, their stack usages, and the priorities at which they can fire.
-j	Displays all functions and their maximum stack usages.
--legacy	Run **gstack** in legacy mode.
-max_instances ***func=n***	Specifies the maximum number of instances *n* of the function *func* that can exist on the stack at any given time. This option helps **gstack** compute the stack use of recursive clusters.
-o ***filename***	Specifies the name of the report output file.
-quiet	Hides the message pointing to the location of **gstack** output, unless a problem is detected.
-skip_report	Skips report generation unless a problem is detected.
-stack_limit ***n***	Specifies the maximum amount of stack in bytes that is allocated for each entry point. **gstack** reports an error if the stack limit for any entry point is exceeded.
-u	Displays all nodes that have no callers.
-x ***func=n***	Specifies an additional amount *n* to add to the known maximum stack value for function *func*. This option is the equivalent of `#pragma ghs extra_stack`.

Note

The **-a**, **-d**, and **-c** options may also be applied to nodes that represent strong function pointer types. If one of the names used with any of these functions could refer either to a function node or a type node, **gstack** issues an error. To resolve the ambiguity, prefix the name with either `.f.` or `.t.` to specify whether the option is to be applied to the function node or the type node, respectively.

Specifying C Static Functions

To specify C `static` functions to **gstack**, use the following syntax:

`funcname#filename`

If there is no other function in the program with the same name, you may omit `#`*`filename`*.

The Problems During Execution Report Section

This section explains warnings given for problems that may cause **gstack** results to be incorrect or conservative.

Misunderstood call table entries
: This warning is given when **gstack** is unable to load all of the debug info necessary to compute its results. This could be because the debug info has been corrupted, or because **gstack** is being run on a program that was built with a different version toolchain, which is not supported.

Use of __builtin_set_return_address() compiler intrinsic
: This warning is given when **gstack** detects use of the `__builtin_set_return_address` compiler intrinsic, which is not supported.

Mismatched prototype
: This warning is given when **gstack** detects a mismatch between the prototype and the definition for a library function that takes a strong function pointer as an argument. This issue means the call graph is incorrect, and should be addressed by making sure the prototype for the function comes from the appropriate header file, or by using the **-a** and **-d** options to modify the call graph appropriately. To suppress this warning, use the **--ignore_mismatched_prototype_warnings** option.

Use of -wrap option

This warning is given when **gstack** detects use of the **-wrap** linker option, which is handled safely but conservatively by assuming that all calls to function *foo* may also go to *__wrap_foo*, and all calls to *__real_foo* may also go to *foo*, if the respective nodes exist. When using **-wrap**, you may want to use the **-d** option for **gstack** to manually delete references from the call graph in order to achieve more accurate results. To suppress this warning, use the **--ignore_wrap_warnings** option.

Call(s) through unknown function pointer

This warning is given when **gstack** detects a call through an unknown function pointer, which can cause results to be extremely conservative and can lead to other problems, such as spurious recursions being added to the call graph. To suppress this warning, use the **--ignore_unknown_func_ptr_warnings** option. For more information, see "Using Strong Function Pointers" on page 566.

Interrupt Nesting Depth not specified

This warning is given when the **-interrupt_level** option is used to specify interrupt handlers, without the **-interrupt_depth** option also being used to specify a maximum interrupt nesting depth, which can cause results to be conservative. To tell **gstack** not to use a maximum nesting depth in its calculations without issuing the warning, use **-interrupt_depth max**.

Caveats

- **gstack** does not account for changes to the call graph made by third-party linkers (such as **ld** or **link**).
- **gstack** is not supported for programs that contain code built with third-party tools, including any libraries provided with a third-party operating system.
- **gstack** is not supported for programs that were built with a different version of the toolchain.
- **gstack** does not support the use of the GNU C/C++ extension that allows you to take the address of a statement label.
- **gstack** may not provide accurate results for programs built with linker optimizations such as **-delete** or **-codefactor**.
- Using Green Hills C++ libraries may result in errors about missing information and unbounded recursive clusters, or warnings about unknown function pointers.

The gstrip Utility Program

The **gstrip** utility program can remove line number, symbol table, and debugging information from an executable file to reduce its file size on disk. **gstrip** can only remove DWARF line number and debugging information. Since it does not modify Green Hills debugging information, stripping an executable does not generally affect debugging. However, it can impact some operations, such as reading from a section whose offset within the file has changed as a result of earlier sections being stripped out.

The **gstrip** utility processes executable files created by the Green Hills Software linker as well as those created by some native linkers.

The syntax for this utility is as follows:

gstrip [*options*] *filename*

where *filename* is the name of an executable file.

If no options are specified, **gstrip** removes line number, symbol table, and debugging information. The *options* for **gstrip** are as follows:

-help Displays information about all the options.
-l Removes line number information only, without stripping the symbol table or debugging information.
-prefixed_msgs **-no_prefixed_msgs** Enables or disables the insertion of the words `WARNING` and `ERROR` before every warning or error message.
-V Displays the version number of **gstrip** to standard error output.
-x Does not remove the symbol table; debugging and line number information might be removed.

The gversion Utility Program

The **gversion** utility program extracts and prints date and time information from an executable file provided by Green Hills. If no options or slots are specified, then **gversion** outputs the executable's revision and release dates from slots `0` and `6`.

The syntax for this utility is as follows:

gversion [*options*] [*slot*] [*file1*] [*file2* ...]

where:

- *options*: Listed in the table below.
- *slot*: A single digit number, specifying the slot to print (see Example 11.31. Using gversion on page 581). **gversion** displays the file modification date when no valid time stamp corresponding to the slot requested can be found.
- *file1*, *file2*: The executable files for which you want to print date and time information.

-all Displays all non-zero dates, marked `[0]` through `[9]`. Each date is preceded by a digit in square brackets "[]" called a time stamp. The revision date is preceded by `[0]` and the release date is preceded by `[6]`. See Example 11.32. Using -all on page 581.
-date (default) Displays the value as a date string.
-help Displays information about all options.
-mtime **-nomtime** (default) Enables or disables display of the file's modification date. The **-mtime** option must be used with **-all**. See Example 11.33. Using -mtime on page 581.
-quiet Suppresses errors for files that are not stamped.
-value Displays all stamp values without converting to date.

Example 11.31. Using gversion

In this example, suppose the executable **gversion** has only time slot `0` set:

```
> ./gversion gversion
gversion: Green Hills Software, MULTI v2012.0
gversion:                       Revision Date Mon Nov  9 10:48:25 PST 2009
gversion:                       Release Date  Mon Dec 14 17:08:14 PST 2009
```

If asked to print the value of slot 1, which is not stamped, **gversion** displays an error message:

```
> ./gversion 1 gversion
gversion: Green Hills Software, MULTI v2012.0
gversion has not been time stamped
gversion:                       [m] Mon Nov  9 10:48:25 2009
```

Example 11.32. Using -all

When using `-all`, **gversion** displays all non-empty time stamps.

```
> ./gversion -all gversion
gversion: Green Hills Software, MULTI v2012.0
gversion:                       [0] Mon Nov  9 10:48:25 PST 2009
gversion:                       [6] Mon Dec 14 17:08:14 PST 2009
gversion: License Managed by Green Hills License Manager
```

Example 11.33. Using -mtime

The **-mtime** option must be used with **-all**. It provides the file's modification date.

```
> ./gversion -mtime -all gversion
gversion: Green Hills Software, MULTI v2012.0
gversion:                       [0] Mon Nov  9 10:48:25 PST 2009
gversion:                       [6] Mon Dec 14 17:08:14 PST 2009
gversion: License Managed by Green Hills License Manager
gversion:                       [m] Mon Nov  9 10:48:25 2009
```

The gwhat Utility Program

The **gwhat** utility program performs a similar function to the Linux/Solaris **what** program. It reports or updates the version information of a file.

The syntax for this utility is as follows:

gwhat [*options*] [*file1*] [*file2 ...*]

The options to **gwhat** are as follows:

-all Prints out all `what`-strings.
-c *what_str* Overwrites the default `what`-string space reserved for customers with *what_str*.
-f Overwrites the `what`-string, even if it is read-only.
-help Displays help message.
-m *what_str* Overwrites the Green Hills `what`-string space with *what_str*.
-s*n* Stops after the *nth* occurrence of the pattern. The default is 1 (first occurrence). There is no space between the **-s** and *n*.
-w *what_str* Changes the `what`-string marker from `@(#)` to *what_str*.

If neither the **-c** nor **-m** option is specified, then by default **gwhat** outputs the first string in the file. You can set the number of strings to output with the **-all** or **-s** option.

Version Extract Mode

In version extract mode, **gwhat** searches each filename for occurrences of the following 4-byte `what`-string marker:

```
@(#)
```

It then displays all subsequent characters up to but not including any of the following terminator characters (expressed in C) or end-of-file:

Character	Character name
"	Straight double quotation mark
>	Greater than sign
\\	Backslash
\n	Newline
\0	Null character

For example, if the C program in file **program.c** contains:

```
char sccsid[] = "@(#)Version 3.0";
```

and if **program.c** is compiled to yield **program.o** and linked to yield **a.out**, then the command:

```
gwhat program.c program.o a.out
```

produces:

```
program.c:
     Version 3.0
program.o:
     Version 3.0
a.out:
     Version 3.0
```

Version Update Mode

In version update mode, **gwhat** can write new version information into a file. This feature has several applications. All Green Hills executables are built with two default `what`-strings. For example:

```
"@(#)Green Hills Software, Version 201200"

"@(#)[Customer version stamp space]"
```

Under some circumstances, Green Hills may change the Green Hills release `what`-string, using the **-m** option. For example:

```
gwhat -m 'special release abc' elxr
gwhat -s2 elxr
        Green Hills Software, special release abc
        [Customer version stamp space]
```

This capability is available to Green Hills customers as well, although it is recommended that the Green Hills release `what`-string be left unchanged.

If you want to stamp a Green Hills executable with a private identification mark, change the `[Customer version stamp space]` that is provided for this purpose. For example:

```
gwhat -c 'Mary likes this version' elxr
gwhat -s2 elxr
        Green Hills Software, Version 201200
        Mary likes this version
```

The **-c** option can only be used once on a binary because it looks for the original `[Customer version stamp space]` identifier.

You might want to build a custom `what`-string into your own software to mark the executables. For example, suppose your executable, **acmeprog**, contains a C source file that contains:

```
char sccsid[] = "@(#)ACME SOFTWARE         ";
```

The following shows the different outputs of **gwhat** before and after using the **-m** and **-w** options.

```
gwhat acmeprog
        ACME SOFTWARE
gwhat -w "@(#)ACME SOFTWARE " -m "version 12" acmeprog
gwhat acmeprog
        ACME SOFTWARE version 12
```

The mevundump Utility Program

The **mevundump** utility program converts (and merges, if multiple input files are given) an input event log of any format to **mevgui**'s event log format.

The syntax for this utility is as follows:

mevundump *options InputConfig1 InputEventLog1* [*InputConfig2 InputEventLog2...*] *OutputName*

The options to **mevundump** are as follows:

-h Prints out usage dialogue.
-d Enables extra debugging info and warnings to be output to stderr.
**-a *N*	*H*** `N` - Natural alignment. The original timestamps in each input event log are used so if one log begins at 500 ms and another begins at 1200 ms, the event at time 500 ms will be considered 0 time for all event logs (so the events from the second log will begin 700 ms after the first event in the output log). If no option is given, **mevundump** defaults to this behavior. `H` - Head alignment. The timestamps in each input event log are justified against the first event of that log. If one log's events start at time 500 ms and another starts at time 1200 ms, the first event from both logs will appear at time 0 in the output event log.
-confDir *dir_path* Designates the directory in which to look for the input configuration file(s).
-logDir *dir_path* Designates the directory in which to look for the input event log file(s).
-outConfDir *dir_path* Designates the directory in which to create the output **mevgui** configuration file(s).
-outLogDir *dir_path* Designates the directory in which to create the output event log files.

mevundump is a stand alone conversion tool for event logs. It accepts, as input, up to 16 pairs of input configuration files and event logs (in their native format) and a single output name and produces a **mevgui** compatible event log triplet of files (**OutputName.mes**, **.mei**, and **.mev**) and a **mevgui** compatible configuration

file (**OutputName.mc**). If more than one input pair is given, the inputs are merged according to the event alignment option given on the command line.

There are some differences between input and output event logs to be aware of:

- The timestamps will change. In both timestamp alignment options above, the timestamps will be justified to some event. Unless the event they are being aligned with has a timestamp of 0, the timestamps will change by a difference of the aligning event's timestamp.
- The task IDs will change. Each task ID will be shifted left by 4 bits and have a process ID masked in to it. The process ID denotes which input log the event corresponds to. If only one input log is given, this process ID will always be 0. Also note that because of the shift, the top 4 bits of the task ID will be lost (currently uses a 32 bit `int` to store/report task ID).
- The specific values used for each event may change. These output values/codes are generated based on the order the events are presented in the input configuration file. Note: this does not affect how the input file is read, this only affects the output event log and output configuration file.
- The values of statuses will change. Like task IDs, statuses will have a process ID shifted in to their value. This feature allows statuses of the same value to be treated differently depending on which file they come from.

Again, the top 4 bits will be lost in the shift (currently using 32 bit `int`).

These differences do not affect the visual output of the logs, nor the order of the events on the time line, but the user should be aware of them if trying to find specific information in the log, such as a task or status by its value in the input log. The best way to find the output values of events and statuses is to open the output configuration file and check there.

Input Configuration File

The input configuration file describes how the input event log should be interpreted by **mevundump**. An input configuration file is made up of 3 main sections

- a single *Header* block
- one or more *Event* blocks
- 0 or more *Status* blocks

Each of these blocks has its own list of options and sub-blocks, which are described below. The *Header* block contains general information about how to read the input event log and where specific information may be found in the file. Each *Event* block describes where to find information for that event and the formatting to be output to the output configuration file. Each *Status* block describes the configuration information that will describe the status in the output configuration file. If no *Status* blocks are given, the status list will be generated from the event's *Status_Change* sub-blocks using a rotating color scheme and default values.

The table below outlines the options that can be set in this file. Any values with a vertical bar between them (`|`) imply that those values should be used literally (i.e. `'endian: B|L'` means either a literal `B` or `L` should be used as the value). A value of `#` implies any number can be given, as decimal or (with a leading `0x`) as hexadecimal. A value of '`OFFSET`' can be written as either a number (in decimal or hex, if led with an `0x`) or as a file offset to be read to find the value. This offset should be led with a `*`. For example, to use the value `32`, you can write `32` or `0x20`. To use the offset `33` you would write `*33` or `*0x21`. All offsets are counted in bytes from the beginning of the file, unless otherwise noted. **mevundump** will read `addr_size` bytes, beginning at the offset, using the given endianness, as the value for the given option.

For `OFFSET` type values, think of it as an address to dereference. For example, something like `version: OFFSET`, which expects a number, would dereference the given `OFFSET` and read the file from there to get the version # (think `int*`). However, a field like `event_list_head: OFFSET`, which expects an offset to the first event, would dereference the given `OFFSET`, read the value at `OFFSET`, and use that value as the file offset to the first event (`void**`).

Header Block

General setup options

Option	Default Value	Comments
`endian: B|L`	`B`	Indicates what endianness the input event log should be read in. All output event logs will be in big endian format.
`version: OFFSET`	`#define CURRENT_VERSION (1)`	Indicates what version of **mevundump** this input configuration is built for.
`addr_size: #`	`4 bytes`	The number of bytes used to represent an address in this file. Also will be used as the # of bytes to read for a majority of OFFSET values.
`timer_tick_freq: OFFSET`	`100000`	Default value is used by uVelOSity event logs.
`timestamp_format: U|L`	`U (Upper)`	Indicates which word of a 64 bit value comes first (Upper or Lower). For example a timestamp of `0x1234 5678 abcd cdef` in Lower format would be written as `0xabcd cdef 1234 5678`
`filetype bin|text`	`bin(ary)`	Indicates whether the event log is a binary or text file.
`event_list_head: OFFSET`	Mandatory	Offset to the first event in the log, chronologically.
`event_list_tail: OFFSET`	`event_list_head`	Offset to the last event in the log, chronologically.
`event_pool: OFFSET`	`0`	Offset to the start of the event list (by location in file). This is necessary for circular buffers, so that the front of the list can be found and wrapped after reaching the end.

Option	Default Value	Comments
`event_header_start: OFFSET`	`0`	The offset to the start of the header block of information in the original memory block. For uVelOSity logs, this is necessary because the other offsets (head, tail, etc.) are all just addresses to memory of the running program. When saved to a log file, they must be changed to be relative to the file, which was output from the memory beginning at `event_header_start`. If not given, **mevundump** will attempt to calculate it based on other information, like expected `task_count` and `task_chunk_size` (assumes event list comes immediately after task list, like in a ThreadX log)
`event_list_size: OFFSET`	`0`	Total # of bytes used for the event list. Especially important for circular buffers, as this tells **mevundump** when to return to the beginning (`event_pool`).
`event_chunk_size: OFFSET`	`0`	The # of bytes for every event in the log. Use this if all of the events have the same size, or all events store their size at the same location. Otherwise, each event can define its own size.
`event_chunk_size_size: #`	`addr_size`	# of bytes used to store the event chunk size.
`event_type_offset: #`	Mandatory	Offset from the start of each event 'chunk' to the type value. This value should be the event identifier (the `native_type` field of each event).
`event_type_size: OFFSET`	`addr_size`	# of bytes used to store the event type.
`task_input: 1|2|3`	`3`	Tells **mevundump** how to read in the task list. 1 - A task list will be given in the configuration file (see next option). 2 - A task list is given in the event log, and the options to read it will follow (see next several options). 3 - Tasks will be reported on the fly through special events. In this mode, any new task ID discovered will be considered a new task.

Option	Default Value	Comments
task_list: String	NULL - mandatory for task_input: 1 mode	Describes the executing tasks in the event log. The list is semicolon delimited and should follow this format: TaskName1, TaskID1, TaskPriority1; TaskName2, TaskID2, ...
task_list_offset: OFFSET	0 - mandatory for task_input: 2 mode	Offset from start of file to the first byte of the task list.
task_size: OFFSET	0 - mandatory for task_input: 2 mode	# of bytes used for each task.
task_count: OFFSET	0 - mandatory for task_input: 2 mode	The expected number of tasks to be read in.
task_count_size: #	addr_size	# of bytes used to store task_count.
valid_offset: OFFSET	0	Offset to any kind of valid flag to know if the current task should be used. If not given, every task read will be assumed to be valid.
id_offset: OFFSET	0 - mandatory for task_input: 2 mode	Offset within each task 'chunk' to find the task ID.
id_size: OFFSET	addr_size	# of bytes used to store the task ID.
priority_offset: OFFSET	0 - mandatory for task_input: 2 mode	Offset within each task 'chunk' to find the task priority.
priority_size: OFFSET	addr_size	# of bytes used to store the task priority.
name_offset: OFFSET	0 - mandatory for task_input: 2 mode	Offset within each task 'chunk' to find the task name.
name_len: OFFSET	32 chars - mandatory for task_input: 2 mode	Length of each task name in bytes.
creation_type: #	-1 - mandatory for task_input: 3 mode	The event type code for task creation events.

Event Block

Event specific options and information. One block per event type.

Option	Default Value	Comments
`native_type: #`	Mandatory	The unique identifier for this event type.
`size: OFFSET`	`event_chunk_size`	# of bytes used by this event type.
`subtype_offset: #`	Mandatory	# of bytes from the start of the event chunk to the subtype.
`subtype_size: #`	`2`	# of bytes used by this event subtype. NOTE: currently only 2 bytes are supported, as **mevgui** only accepts 2 bytes for event type and subtype.
`taskID_offset: #`	Mandatory	# of bytes from the start of the event chunk to the task ID responsible for the event.
`taskID_size: #`	`addr_size`	# of bytes used by the task ID.
`timestamp_offset: #`	`0`	# of bytes from the start of the event chunk to the timestamp.
`timestamp_size: #`	`8`	# of bytes used by this event timestamp.
`task_change: true|false`	`false`	Indicates this is a special task change event (sets the associated task's status as executing).
`name_change: true|false`	`false`	Indicates this is a special name change event (often used as a synthetic refresh to make **mevgui** aware of a new task being created or used to change task priority).
`halt_others: true|false`	`false`	Indicates this task halts the execution of all other tasks when it runs.

Status_Change Sub-Block

If an event or event subtype contains this sub-block, it implies that the event/subtype is a status change event. Also, the `status` sub-sub-blocks contains within it can be used to generate a full status list, using a rotating color scheme and default values. Also, at least one status sub-sub-blocks must be present in each `status_change` sub-block.

Option	Default Value	Comments
`code_offset: #`	`0`	# of bytes from the start of the event to the status code this event changes its task to. If not given, this event will always switch to the first `status` sub-sub-block contained within this `status_change` block.

Status Sub-Block

This sub-sub-block is essentially a reference to the status blocks described below. If this sub-sub-block describes as status that does not have a corresponding status block, **mevundump** will automatically generate a status block for it, using default values and a rotating color scheme.

Option	Default Value	Comments
`code: # | *`	Mandatory	The value that identifies this status, or * to describe all unmatched statuses.
`disp_name: String`	Mandatory	The name that identifies this status and will be displayed by **mevgui**
`visible: true|false`	`false`	If true, the associated icon (described in the status block) will be displayed by **mevgui**. This is only useful for an auto-generated list, as the main status block's visible: option will override this one.

Extra Sub-Block

This sub-block described the extra data fields of each event and how they should be reported to **mevgui**.

Option	Default Value	Comments
`name: String`	NULL - Mandatory	The name of this extra data field as **mevgui** will report it
`name: String`	NULL - Mandatory	The name of this extra data field as **mevgui** will report it
`name: String`	NULL - Mandatory	The name of this extra data field as **mevgui** will report it

Subtype Sub-Block

This sub-block described the extra data fields of each event and how they should be reported to **mevgui**. Each event must have at least one subtype (that one may be a `*` subtype, to capture all subtypes of the given event).

Option	Default Value	Comments
`subtype: # | *`	`*`	The native value of this subtype, or `*` to match all unmatched subtypes of the parent event.
`disp_name: String`	NULL - Mandatory	The name to display in **mevgui** for this subtype of event.
`bitmap_name: String`	`disp_name`	The name of the bitmap icon to use for this event in **mevgui**.
`extra: String`	`NULL`	A `printf` format string to use in place of the parent event's extra data output. This allows subtypes to display the extra data of the event differently, or (if an empty string is given) to not display any extra data, even if the parent event collects it. If not given, the parent event's extra data will be used.
`category: String`	`Default`	Describes which event category this subtype belongs to. Subtypes will be grouped by category in the output **mevgui** configuration file.
`visible: true|false`	`true`	Describes whether or not this subtype's icon will be displayed by **mevgui**.

Option	Default Value	Comments
`task_change: true|false`	Inherits from parent `Event`	Indicates this is a special task change event (sets the associated task's status as executing).
`name_change: true|false`	Inherits from parent `Event`	Indicates this is a special name change event (often used as a synthetic refresh to make **mevgui** aware of a new task being created or used to change task priority).
`halt_others: true|false`	Inherits from parent `Event`	Indicates this task halts the execution of all other tasks when it runs.
status_change{}	`N/A`	A `status_change` block within a subtype is formatted the same as one in an event. See above for reference.

Status Block

Gives a full description of a task status, including all configuration info for **mevgui**.

Option	Default Value	Comments
`code: # | *`	`*`	The value that identifies this status, or * to describe all unmatched statuses.
`disp_name: String`	NULL - Mandatory	The name **mevgui** will display for this status.
`bitmap_name: String`	`disp_name` (with any non-alphanumeric characters changed to an _)	The name of the bitmap file to display for the event of the actual status change. Usually, just "statuschange" and the icon's visibility is set to false (see below).
`rgb: #`	`0x000000 (Black)`	The color to use for the status's line segment.
`style: -|.|s`	`s (Solid)`	The style of the line to use for the status's line segment. - = Dashed, . = Dotted, s = Solid.
`thickness: #`	`1`	The thickness of the line to use for the status's line segment. Note, for easy visibility, auto-generated statuses use a thickness of 2 (not the default value).
`visible: true|false`	`true`	Describes whether or not this status change's icon will be displayed by **mevgui**. This is for the actual event during which the status changes, this does not affect the visibility of the line segment representing the status.
`type: exec|ready|init`	`NULL (Not a special status)`	This will mark the status as a special case. The three special cases are used as follows: `exec` - This is the status a task will change to whenever there is a context switch to it. `ready` - This is the status all other tasks will be set to when a "`halt_others`" context switch occurs. `init` - This is the initial status all new tasks are started in. This often appears as "Unknown". If the status is not a special status, do not include this option. If any of the three special statuses are never defined in the input configuration file, the default values are `0x0` for `ready`, `0x1` for `exec`, and `0x2` for `init`.

Input Configuration Example

The following is an excerpt from the `uVelOSity` input configuration file.

```
header{
 os: uVelosity
 event_list_head: *48
 event_list_tail: *44
 event_type_offset: 1
 event_pool: *0x1164
 event_header_start: *36
 event_list_size: *32
 timestamp_format: L
 event_chunk_size: *0
 event_chunk_size_size: 1
 task_input: 1     #task_list must come next
 task_list: task_1, 0xbe98, 0xa; task_2, 0xbf28, 0xb; Interrupts, 0, 0;
}
event{    #refresh
 native_type: 0x6
 subtype_offset: 0x3
 subtype_size: 2
 taskID_offset: 13
 timestamp_offset: 0x5
 timestamp_size: 8
 name_change: true
 subtype{
  subtype: 0xa
  disp_name: TaskNameRefresh
  bitmap_name: statuschange
  visible: false
  category: uv_mpool
  extra:
  status_change{
   status{
    code: 0xa
    name: Unknown
   }
  }
 }
 subtype{ #catch any other type 0x6 events
  disp_name: TaskNameRefresh
  bitmap_name: statuschange
  category: uv_mpool
  extra:
 }
}
status{
 code: 0xa
 disp_name: Unknown
 bitmap_name: statuschange
 style: -
 type: init
}
```

mevundump_lib RPC Library

The functions of **mevundump** can also be accessed through a remote procedure call shared object, **mevundump_lib.so**. This library provides the same functionality as the C implementation (full translation and merging of event logs) but also exposes pieces of **mevundump** so that it can be extended easily.

The functions available and a brief description of each can be found in the tables below. Each function accepts a `JsValue` structure with the given key/value pairs added to it. Each function also returns a `JsValue` structure with the given key/value pairs defined in it. If the function called fails, `err` will be set to a value other than 0, and `errMsg` will contain the error message, otherwise `err` will be set to 0.

Input Configuration Reading Functions

Function	Description
`ParseConfigFile(filename=(char*), cDir=(char*), evBase=(int), procInd=(int))` returns `(configIndex=(int), evCount=(int), err=(int), errMsg=(char*))`	Parses the given configuration file (at cDir/filename) and stores the returned `MevConfigData` structure to an internal list. On success, it returns the index (in the internal list) of the new `MevConfigData`, and the `evCount` - the number of events parsed in the new `MevConfigData`.
`ClearConfigList()` returns `(Empty JsValue)`	This will empty the internal list of `MevConfigData` structures.

Event Log Writer Functions

Function	Description
`OpenMESFile(filename=(char*), oDir=(char*), osName=(char*), freq=(int), addrSize=(int))` returns `(err=(int), errMsg=(char*))`	Creates and initializes a new output event log.
`CloseMESFile()` returns `(Empty JsValue)`	Flushes and Closes the output event log.
`OutputTask(id=(int64), idS=(int), priority=(int64), prioS=(int), name=(char*), nameLen=(int), refresh=(int), init=(int), bigEndian=(bool), pNameLen=(int), procName=(char*), pInd=(int))` returns `(err=(int), errMsg=(char*))`	Writes a task creation event to the output event log using the given information. For clarification, "`id`" is the task ID, "`priority`" is the task priority, "`name`" is the task name, "`refresh`" is the code for a task creation/refresh event, "`init`" is the status code to start the new task in. "`idS`", and "`prioS`" is the number of bytes to be used for "`id`" and "`priority`", respectively. However, **mevundump** currently only supports 4 byte task IDs. "`pNameLen`", "`pInd`", and "`procName`" should only be used if this task needs process information associated with it (i.e. when merging multiple input event logs). "`pInd`" is the process index for this task's process, "`procName`" is the name of the process, and "`pNameLen`" is the length of `procName`. If including process information with this task, `pNameLen` must be > 0, otherwise it will be ignored.

Function	Description
`OutputEvent(time=(int64), type=(int), subtype=(int), buffer=(int[]), procInd=(int), execStat=(int), readyStat=(int), stat=(bool), task=(bool), name=(bool), halt=(bool))` returns `(err=(int), errMsg=(char*))`	Writes an event to the output event log using the provided information. "`time`" is the timestamp of the event, "`type`" is the output type code of the event, "`subtype`" is the subtype of the event, "`buffer`" is a byte array containing the extra data information, "`procInd`" is the process number that is executing this event, "`execStat`" is the status code for an executing task, "`readyStat`" is the status code for a task that is ready to execute, "`stat`", "`task`", "`name`", and "`halt`" describe the effect this event has on the status of the system - each would be true if the event is a status change, task change, name change (refresh), or if this event halts all other running tasks in its process.
`TranslateEvent(buffer=(int[]), proc=(int), bigEndian=(bool), LLFormat=(bool), typeOff=(int), subtypeOff=(int), timestampOff=(int), taskIDOff=(int), typeSize=(int), subtypeSize=(int), timestampSize=(int), taskIDSize=(int), edList=(offset=(int), size=(int)))` returns `(err=(int), errMsg=(char*), type=(int64), subtype=(int64), taskID=(int64), timestamp=(int64), extra=(int[]))`	Converts *buffer*, an array of bytes, into event data. "`proc`" is the process number associated with this byte array, "`bigEndian`" is the endianness of the byte array, "`LLFormat`" is the word order for a 64 bit value (true if the lower word comes first), "`typeOff`", "`subtypeOff`", "`timestampOff`", and "`taskIDOff`" are byte offsets into *buffer* to find the given information and the corresponding *Size values are the number of bytes each piece of information is stored in. "`edList`" is the list of extra data configuration values - each one has an "`offset`" and a "`size`".
`ConvertAndMerge(memNames=(char*[]), lDir=(char*), output=(char*), outputDir=(char*), align=(int))` returns `(err=(int), errMsg=(char*))`	Takes "`memNames`", a list of input event log filenames, and "`output`", the name of the output event log file (and their respective directory paths), and converts and merges the input event logs into the output event log using the internal `MevConfigData` list. Calls to this function assume that `ParseConfig` has already been run on each associated input configuration file.

Output Configuration Writer Functions

Function	Description
`WriteFullConfig(filename=(char*), oDir=(char*))` returns `(err=(int), errMsg=(char*))`	Writes the configuration information in the internal `MevConfigData` list to the given output configuration file. This function will open the output configuration file (No need to open it beforehand) but does not flush/close the file, so the user can add more after this call completes.
`OpenMEVConfig(filename=(char*), oDir=(char*))` returns `(err=(int), errMsg=(char*))`	Opens the given output configuration file for writing. Also, writes in the default `MEV_Misc` settings.
`CloseMEVConfig()` returns `(Empty JsValue)`	Flushes and closes the current output configuration file. Note: only one file can be open at a time. If the users attempts to open multiple files, all except the first open will be silently ignored.
`WriteConfigLine(text=(char*))` returns `(err=(int), errMsg=(char*))`	Writes a line of text to the configuration file, appending a new line afterwards.
`WriteConfigStatus(code=(char*), name=(char*), rgb=(int), style=(int), thickness=(int), visibility=(bool))` returns `(err=(int), errMsg=(char*))`	Writes a `MEV_Status` to the configuration file using the given information. "`rgb`" is a hex RGB color and for "`style`", 0 = Solid, 1 = Dashed, 2 = Dotted lines.
`WriteConfigEventCategory(name=(char*))` returns `(err=(int), errMsg=(char*))`	Writes a `MEV_Event_Category` to the configuration file. All following events (until the next `MEV_Event_Category`) will be included in this category.
`WriteConfigEvent(type=(int), subtype=(int), name=(char*), bitmap=(char*), extra=(char*), visibility=(bool))` returns `(err=(int), errMsg=(char*))`	Writes a `MEV_Event` to the configuration file. "`name`" is the name that **mevgui** will display for the given event, "`bitmap`" is the name of the .bmp file to use as the icon for the event (do not include file extension), "`extra`" is the extra data format string.
`WriteConfigSEvent(type=(int), subtype=(char*), name=(char*), bitmap=(char*), extra=(char*), visibility=(bool))` returns `(err=(int), errMsg=(char*))`	Writes a `MEV_Event` to the configuration file. This function is the same as **WriteConfigEvent**, except that it accepts a string for subtype instead of an int.

Function	Description
`WriteConfigArrEvent(type=(int), subtype=(int), name=(char*), bitmap=(char*), extra=(size=(int), name=(char*)), visibility=(bool))` returns `(err=(int), errMsg=(char*))`	Writes a `MEV_Event` to the configuration file. This function is the same as **WriteConfigEvent**, except that it accepts an array of extra data information: "`size`" is the number of digits to use for the extra data value and "`name`" is the name it should be marked as. For example, if `size=4` and `name="PC"`, the resulting event would have an extra data format string of `"PC=%4X"`.

Python RPC Example

The next page is a Python implementation of **mevundump**, using the RPC library. It accepts the same arguments as **mevundump**. Remember to import the **mevundump_lib** shared object:

```
#load the MEV shared object library
import shlibproxy
proxy = shlibproxy.SharedLibraryServiceProxy(
 "/export/devl/linux86-deploy/mevundump_lib.so",
 "InitMevUndump",
 "MevUndumpDispatch")
```

```
#!/usr/bin/env python
# pymev.py: Python library to access mevundump functions

import sys
#load the MEV shared object library
import shlibproxy
proxy = shlibproxy.SharedLibraryServiceProxy(
 "/export/devl/linux86-deploy/mevundump_lib.so",
 "InitMevUndump",
 "MevUndumpDispatch")

# parse command line, initialize vars, etc. not shown
######################### parse in input config(s) ########################
evCount = 0
for i in range(0, len(configNames)):
 rv = proxy.ParseConfigFile(filename=configNames[i], \
     cDir=confDir, evBase=evCount)
 print configNames[i]
 print confDir
 if (rv["err"] == 0): #success
  evCount += rv["evCount"]
  print rv["errMsg"]
 else:
  print "Error processesing " + configNames[i]
  print "Error Code: %d\nErrorMsg : %s\n" % \
      (rv["err"], rv["errMsg"])
  #delete any bad ones
  del configNames[i]
  del logNames[i]
if (len(logNames) <= 0):
 sys.exit("Did not successfully parse any configuration files.")

###################### merge input to output logs ########################
rv = proxy.ConvertAndMerge(lDir=logDir, output=outputName, \
  outputDir=outDir, align=alignIn, memNames=logNames);
if (rv["err"] != 0): #something went wrong
 print "Error converting memfiles."
 print "Error Code: %d\nErrorMsg : %s\n" % \
      (rv["err"], rv["errMsg"])
 sys.exit("Check files and try again.\n");
print rv["errMsg"]

######################### write mev config file ##########################
rv = proxy.WriteFullConfig(filename=outputName, oDir=outCDir);
if (rv["err"] != 0): #something went wrong
 print "Error writing MEV config file " + outputName
 print "Error Code: %d\nErrorMsg : %s\n" % \
      (rv["err"], rv["errMsg"])
 sys.exit("Check files and try again.\n");
print rv["errMsg"]

proxy.CloseMEVConfig()
proxy.ClearConfigList()
```

AUTOSAR and OSEK Operating System Awareness

AUTOSAR and OSEK operating system awareness enables the MULTI IDE debugger to present information specific to your operating system implementation in the debugger, and to be aware of operating system elements such as threads and other resources. This awareness is enabled using an industry-standard ORTI (OSEK Run Time Interface) file typically provided by your operating system vendor. With that in mind, this documentation assumes that you have either written your own standards compliant ORTI file or have obtained one from your operating system vendor or its associated configuration software. This documentation is not an instruction manual on writing an ORTI file. If you are interested in writing your own ORTI file, please refer to the ORTI documentation easily found online.

This documentation is broken into five distinct sections. The first two sections (Compiling the ORTI File and Launching the ORTI OSA Explorer) below describe compiling an ORTI file, and using the result in the ORTI OSA Explorer inside the MULTI IDE debugger. These sections are designed to be a quick start reference for readers looking to get started using the operating system awareness features quickly. The next two sections (**ccorti** Reference and ORTI OSA Explorer Reference) provide a comprehensive reference to both the ORTI compiler (**ccorti**) and the ORTI OSA Explorer. These sections serve as both a convenient command reference, as well as documentation on how to overcome limitations that may be encountered with the ORTI file provided by the operating system vendor. The final section (ORTI Troubleshooting) expands upon this by providing a general troubleshooting guide for frequently encountered issues.

Using AUTOSAR and OSEK operating system awareness with the Green Hills MULTI IDE debugger requires two separate steps:

- First, the user's ORTI file(s) must be compiled prior to invoking the MULTI IDE.
- And then finally the MULTI IDE can be used to debug OS(s) that match the information in the ORTI file(s), using the ORTI OSAExplorer.

An ORTI file is necessary to use the AUTOSARand OSEK operating system awareness. These files describe the basic properties of a simple operating system executive in a standardized format. In particular they provide details on context information (where to find registers for non-running tasks), message containers, alarms, etc. as well as vendor-specific extensions.

It is not uncommon for an operating system vendor to not provide sufficient information to visualize all operating system objects, or perform all operating system awareness tasks (such as retrieving register values for non-running tasks). Several methods to overcome this are described in the documentation below, and the reader may contact Green Hills support for additional assistance. However, the AUTOSAR and OSEK operating system awareness feature is designed to be flexible, and will handle even an ORTI file that fails to provide any task information whatsoever.

Compiling the ORTI File

In order to use AUTOSARand OSEK operating system awareness with the ORTI OSA Explorer, two files are required: the ORTI file and a file describing various configuration and layout options of the ORTI OSA Explorer, commonly referred to as as an **.osa** file.

The format and specification of an **.osa** file is internal to the MULTI IDE, therefore the ORTI compiler (**ccorti**) is required to generate one from the ORTI file.

Note that both the **.osa** file and the ORTI file will be needed later to use the ORTI OSA Explorer in the MULTI IDE debugger.

An **.osa** file can easily be created using the following example command:

```
ccorti myortifile.orti -o myosafile.osa
```

Note

The **.osa** file will contain a reference to the ORTI file so do not delete it. If you move it, please edit the **.osa** file to refer to the new location. Otherwise the ORTI OSA Explorer will fail to operate correctly.

Since the MULTI IDE will automatically look in the subdirectory **defaults/os_aware** off of its installation directory for **.osa** files by default, it is recommended that **.osa** files be kept in that location for convenience. It may also be useful to store ORTI files in that location as well.

For additional **ccorti** options, please see the **ccorti** Reference.

If the compiler returns an error, please consult the Troubleshooting section.

Launching the ORTI OSA Explorer

Users can connect to the ORTI OSA Explorer by invoking the MULTI IDE debugger with the **-osa** command line option. The syntax for this option is:

-osa *osa_name*[#**cfg**=*configuration_file*][#**lib**=*library_name*][#**log**=*log_file*]

where *osa_name* is the name of your **.osa** file created with **ccorti**, but without the **.osa** extension. For more details, please see the MULTI documentation. However, if your **.osa** file created with **ccorti** was placed in the **defaults/os_aware** subdirectory of your MULTI installation, and you called it, for example, **myortiosa.osa**, then **-osa myortiosa** should be sufficient for most use cases.

Users can also connect to the ORTI OSA Explorer from within the MULTI IDE debugger, which is useful if MULTI was not invoked with the appropriate option. To do so, the user should use the **osasetup** command from the MULTI command (MULTI>) pane. Its general syntax is:

osasetup *osa_name* [**-cfg** *config_filename*] [**-lib** *module_name*] [**-log** *log_file*]

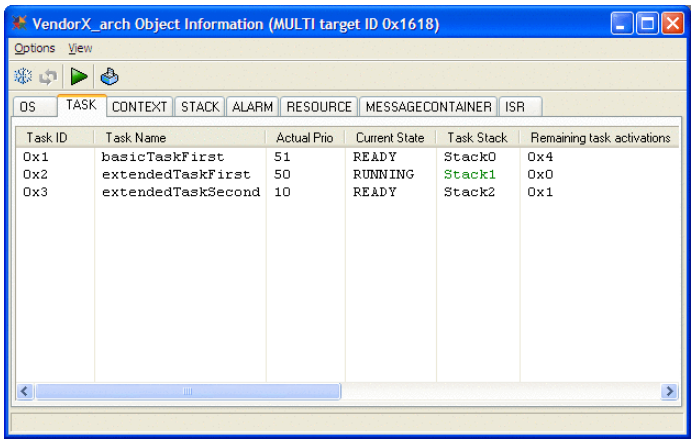
where *osa_name* is the name of your **.osa** file created with **ccorti**, but without the **.osa** extension. For more details, please see the documentation for this command in the MULTI documentation. Similar to the command line options, if your **.osa** file created with **ccorti** was placed in the **defaults/os_aware** subdirectory of your MULTI installation, and you called it, for example, **myortiosa.osa**, then in most cases **osasetup myortiosa** will be sufficient. In other words, **osasetup myortiosa** will launch the OSA Explorer for the **.osa** file **myortiosa.osa** located in **defaults/os_aware**.

Enabling the ORTI OSA Explorer

To get the benefit of the operating system awareness provided with the ORTI OSA Explorer, you must first connect to your target, download you application, and begin running it. If you are not familiar with this process, please refer to the MULTI documentation.

Once connected to your running application, you can access the operating system awareness capabilities at any time by stopping your application and opening the ORTI OSA Explorer (if it was not previously opened). You can open the ORTI OSA Explorer from the MULTI Debugger menu, under **View -> OSA Explorer...**

The ORTI OSA Explorer Window



The ORTI OSA Explorer window contains one tab for each underlying operating system object that was defined in the ORTI file. The ORTI 2.2 specification allows for seven standard object types (all optional) and an unlimited number of vendor extensions.

The ORTI OSA Explorer supports all seven standard object types, which are

- OS
- TASK
- CONTEXT
- STACK
- ALARM
- RESOURCE
- MESSAGECONTAINER

These standard object types allow the **ccorti** compiler and the ORTI OSA Explorer to make certain assumptions about their underlying attributes. In the sample screenshot above, a generic OS is displayed containing all seven standard objects and one vendor extension (ISR).

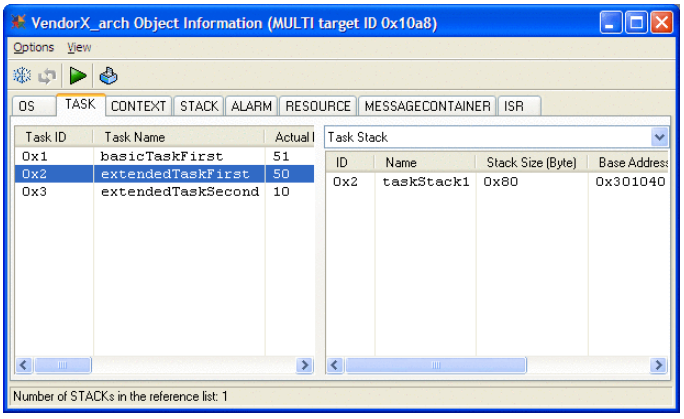
The ORTI OSA Explorer provides special double-click capabilities for TASK, STACK, and MESSAGECONTAINER objects.

- Double-clicking a TASK entry will highlight the task in the MULTI debugger.

- Double-clicking a STACK entry will open up a Memory View Window on the associated STACK's Base Address (if available). Double-clicking a MESSAGECONTAINER will open up a Memory View Window on the associated MESSAGECONTAINER's First Message attribute (if available).

The ORTI OSA Explorer also provides special right-click context menu capabilities for STACK and MESSAGECONTAINER objects, in addition to the right-click context menu items available with all MULTI OSA Explorers. For STACK objects, the ability to open Memory View windows and standard View windows on the Base Address attribute's value, when the Base Address is available. Similarly, Memory View windows and standard View windows can also be opened on the value of a MESSAGECONTAINER's First Message attribute, when that attribute is available. Standard View windows can also be open on both of these attribute's evaluated underlying expressions from the right-click context menu.

Cross-reference Capabilities



The ORTI specification provides a means of cross-referencing an attribute of one operating system object to another operating system object. For example, an ORTI file might provide a way to map a TASK's Task Stack attribute to the actual STACK object associated with it. If this information is available, the ORTI OSA Explorer will provide a way to access this through the right-click context menu. In our above example, a user would right-click a specific task entry and select **Go to STACK**.

It is also possible to display these cross-references in the GUI. However, as this capability consumes a large amount of the space in the ORTI OSA Explorer window, it is not enabled by default. To enable this capability, the ORTI file must be compiled

with **ccorti**'s **-x** option. See **ccorti** Reference for further details on this option. A sample screenshot with this option enabled is shown above.

ccorti Reference

The **ccorti** compiler options are:

--extended_syntax Enables support for non-standard ORTI syntax that some vendors use.
--help Prints a list of supported commands and associated help text.
--check_context_validity See the command **set check_context_validity** in the ORTI OSA Explorer Reference for details.
--cmd *command* Instructs the ORTI OSA Explorer to run the specified command on startup. See the ORTI OSA Explorer Reference for a list of possible commands
--loglevel *number* For debugging purposes; this option can affect the verbosity of messages displayed by the ORTI OSA Explorer.
-m See the Missing Context Information for details.
-o *string* Specifies the output filename.
-t *string* Specifies what name to give the ORTI OSA Explorer window.
-x See Cross-reference Capabilities for details.

ORTI OSA Explorer Reference

Commands can be issued to the ORTI OSA Explorer from the MULTI IDE Debugger's command (MULTI>) pane, or at startup in the **.osa** file using **ccorti**'s **--cmd** option. For details on the **--cmd** option approach, see the **ccorti** Reference.

Issuing commands from the MULTI IDE Debugger's command pane is done using the **osacmd** command. For example:

```
MULTI> osacmd version
ORTI OSA Module 0.1, Copyright (c) 2009 Green Hills Software Inc.
```

The following commands are supporting by the ORTI OSA Explorer:

clearcache

Clears all cached information.

help

This command displays a list of currently supported commands along with a short description.

loadconfig *filename*

Loads previously saved configuration data from the specified filename. This command is the complement to the **saveconfig** command. If no filename is specified, the name of the current ORTI file will have **.cfg** appended to it, and a configuration file of this name will be loaded (if it exists) from the user's local user directory (exact location varies based on host operating system). For example, if your ORTI file was named **sample.orti**, then the configuration file would be loaded from **~/.ghs/sample.orti.cfg**.

mapregname *register_id register_name*

Maps a MULTI IDE debugger register ID (from a GRD file) to a register name.

mapregloc *task_id register_name multi_expression*

This command is used to augment the task context information provided in the ORTI file. It is most commonly needed when an error like the following is encountered:

```
Task 0x1 (basicTaskFirst) is missing complete context information
in your ORTI file.
Register 5 (r5) was not found
Task aware debugging may not be fully available.
The debugger will not be able to reconstruct all of this task's registers.
```

If the user knows how to access the `r5` register in `basicTaskFirst`'s saved register context, then they can use the **mapregloc** command to teach the ORTI OSA Explorer this information. For example, suppose this register is located at:

```
*((unsigned int*)[taskStacks[1]->stack->registers[5*4])
```

Then the following command would instruct the ORTI OSA Explorer to find `r5` for Task 1 at this location:

```
mapregloc 1 r5 *((unsigned int*)[taskStacks[1]->stack->registers[5*4])
```

Note that *multi_expression* can be any expression valid in the MULTI IDE Debugger. You may want to test the expression by attempting to view or print its value first before issuing this command.

regname *register_id* Displays the register name(s) for the specified MULTI IDE debugger register ID.
saveconfig *filename* Saves current configuration to the specified filename. If no filename is specified, the name of the current ORTI file will have **.cfg** appended to it, and a configuration file of this name will be saved in the user's local user directory (exact location varies based on host operating system). For example, if your ORTI file was named **sample.orti**, the configuration file would be saved to **~/.ghs/sample.orti.cfg**.
set check_context_validity *[on|off|true|false|1|0]* When set to on, the ORTI OSA Explorer will not attempt to access a register saved in a task's saved context information if the context information is flagged as not valid. By default the ORTI OSA Explorer ignores the valid markers for context information.
set suppress_context_errors *[on|off|true|false|1|0]* By default the ORTI OSA Explorer will complain if the OS vendor's ORTI file did not contain information about where to find a task's saved registers when the task is not the currently running task. This option can be used to suppress those warnings. When the warnings are suppressed, the ORTI OSA Explorer will use the current value of any register requested.
togglesymbols Replaces values with the expressions used to obtain those values in the ORTI OSA Explorer window.
version This command displays a version string that can be used to identify the ORTI OSA Explorer software being used when conversing with Green Hills Software, Inc.'s support staff.

ORTI Troubleshooting

ccorti Syntax Errors

```
ccorti: error: syntax error parsing (line 102):
CONTEXT
ccorti: info: Done parsing ORTI KOIL information from myortifile.orti
ccorti: error: Customers current on maintenance may wish to contact support@ghs.com
for assistance with these errors
ccorti: fatal: 1 errors encountered
```

Syntax errors are most commonly encountered when the ORTI file provided by the OS vendor contains an error, or is not in conformance with the ORTI standard supported by **ccorti**.

To receive assistance with these problems, and to possibly receive instructions on how to modify the ORTI file to work around or avoid the error, customers current on maintenance with their product can contact Green Hills Software's technical support for assistance. Please be sure to provide a copy of the ORTI file in question.

Missing Context Information

```
ccorti: warning: Your ORTI file did not contain any register context information.
 You may want to recompile with -m to avoid errors with the ORTI OSA Explorer
```

The above warning is given if the ORTI file did not contain information recognized by the compiler describing where to locate registers associated with tasks that are not currently running. This information is necessary in order for the MULTI IDE to indicate where a pended task was before its context was switched; specifically MULTI needs to be able to access the saved program counter (PC) register. The saved stack pointer and return/link register, as well as other registers may be required on some architectures to display this information. Additionally, without context information, MULTI will not be able to display the saved register values for a task.

This warning is typically the result of the OS vendor failing to provide the necessary information in the ORTI file.

Users will notice these problems when the ORTI OSA Explorer displays an error such as the following error in the MULTI debugger indicating that register `r5` could not be read due to missing context information:

```
Task 0x1 (basicTaskFirst) is missing complete context information
in your ORTI file.
Register 5 (r5) was not found
Task aware debugging may not be fully available.
MULTI will not be able to reconstruct all of this task's registers.
```

Users have two options when this situation arises:

1. Ignore the problem
2. Attempt to teach the ORTI OSA Explorer how to overcome the problem by dynamically updating the ORTI information.

Ignoring the Problem

If the user wishes to ignore the problem, these messages can be suppressed. When these messages are suppressed, the ORTI OSA Explorer will substitute the current values of the registers from the current running task

The easiest way to achieve this is to do nothing. The first time these errors are printed for a specific register, the ORTI OSA Explorer will remember the problem and not print the error again for future accesses to the same register. This setting, however, is not remembered across sessions.

Alternatively, the user can instruct the ORTI OSA Explorer to substitute the values of all missing registers with the values those registers have on the target. This can be done temporarily by issuing this command:

```
osacmd set suppress_context_errors
```

Or it can be done permanently by recompiling the ORTI file with the **-m** option, as suggested by the original error message produced by **ccorti**.

Updating the ORTI Information Dynamically

The user can instruct the ORTI OSA Explorer on how to find the missing register by using the **mapregloc** command. Please see the documentation for this command in the ORTI OSA Explorer Reference.

To preserve these settings across a session, the ORTI file can be re-compiled with the appropriate **--cmd** options to **ccorti**. Please see the **ccorti** Reference for details on this option.

Missing ORTI File

```
Target: Could not open file C:/svn/orti/sample.orti: No such file or directory
Target: Failed to execute command "load_orti "C:/svn/orti/sample.orti"" in orti OSA.
```

This error is given when the **.osa** file references an ORTI file that does not exist. The **.osa** file should be edited, and the `OSA_CONFIG:OSA_INIT_COMMAND:load_orti` line updated to point to the proper location of the ORTI file used to create the **.osa** file. On Windows, the `\` path

delimiter must be escaped. Alternatively, the forward slash (/) character may be used.

Other Issues

If you have questions or problems not addressed by the documentation herein, and are under maintenance for this product, please contact Green Hills Software's technical support for further assistance.

The protrans Utility

Note

This information is only relevant if you are profiling a run-mode task or AddressSpace on an INTEGRITY target, or if you are profiling a stand-alone program.

Running **protrans** separately is useful when you want to automate the acquisition of large amounts of profiling data and then use the **Profile** window to display the data once all the executions are complete. Normally, when you use MULTI to collect profiling data, the actions of the **protrans** utility are transparent to you. This utility reads the profiling data that is produced when a program is executed, and it can accumulate profiling data over multiple executions of a program. It then translates the data into an intermediate format that the **Profile** window uses when displaying profiling information.

The information in this section is provided for advanced users who want to run **protrans** outside of the MULTI environment. To run this utility manually, issue the following command from a command shell:

```
protrans options program
```

where *options* can be any non-conflicting combination of the options described in the following table.

Option	Meaning
-a	Adds the profiling data for the current execution to a summary profile with a **.pro** extension. Without this switch, a new profile, which overwrites the previous profile, is generated with each execution.
-e	Writes the error messages to a file with the same base name and an **.err** extension.
-q	Suppresses message printing. Without this switch, **protrans** prints a small message for each execution, describing the type of profiling data found.
-m *file*	Specifies *file* as a **mon.out** file containing call count (but not call graph) profiling data. You can specify multiple files with multiple uses of this option.
-b *file*	Specifies *file* as a **bmon.out** file containing coverage analysis profiling data. You can specify multiple files with multiple uses of this option.

Option	Meaning
-b64 ***file***	Specifies *file* as a **bmon64.out** file containing coverage analysis profiling data containing 64-bit counters. You can specify multiple files with multiple uses of this option.
-G ***file***	Specifies output *file* as a **.gpro** profile recording that contains debug information and profile information.
-cc ***file***	Specifies *file* as a raw memory dump of the `.ghcovcz` section containing coverage counts. You can specify multiple files with multiple uses of this option.
-cd ***file***	Specifies *file* as a raw memory dump of the `.ghcovdz` section containing 64-bit coverage counts. You can specify multiple files with multiple uses of this option.
-cf ***file***	Specifies *file* as a raw memory dump of the `.ghcovfz` section containing coverage flags. You can specify multiple files with multiple uses of this option.
-ccdir ***dir***	Specifies *dir* as a directory that contains one or more raw memory dumps of the `.ghcovcz` section containing coverage counts.
-cddir ***dir***	Specifies *dir* as a directory that contains one or more raw memory dumps of the `.ghcovdz` section containing 64-bit coverage counts.
-cfdir ***dir***	Specifies *dir* as a directory that contains one or more raw memory dumps of the `.ghcovfz` section containing coverage flags.
-d ***file***	Specifies *file* as a **.dbp** file containing PC samples. The contents of the file depend on the data collected.
-B	Specifies the target is big endian. The default endianness is that of the host.
-L	Specifies the target is little endian. The default endianness is that of the host.

Suppose you have a big endian V850 and RH850 program called **dylan**, built with call graph and legacy coverage analysis profiling (**-a**), and with a collection of sample input files: **sampleinput1**, **sampleinput2**, etc. Now consider the following **csh** shell script for Linux/Solaris hosts.

```
#!/bin/csh
foreach p (sampleinput*)
    sim850 dylan $p
    protrans -a -B -q dylan
end
rm -f gmon.out bmon.out
```

This script repeatedly runs the **dylan** program with the sample input using **sim850**, and calls the **protrans** utility to read in the generated profiling data for each execution. The argument **dylan** specifies the profiled program for the generated data (the default is **a.out**).

After the script finishes, an intermediate profiling data file is generated that contains the summary profile. The intermediate file has a **.pro** extension appended to the program name if it has no extension.

In the preceding example, the file **dylan.pro** is generated in the same directory where the shell script is executed.

After you have generated this file, you can process the data in the Debugger. You can read in the summary profile stored in **dylan.pro**, and then use the various features of the **Profile** window to view the information.

You can also dump the contents of this file using the **gdump** utility, which will display the data in an easy-to-read format. For information about the **gdump** utility, see "The gdump Utility Program" on page 524.

The last line in the preceding script deletes any remaining profiling data files after executions of the profiled program. By default, the **protrans** utility looks for the following files:

- **mon.out** — This file is produced after running a program built for call count profiling (without call graph support).
- **gmon.out** — This file is produced after running a program built for call count profiling with call graph support enabled.

Note

A program can only produce one or the other of **mon.out** or **gmon.out**.

- **bmon.out** — This file is produced after running a program built for coverage analysis. This type of profiling is done alone or in conjunction with either call count profiling or call count with call graph profiling.
- **bmon64.out** — This file is produced after running a native program built for coverage analysis with 64-bit counts. This type of profiling is done alone or in

conjunction with either call count profiling or call count with call graph profiling.

For more information about the profiling options that generate these files, see "Obtaining Profiling Information" on page 57. Files are not created by default for the **-coverage=** option. To get profiling information into a file on your host machine while using **-coverage=**, you must dump the relevant memory section from your target into a file on your host. For more information about these memory sections, see "Enabling Instrumented Coverage or Performance Profiling" on page 58. For information about dumping a memory section, see "Beginning, End, and Size of Section Symbols" on page 468.

You can also specify other specific data files to **protrans**.

Thus, the shell script example for Linux/Solaris hosts that is given earlier could be rewritten to:

```
#!/bin/csh
foreach p (sampleinput*)
    sim850 dylan $p
    mv gmon.out gmon.$p
    mv bmon.out bmon.$p
end
foreach p (sampleinput*)
    protrans -a -B -q -g gmon.$p -b bmon.$p dylan
    rm -f gmon.$p bmon.$p
end
```

The first loop runs **dylan** with the sample input and stores the generated data files into uniquely named temporary files. The second loop then calls **protrans** to read in the data from these files and produces a summary profile. The **-g** option specifies a call count with call graph profiling data file, and the **-b** option specifies a coverage analysis data file. You can specify multiple files of a single profiling data type with multiple uses of these options. For example:

```
protrans -g gmon.1 -g gmon.2 -g gmon.3
```

Part III

Language Reference

Chapter 12

Green Hills C

Contents

This chapter describes the Green Hills implementation of C, including aspects of the language particular to our compilers.

Specifying a C Language Dialect

The Green Hills C Compiler supports six main variants of the C Language.

To specify a C dialect:

Set the **C/C++ Compiler→C Language Dialect** option to one of the following settings:

- **Strict ISO C99 (-C99)** — Accepts the ISO C99 language without extensions, as defined by *ISO/IEC 9899:1999*.
- **ISO C99 (-c99)** — Accepts the ISO C99 language with extensions (see "ISO C99" on page 636).
- **Strict ANSI C (-ANSI)** — Accepts the ANSI C language without extensions, as defined by *X3.159–1989*.
- **ANSI C (-ansi)** — [default] Recommended for all new development, and for the compilation of any existing C code which approximates to ANSI C. For more information, see "ANSI C" on page 623.
- **GNU C (-gcc)** — Recommended for porting existing GNU code to the Green Hills development environment. For more information, see "GNU C" on page 630.
- **K+R C (-k+r)** — Support for K&R mode is deprecated and may be removed in future versions of MULTI. Recommended only for porting existing K&R code to the Green Hills development environment. For more information, see "K&R C" on page 638.

The ANSI C standard can be considered the base of all dialects. Each dialect provides certain extensions and features in addition to the basic ANSI language, except K&R C, which omits some features and allows various non-standard constructs. If your program is written in standard ANSI C, any of the modes except for K&R C should compile your program correctly with few exceptions. All dialects use the same library and header files, which provides the complete ISO C99 library. A few header files and their associated functions are only supported in the ISO C99 dialect. We strongly recommend that you compile all modules in a particular program using the same dialect, as a failure to do so could result in obscure failures at link time or run time.

In addition to the six primary C dialects, we also provide support for:

- the Motor Industry Software Reliability Association (MISRA) C rules (see “MISRA C 2004” on page 168)
- the Japanese Automotive C extensions (see “Japanese Automotive C Extensions” on page 643)

ANSI C

The ANSI C dialect is the default dialect. We recommend that you use this mode for all new development, as well as for the compilation of any existing C code that comes close to ANSI C.

To use this dialect:

Set the **C/C++ Compiler→C Language Dialect** option to **ANSI C** (**-ansi**).

The compiler accepts the ANSI C mode as defined by *X3.159–1989*, with the addition of support for numerous extensions. These extensions may be useful, or even necessary, in certain cases. Some cases which are prohibited by the ANSI standard generate a warning in this mode.

ANSI C Extensions

The extensions provided in this mode are as follows:

- Bitfields may have base types that are enumerated or integral types besides `int` and `unsigned int`. This matches A.6.5.8 in the ANSI Common Extensions appendix.
- Empty source files.
- A translation unit (input file) can contain no declarations.
- The last member of a structure may have an incomplete array type. It may not be the only member of the structure (otherwise, the structure would have zero size).

- A file-scope array can have an incomplete structure, union, or enumerated type as its element type. The type must be completed before the array is subscripted (if it is), and by the end of the compilation if the array is not `extern`.
- `enum` tags may be forward-declared; one may define the tag name and resolve it later (by specifying the brace-enclosed list).
- The values of enumeration constants may be given by expressions that evaluate to unsigned quantities that fit in the `unsigned int` range, but not in the `int` range. A warning is issued for suspicious cases. For example:

```
/* when ints are 32 bits: */
enum a { w = -2147483648}; /* No warning */
enum b { x = 0x80000000};  /* No warning */
enum c { y = 0x80000001};  /* No warning */
enum d { z = 2147483649};  /* Warning */
```

- An extra comma is allowed at the end of an `enum` list.
- A label or `case` label can be immediately followed by a right brace. (Normally, a statement must follow a label.)
- An initializer expression that is a single value and is used to initialize an entire static array, structure, or union need not be enclosed in braces. Strict ANSI C requires the braces. This extension is not allowed in GNU mode.
- In an initializer with a file-scope variable, a pointer constant value may be cast to an integral type if the integral type is big enough to contain it.
- The address of a variable with `register` storage class can be taken. A warning is issued.
- In an integral constant expression, an integer constant may be cast to a pointer type and then back to an integral type.
- `long float` is accepted as a synonym for `double`.
- The `long long` and `unsigned long long` types are accepted. Integer constants suffixed by `LL` are given the type `long long`, and those suffixed by `ULL` are given the type `unsigned long long` (any of the suffix letters may be written in lowercase).
- Benign redeclarations of `typedef` names are allowed. That is, a `typedef` name may be redeclared in the same scope as long as it is declared with the same type. A warning is issued.
- Dollar signs (`$`) are accepted in identifiers, even in strict dialects.

- Numbers are scanned according to the syntax for numbers rather than the `pp-number` syntax. Thus, `0x123e+1` is scanned as three tokens instead of one invalid token.
- Assignment and pointer difference are allowed between pointers to types that are interchangeable but not identical (for example, `unsigned char *` and `char *`). This includes pointer to same-sized integral types (for example, typically, `int *` and `long *`). Assignment of a string constant to a pointer to any kind of character is allowed without a warning.
- Assignment of pointer types is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `const int **`). Comparison and pointer difference of such pairs of pointer types are also allowed. A warning is issued.
- The expressions `p+i`, `p-i`, and `p-q` are allowed when `p` and `q` are of type `void *`, and behave like the same operations on `char *`.
- Pointers to different function types may be assigned or compared for equality (==) or inequality (`!=`) without an explicit type cast. A warning is issued.
- A pointer to `void` may be implicitly converted to or from a pointer to a function type.
- The `#assert` preprocessing extensions of AT&T System V Release 4 are allowed. This feature is deprecated and may be removed in future versions of MULTI. These allow definition and testing of predicate names. Such names are in a name space distinct from all other names, including macro names. A predicate name is given a definition by a preprocessing directive of the form:

  ```
  #assert name
  #assert name(token-sequence)
  ```

  which defines the predicate *name*. In the first form, the predicate is not given a value. In the second form, it is given the value `token-sequence`. Such a predicate can be tested in a `#if` expression, as follows:

  ```
  #name(token-sequence)
  ```

  which has the value `1` if `#assert` of that `name` with that `token-sequence` has appeared, and `0` otherwise. A given predicate can be given more than one value at a given time. A predicate may be deleted by a preprocessing directive of the form:

```
#unassert name
#unassert name(token-sequence)
```

The first form removes all definitions of the indicated predicate name; the second form removes just the indicated definition, leaving any others that might exist.

- The keyword `asm` is recognized as a synonym for `__asm`. This feature is disabled in Strict ANSI C mode because it conflicts with the ANSI C Standard for statements such as:

  ```
  asm("xyz");
  ```

  ANSI C interprets this syntax as a call of an implicitly defined function `asm`, which (by default) the compiler interprets as an `asm` statement.

- `asm` macros are accepted, and `__asm` is recognized as a synonym for `asm`. An `asm` macro must be declared with no storage class (except `static`) and with a prototyped parameter list:

  ```
  asm void f(int a, int b) {
      ...
  }
  ```

  For more information, see Chapter 18, “Enhanced asm Macro Facility for C and C++” on page 847.

- An extension is supported to allow constructs similar to C++ anonymous unions. In addition, anonymous structures are allowed—that is, their members are promoted to the scope of the containing structure and looked up in the same manner as ordinary members.

- An ellipsis can appear as the only thing in a function's parameter list:

  ```
  void f(...);
  ```

- External entities declared in other scopes are visible. A warning is issued. For example:

  ```
  void f1(void) { extern void f(); }
  void f2() { f(); /* Using out of scope declaration */ }
  ```

- C99-style comments (using `//` as delimiter) are supported.

- The C99 `_Pragma` preprocessing operator is accepted in all dialects.

- A structure that has no named fields, but at least one unnamed field (such as a zero-length bitfield), is accepted by default. A warning is issued.
- The `#warning` preprocessor directive, similar to `#error`, is supported in all dialects.
- Attributes, introduced by the keyword `__attribute__`, can be used on declarations of variables, functions, types, and fields. This extension mimics the GNU compiler for the attributes that are accepted. The following attributes are accepted:
  - `alias`, `aligned`, `cdecl`, `common`, `const`, `deprecated`, `format`, `format_arg`, `no_check_memory_usage`, `no_instrument_function`, `nocommon`, `noreturn`, `packed`, `pure`, `section`, `stdcall`, `transparent_union`, `unused`, `used`, `volatile`, `weak`
- The `externally_visible` attribute expresses that a symbol might be referenced from outside of the program, and affects Wholeprogram optimizations. Similarly, you can specify that a symbol is externally visible using the **-external=** and **-external_file=** options (see "Optimization Scope" on page 148), or by using `#pragma ghs start_externally_visible` and `#pragma ghs end_externally_visible` (see "Green Hills Extension Pragma Directives" on page 753). To learn more about the use of the `externally_visible` attribute, see "Wholeprogram Optimizations" on page 310.
- The `__alignof__` keyword is supported. It is similar to `sizeof`, but returns the alignment requirement value for a type, or `1` if there is no alignment requirement. It can be followed by a type or expression in parentheses:

  ```
  __alignof__(type)
  __alignof__(expression)
  ```

  The expression in the second form is not evaluated.
- Preprocessing directive `#import` is supported.

  ```
  #import "filename"
  ```

  or

  ```
  #import <filename>
  ```

causes *filename* to be included only once. This feature is deprecated and may be removed in future versions of MULTI.

- The preprocessor directive `#include_next` is deprecated and may be removed in future versions of MULTI.
- Keyword `__inline__` is supported (see "Manual Inlining In C" on page 295).
- Keyword `__typeof__` is supported. It takes an expression as the argument, and returns the type of the expression. For instance, the following code:

  ```
  int i;
  __typeof__(i) j;
  ```

  would declare variables `i` and `j`, both of type `int`.
- The macros `__FUNCTION__` and `__PRETTY_FUNCTION__` are supported. They expand to the name of the function surrounding them. In C++, `__PRETTY_FUNCTION__` expands to the fully qualified name of the surrounding function.
- A function can be declared with the `__linkonce` storage class. The compiler will place the function in a special section whose name begins with `.ghs.linkonce` and whose name encodes the function name. The linker arbitrarily selects one object file from which to include a link-once section corresponding to each unique name. `__linkonce` functions cannot be static.
- The language is extended with the `__packed`, `__bytereversed`, `__bigendian`, and `__littleendian` type qualifiers. For more information about these type qualifiers, see "Type Qualifiers" on page 644
- The C99 `_Bool` type is accepted in all dialects.
- C99 digraph tokens, such as `<%` and `%:`, are accepted in all C and C++ dialects.
- As in C99, the C preprocessor treats all integer types as equivalent to `intmax_t` or `uintmax_t` during expression evaluation. For example:

```
int main()
{
    #if 0x80000000*2 != 0
        puts("The preprocessor uses 64-bit arithmetic on most targets");
    #endif
    if (0x80000000*2 == 0)
        puts("Normal arithmetic uses 32-bit arithmetic on 32-bit targets");
}
```

Strict ANSI C

-ANSI implements the standard C language that is defined in *X3.159–1989*. In this mode, only a few extensions to the standard are permitted (for example, `long long`) and a number of rules are strictly enforced.

To specify use of this dialect:

Set the **C/C++ Compiler→C Language Dialect** option to **Strict ANSI C** (**-ANSI**).

GNU C

The GNU C dialect provides close compatibility with code written for the GNU compiler. It accepts all of the ANSI C dialect features, together with a number of GNU-specific extensions listed below. It is recommended that you use this dialect only when porting existing GNU-compiled code to the Green Hills development environment.

To specify use of this dialect:

Set the **C/C++ Compiler→C Language Dialect** option to **GNU C** (**-gcc**).

GNU C Only Extensions

This dialect enables all the ANSI extensions listed in "ANSI C Extensions" on page 623, together with the extensions listed in this section, and those listed in "GNU C/C++ Extensions" on page 633.

- Variable length arrays (VLAs) are supported. GNU C also allows VLA types for fields of local structures, which can lead to run-time dependent sizes and offsets. The front end does not implement this, but instead treats such arrays as having length zero (with a warning); this enables some popular programming idioms involving fields with VLA types.

```
void f(int n)
{
    struct {
    int a[n]; /* Warning: variable-length array field type will be
                 treated as zero-length array field type */
    };
}
```

- A nonconstant may appear in the initializer list of an aggregate variable with automatic storage class.
- You can use GNU C-style inlining by using the keywords `inline` or `__inline__`. For more information about inlining functions manually, see "Manual Inlining" on page 295.

- Structs and unions without fields are accepted. They have size zero. Class types containing only fields of size zero (zero-length bitfields, zero-length arrays, and class types of size zero) also have size zero.
- In conditional expressions, the middle operand can be left out. When that is the case, the result of the expression is the value of the first operand if that first operand is "true".

```
x = x ?: -1; /* If x is zero, evaluates to -1; otherwise, just x */
```

- Old-style definitions with unpromoted parameter types can follow prototype declarations with those same types.

```
void f(short);        // Prototype
void f() short p; {} // Old-style definition OK
```

- An unprototyped declaration can follow a prototyped declaration. After such a redeclaration the original prototype is ignored. A warning is issued.

```
void f(char);        // Original prototype
void f();            // Unprototyped redeclaration
void f(double x) {} // Accepted in GNU C mode
                     // (original prototype ignored)
```

- **GNU 3.3 ONLY** Certain "generalized lvalues" are supported. The ternary conditional operator produces an lvalue if its second and third operand are lvalues of the same type. Similarly, the comma operator produces an lvalue if its second operand is an lvalue. Some casts also preserve the lvalueness of their operand. For example:

```
unsigned x, y, z;
(x ? y : z) = 4; // Updates y or z depending on x (GNU 3.3 only)
(x += y, y) = 3; // Updates y (allowed in all non-strict modes)
(int)x = -1;     // Updates x (warning in all non-strict modes))
```

  A "`?`" operator can also be treated as an lvalue if its operands have types that are different but are close enough that an lvalue cast can bridge the gap (for example, `int` and `int *`). A warning is issued.
- An expression can be cast to a union type containing a field with the type of that expression. For example:

```
union X { int i; char c; };
union X f(int i) { return (union X)i; }
```

- Implicit conversions are allowed between integral and pointer types, and between incompatible pointer types, with a warning. Implicit conversions between pointer types can drop cv-qualifiers (for example, `const char *` to `char *`).
- Pointer arithmetic applies to `void` pointers and function pointers as if they were `char *`.
- Explicit casts to struct and union types are allowed if they are do-nothing casts, that is, if the source type is the same as the destination type. (cv-qualifier differences are allowed but ignored.) The result is an lvalue if the source is an lvalue.
- Functions with return type `void` can have return expressions. A warning is issued if the return expression does not have type `void`.
- **GNU 3.3 ONLY** Bitfield lengths that are too large for the associated type are ignored with a warning (that is, the field becomes an ordinary field whose address can be taken). For example:

```
struct S {
  char c: 9; // Oversized bitfield becomes regular field of type char
};           // (assuming a char can hold at most 8 bits)
```

- **GNU 3.3 ONLY** Casts on an lvalue that do not fall under the usual "lvalue cast" interpretation (for example, because they cast to a type having a different size) are ignored, and the operand remains an lvalue. A warning is issued.

```
int i;
(short)i = 0; // Accepted,cast is ignored; entire int is set
```

- "`?`" operators with mixed void/non-void operands in the second and third operands are accepted. The result has type `void`.
- **GNU 3.3 ONLY** In function definitions, local declarations can hide parameters.

```
void f(int x) {
  double x; // Accepted in GNU C mode (with a warning).
}
```

- An old-style parameter list can be followed by an ellipsis to indicate that the function accepts variable-length argument lists.

```
void f(x) int x; ... {} // Accepted in GNU C mode
```

GNU C/C++ Extensions

The extensions listed in this section are enabled when GNU C or GNU C++ is selected. See also "GNU C Only Extensions" on page 630 and "GNU C++ Extensions" on page 699.

- A function can be declared `inline`. It denotes that the inliner should attempt to inline calls to the function. The keyword `inline` is ignored (with a warning) on variable declarations and on block-`extern` function declarations.
- Keyword `typeof` is supported. It behaves exactly like `__typeof__`. For more information, see "ANSI C Extensions" on page 623.
- Implicit conversion of pointer types with the source type having a more qualified base type is allowed with a warning. Hence, only a warning is issued in the following case:

```
const char *s;
char       *p;
/* ... */
p = s;            /* error in ANSI, warning with GNU */
```

- Compound literals are accepted. In addition to the features of C99 compound literals, GNU C mode also allows such literals to be used to initialize variables in file scope (if they only involve constant-expressions) and to initialize elements of aggregate types in brace-enclosed aggregate initializers.
- The `__extension__` keyword is accepted preceding declarations and certain expressions. It has no effect on the meaning of a program:

```
__extension__ __inline__ int f(int a) {
  return a > 0 ? a/2 : f(__extension__ 1-a); }
```

- Zero-length array types are supported. These are complete types of size zero.
- C99-style flexible array members are accepted. In GNU C++ mode, flexible array members are treated exactly like zero-length arrays.

- The `sizeof` operator is applicable to `void` and to function types and evaluates to the value `1`.
- The "assembler name" of variables and routines can be specified. For example:

  ```
  int counter __asm__("counter_v1") = 0;
  ```

  Assembler names containing whitespace, non-identifier characters such as `$` or `@`, leading periods (`.`), or leading underscores (_) may not behave as expected.
- Excess aggregate initializers are ignored with a warning.

  ```
  struct S { int a, b; };
  struct S a1 = { 1, 2, 3 }; // "3" ignored with a warning; no error
  int a2[2] = { 7, 8, 9 };   // "9" ignored with a warning; no error
  ```

- The `__restrict__` keyword is accepted. It is identical to the C99 `restrict` keyword, except for its spelling.
- Extended variadic macros are supported.
- C99 hexadecimal floating point constants are recognized.
- The `\e` escape sequence is recognized and stands for the ASCII "`ESC`" character.
- The address of a statement label can be taken by use of the prefix "`&&`" operator, for example, `void *a = &&L`. A transfer to the address of a label can be done by the "`goto *`" statement, for example, `goto *a`.
- The last line of an include file can end with a backslash.
- Case ranges are supported. For example:

  ```
  case 'a'...'z'
  ```

- A large number of special functions of the form `__builtin_xyz` (for example, `__builtin_alloca`) are predeclared.
- Some expressions are considered to be constant-expressions even though they are not so considered in standard C and C++. For example:
  - `((char *)&((struct S *)0)->c[0]) - (char *)0`
  - `(int)"Hello" & 0`
- The macro `__GNUC__` is predefined to the major version number of the emulated GNU compiler. Similarly, the macro `__GNUC_MINOR__` is predefined to the corresponding minor version number. Finally, `__VERSION__` is predefined to a string describing the compiler version.

- Nonstandard casts are allowed in null pointer constants. For example, the following is considered a null pointer constant in spite of the pointer cast in the middle:

  ```
  (int)(int *)0
  ```

- Either the `__GNUC_STDC_INLINE__` macro or the `__GNUC_GNU_INLINE__` macro will be predefined, depending on the style of inlining in use by the compiler.
- Statement expressions such as `({int j; j = f(); j;})` are accepted. Branches into a statement expression are not allowed. In C++ mode, branches out are also not allowed. Variable-length arrays, destructible entities, try, catch, local non-POD class definitions, and dynamically initialized local static variables are not allowed inside a statement expression.
- Labels can be declared to be local in statement expressions by introducing them with a `__label__` declaration.

  ```
  ({ __label__ lab; int i = 4; lab: i = 2*i-1; if (!(i%17)) goto lab; i; })
  ```

- The C99 predefined identifier `__func__` is recognized.

The following GNU extensions are not currently supported in any GNU mode:

- GNU-style complex numbers (including complex literals).
- Nested functions.
- The `decltype` keyword
- Type trait helpers (`is_enum()`, `is_array()`, etc.)
- Binary literals

Strict ISO C99

The Green Hills compiler and libraries fully support the ISO C99 standard, which introduced numerous features in both the language and libraries. Previous releases of the Green Hills compiler and library included some features of ISO C99 as extensions to ANSI C (in the **-ansi** dialect), but now the complete language and libraries are also provided as a separate dialect.

To use the Strict ISO C99 dialect:

Set the **C/C++ Compiler→C Language Dialect** option to **Strict ISO C99** (**-C99**).

ISO C99

The ISO C99 dialect provides all of the features of the language defined by *ISO/IEC 9899:1999*, in addition to several useful extensions. This mode is recommended when developing projects that can benefit from the more precise definition of the language and extended features provided by C99. For a complete description of the ISO C99 language, see the *ISO/IEC 9899:1999* standard.

To use the ISO C99 dialect:

Set the **C/C++ Compiler→C Language Dialect** option to **ISO C99** (**-c99**).

In a few cases, the exact definition of a feature in the ISO C99 standard is different from the existing Green Hills implementation.

Features that are supported in C99 mode but not in ANSI mode:

- Complex types and **complex.h**.
- Hexadecimal floating point constants.
- The `restrict` keyword.
- Type-generic macros and **tgmath.h**

- Declarations after statements in a function or block, as in C++.
- Declaration of variables in the header of a `for` loop, as in C++.
- Non-constant initializers for automatic arrays, structures, and unions.
- `static` in parameter array declarators.
- Universal character names (UCNs) designated with `\u` or `\U`.

Preprocessor Support in C99 Mode

Preprocessor features that are only supported in C99 mode:

- The macro `__STDC_VERSION__` has the value `199901L` (in other modes the macro has the value `199409L`).
- `#pragma STDC FP_CONTRACT` and `#pragma STDC FENV_ACCESS`.
- `__func__` predefined identifier.
- The macros `__STDC_IEC_559__`, `__STDC_IEC_559_COMPLEX__`, and `__STDC_ISO_10646__` are undefined, which indicates that the implementation does not conform to the IEC-559 standard for floating point or the ISO-10646 standard for `wchar_t`.

Enforced Requirements in C99 Mode

Requirements that are enforced in C99 mode:

- Functions must be declared before use, although the prototype form is not required.
- Return statements must return a value if the function is non-void, and cannot return a value if the function is void.

Differences Between C99 Mode and ANSI C Mode

When using C99 mode, the following features differ from ANSI C mode:

- Large integer constants promote differently unless designated as `unsigned` or `long long` by using suffixes such as **U** or **LL**.
- Many of the extensions accepted in `-ansi` are not accepted in `-c99`.

K&R C

Support for K&R C is deprecated and may be removed in future versions of MULTI.

K&R is the classic dialect of C in which UNIX was originally written. The first edition of *The C Programming Language* by Kernighan and Ritchie describes the K&R dialect. The most important implementation of this dialect was the *Portable C Compiler* (PCC). Although this dialect has important historical significance and might be necessary for compiling legacy applications, it is recommended that wherever possible, applications should be converted to one of the newer modes of the C language.

To specify use of this dialect:

Set the **C/C++ Compiler→C Language Dialect** option to **K+R C** (**-k+r**).

The following is a list of incompatibilities between the Green Hills K&R Mode and PCC:

- Token pasting is not performed outside of macro expansions when only a comment separates two tokens. For example, `a/**/b` is not considered to be `ab`. The PCC behavior in that case can be achieved by preprocessing to a text file and then compiling that file.

  Note that `/**/` may be used to concatenate two strings in macro definitions.
- PCC considers the result of a `? :` operator to be an lvalue if the first operand is constant and the second and third operands are compatible lvalues. The Green Hills compilers do not.
- PCC incorrectly parses the third operand of a `? :` operator in a way that some programs exploit. For example:

  ```
  i ? j : k += l
  ```

  is parsed by PCC as

  ```
  i ? j : (k += l)
  ```

which is incorrect, since the precedence of += is lower than the precedence of `? :`. The compiler will generate an error for that case.

The following section lists K&R Mode extensions to K&R C.

K&R Mode Extensions to K&R C

- the `void` keyword
- `enum` types
- the predefined macro names `__LINE__`, `__FILE__`, `__TIME__`, and `__DATE__`
- `#error`, `#ident`, `#elif`, `#defined(`*`id`*`)` preprocessor directives
- functions that return `struct` types
- `asm(`*`str`*`)` statements
- initialized `extern` variables
- initialized variables of `union` types
- initialized automatic variables of `struct` and array types
- keywords `const`, `signed`, and `volatile` are recognized
- Declarations of the form

  ```
  typedef some-type void;
  ```

  are ignored.
- Assignment is allowed between pointers and integers, and between incompatible pointer types, without an explicit cast. A warning is issued.
- A field selection of the form `p->field` is allowed if `p` does not point to a structure or union that contains `field`. `p` must be a pointer or an integer. Likewise, `x.field` is allowed even if `x` is not a structure or union that contains `field`. `x` must be an lvalue. For both cases, all definitions of `field` as a field must have the same offset within their structure or union.
- Overflows detected while folding signed integer operations on constants cause warnings rather than errors.
- A warning will be issued for an `&` applied to an array. The type of such an operation is “address of array element” rather than “address of array”.

- For the shift operators << and >>, the usual arithmetic conversions are done, the right operand is converted to `int`, and the result type is the type of the left operand. In ANSI C, the integral promotions are done on the two operands, and the result type is the type of the left operand. The effect of this difference is that, in K&R mode, a `long` shift count will force the shift to be done as `long`.
- When preprocessing output is generated, the line-identifying directives will have the K&R form instead of the ANSI form.
- `sizeof` may be applied to bitfields; the size is that of the underlying type (for example, `unsigned int`).
- lvalues cast to a type of the same size remain lvalues, except when they involve a floating-point conversion.
- When a function parameter list begins with a `typedef` identifier, the parameter list is considered prototyped only if the `typedef` identifier is followed by something other than a comma or right parenthesis:

  ```
  typedef int t;
  int f(t) {}    /* Old-style list */
  int g(t x) {}  /* Prototyped list, parameter x of type t */
  ```

  That is, function parameters are allowed to have the same names as `typedef` identifiers. In the normal ANSI mode, any parameter list that begins with a `typedef` identifier is considered prototyped, so the first example above would give an error.
- The names of functions and of external variables are always entered at the file scope.
- A function declared `static`, used, and never defined is treated as if its storage class were `extern`.
- A file-scope array that has an unspecified storage class and remains incomplete at the end of the compilation will be treated as if its storage class is `extern` (in ANSI mode, the number of elements is changed to 1, and the storage class remains unspecified).
- The empty declaration:

  ```
  struct x;
  ```

  will not hide an outer-scope declaration of the same tag.

- In a declaration of a member of a structure or union, no diagnostic is issued for omitting the declarator list; nevertheless, such a declaration has no effect on the layout. For example:

```
struct s {
  char a;
  int;
  char b[2];
} v;              /* sizeof(v) is 3 */
```

- `short`, `long`, and `unsigned` are treated as "adjectives" in type specifiers, and they may be used to modify a `typedef` type.
- A "plain" `char` is considered to be the same as either `signed char` or `unsigned char`, depending on the target and command line options. In ANSI C, "plain" `char` is a third type distinct from both `signed char` and `unsigned char`.
- Free-standing tag declarations are allowed in the parameter declaration list for a function with old-style parameters.
- Declaration specifiers are allowed to be completely omitted in declarations (ANSI C allows this only for function declarations). Thus,

```
i;
```

  declares `i` as an `int` variable. A warning is issued.
- All `float` operations are done as `double`.
- `__STDC__` is left undefined.
- Extra spaces to prevent the pasting of easily confused adjacent tokens are not generated in textual preprocessing output.
- The first directory searched for include files is the directory containing the file containing the `#include` instead of the directory containing the primary source file.
- Trigraphs are not recognized.
- Comments are deleted entirely (instead of being replaced by one space) in preprocessing output.
- `0x` is accepted as a hexadecimal value `0`, with a warning.

- `1E+` is accepted as a floating-point constant with an exponent of `0`, and a warning is emitted.
- The compound assignment operators may be written as two tokens (for example, += may be written as + =).
- The digits 8 and 9 are allowed in octal constants.
- A warning, rather than an error, is issued for integer constants that are larger than can be accommodated in the largest unsigned integer type. The value is truncated to an acceptable number of low-order bits.
- The types of large integer constants are determined according to the K&R rules (they will not be `unsigned` in some cases where ANSI C would define them that way). Integer constants with apparent values larger than `LONG_MAX` are typed as `long` and are also marked as "non-arithmetic", which suppresses some warnings when using them.
- The escape `\a` (alert) is not recognized in character and string constants.
- Macro expansion is performed differently. Arguments to macros are not macro-expanded before being inserted into the expansion of the macro. Any macro invocations in the argument text are expanded when the macro expansion is rescanned. With this method, macro recursion is both possible and checked for.
- Token pasting inside macro expansions is performed differently. End-of-token markers are not maintained, so tokens that abut after macro substitution may be parsed as a single token.
- Macro parameter names inside character and string constants are recognized and appropriately substituted.
- Macro invocations having too many arguments are flagged with a warning rather than an error. The extra arguments are ignored.
- Macro invocations having too few arguments are flagged with a warning rather than an error. A null string is used as the value of the missing parameters.
- Extra `#else` statements (after the first has appeared in an `#if` block) are ignored, and a warning is emitted.
- The promotion rules for integers are different; `unsigned char` and `unsigned short` are promoted to `unsigned int`.
- An identifier in a function is allowed to have the same name as a parameter of the function. A warning is issued.

Motor Industry Software Reliability Association (MISRA) Rules

We provide support for the Motor Industry Software Reliability Association (MISRA) rules. For more information, see "MISRA C 2004" on page 168.

Japanese Automotive C Extensions

We provide support for the Japanese Automotive C extensions to ANSI C, which are used by Japanese automobile manufacturers. For complete specifications, refer to the *C-Language Specification for Automotive Control (Proposal)* by Toyota Motor Corporation, July 29, 1993.

Japanese Automotive C generally conforms to the principles of ISO 9899, equivalent to the ANSI *X3.159–1989* standard, with the exception of the "Implementation-Defined Behavior" specification of Annex G.3, which it modifies and extends to support portability. These extensions conform to the "Common Extension" section, in Annex G.5.

To enable support for the Japanese Automotive C extensions:

Set the **C/C++ Compiler→C Japanese Automotive Extensions** option to **On** (**-japanese_automotive_c**).

Enabling **C/C++ Compiler→C Japanese Automotive Extensions** (**-japanese_automotive_c**) implicitly modifies the following options:

- Sets **C/C++ Compiler→C Language Dialect** option to **ANSI C** (**-ansi**).
- Sets **C/C++ Compiler→Data Types→Signedness of Char Type** to **Unsigned** (**--unsigned_chars**)
- Sets **C/C++ Compiler→Data Types→Signedness of Bitfields** to **Unsigned** (**--unsigned_fields**)
- Sets **C/C++ Compiler→Data Types→Use Smallest Type Possible for Enum** to **Off** (**--no_short_enum**)
- Sets **Compiler Diagnostics→C/C++ Messages→Asm Statements** to **Silent** (**--asm_silent**)

In addition, the Green Hills tools satisfy the following Japanese Automotive C extensions by default:

- `#pragma intvect`, `#pragma asm`, and `#pragma endasm` are supported.
- If `sizeof(void *) == sizeof(int)`, any pointer to an object may be converted to an `int` and then back to a pointer that compares as equal to the original pointer.
- Bitfields may have arbitrary base types (satisfied by the **-ansi** option).

For example, if you compile the following code with the **C/C++ Compiler→C Language Dialect** option set to **Strict ANSI C** (**-ANSI**) :

```
struct {
    char b:3;
} s;
```

the compiler generates the following errors:

```
"x.c", line 2: error #230-D: nonstandard type for a bit field
      char b:3;
      ^
```

Enabling **C/C++ Compiler→C Japanese Automotive Extensions** (**-japanese_automotive_c**) sets the language dialect to **ANSI C** (**-ansi**), which suppresses these errors.

Type Qualifiers

The following type qualifiers are available:

__bytereversed

The `__bytereversed` type qualifier is used to designate that the data is stored in the opposite endianness, and that special means must be used to access it. The`__bytereversed` qualifier should not be used for automatic variables or parameters, although automatic pointers to `bytereversed` objects are allowed.

Example 12.1. Using the __bytereversed Qualifier

```
__bytereversed int x;
void foo(void)
{
    /* OK - automatic pointer to bytereversed data */
    __bytereversed int *p = &x;

    /* Not OK - __bytereversed should not apply to an automatic variable */
    int * __bytereversed p2;

    ...
}
```

`__bytereversed` should be applied only to integral and pointer types wider than `char`. `__bytereversed` variables should not be defined with an initializer.

__bigendian and __littleendian

Similarly, `__bigendian` and `__littleendian` can be used in place of `__bytereversed` to indicate a big or little endian byte ordering on the data. It is recommended that you do not use the type qualifier unnecessarily because this might cause problems or performance degradation if the code is ever ported to a CPU of the opposite endianness.

__packed

Use the `__packed` type qualifier to designate that an object might be misaligned, so the program must access it using a series of smaller, aligned accesses. For example, if you use `__packed` when declaring a pointer, you can safely use that pointer to access a field in a packed structure (see "Structure Packing" on page 28).

volatile

Use the `volatile` type qualifier to designate that an object may change due to an external process.

When your **C/C++ Compiler→C Language Dialect** is set to any mode other than K&R, enabling any of the **Optimization→Optimization Strategy** settings (see "Optimization Options" on page 145) will implicitly enable the **Advanced→Optimization Options→Memory Optimization** option (**-OM**), which assumes that memory locations only change under the control of the compiler. This

assumption is usually, but not always, true. Exceptions include memory which is updated by interrupt routines, and memory-mapped I/O.

This optimization allows the compiler sometimes to avoid and/or delay reads or writes to memory locations by maintaining a copy of the memory location in a register. This is generally much faster because reads and writes to registers are faster than reads and writes to memory. You can use the `volatile` qualifier to disable this optimization for particular variables (indicating that they may change), and retain its use for all other variables.

const

The compiler gives a compile-time error for any attempt to modify an object declared `const`. This qualifier also provides the compiler with additional information for use in optimizations. Wherever the value of a `const` variable is visible, the optimizer makes full use of the fact that this variable is simply a named constant value, combining it with other constants at compile time, and performing other simplifications. Even when the value of a `const` is not visible, the optimizer can make use of the fact that the variable is invariant to re-sequence statements and instructions or to move them outside of loops.

When `const` is used with `volatile`, the compiler never replaces a use of the object with its value, even if the value of the object is visible.

Assignment and Comparisons on struct and union Types

The assignment operator (=), which is supported in all modes of C, can be used to assign a value of a structure or a union type to a variable of the same type.

If there are padding bytes between fields or members of a `struct` or `union` type due to memory alignment requirements, those holes cannot be accessed by means of the structure or union. Note also that global variables will always be initialized to zero, and therefore the holes will always be zero; local variables, however, may have random data in the holes. Therefore, two structures or unions with the same values for every field might not be equal when compared. For structures or unions that will be compared, it is important to either have no holes in the memory representation or to explicitly set the “hole” bytes to zero via a call to `memset` before the comparison.

A structure or a union can be passed as an argument to a function without restriction. The structure or union is copied when it is passed, however, so passing a very large structure or union is much less efficient than passing a pointer. For this reason, we recommend that pointers be passed where possible.

Bitfields

Bitfields in C have a base type which determines the field's alignment, maximum size, and whether the field is signed or unsigned.

ANSI C Limitations

The ANSI C standard requires that the base type of a bitfield be either `int`, `signed int`, or `unsigned int`. This restriction is enforced when the **C/C++ Compiler→C Language Dialect** option is set to **Strict ANSI C** (**-ANSI**), and is ignored without a warning for all other dialects.

Signedness of Bitfields

Bitfields defined without either a `signed` or `unsigned` qualifier are `signed` by default (this default varies between target processor families). To control the signedness of bitfields manually:

Use the **C/C++ Compiler→Data Types→Signedness of Bitfields** option (**--signed_fields**/**--unsigned_fields**).

For example, if you set this option to **--signed_fields**:

- `int x:2` is a signed bitfield, by the default rule.
- `char c:2` has the same signedness as the char type (see “Signedness of Char Type” on page 197).
- `enum { MIN=-2, MAX=1 } e:2` is a signed bitfield, because the `enum` type has at least one negative enumerator.

- `enum { MIN=0, MAX=3 } e:2` is an unsigned bitfield, because the `enum` type has at least one positive enumerator which would not fit in the bitfield if it were signed.
- `enum { MIN=0, MAX=100 } e:2` is an unsigned bitfield, because the `enum` type has at least one positive enumerator which would not fit in the bitfield if it were signed (even though the enumerator still does not fit in the unsigned bitfield).
- `enum { MIN=0, MAX=1 } e:2` is a signed bitfield, by the default rule.

If you set this option to **--unsigned_fields**:

- `int x:2` is an unsigned bitfield.
- `char c:2` is an unsigned bitfield, even if plain `char` is signed.
- `enum { MIN=-2, MAX=1 } e:2` is a signed bitfield, because the `enum` type has at least one negative enumerator.
- `enum { MIN=0, MAX=1 } e:2` is an unsigned bitfield.
- `enum { MIN=0, MAX=3 } e:2` is an unsigned bitfield.
- `enum { MIN=0, MAX=100 } e:2` is an unsigned bitfield.

However, single-bit bitfields of integer type that are not explicitly declared `signed` are always unsigned. Regardless of the setting of **--signed_fields** or **--signed_chars**:

- `int x:1` is an unsigned bitfield
- `char c:1` is an unsigned bitfield

The choice of signed versus unsigned bitfields can also affect code efficiency. Unless the processor has special instructions for this purpose, it may require two or even three instructions to extract the bits of a signed field whenever that field is evaluated. This is because the bits must be sign-extended to the size of an `int`, which may require two arithmetic shifts.

The following example demonstrates another difficulty with signed bitfields:

```
int i;
struct {int x:2; } y;
y.x = 2;
i = y.x;
if (y.x > 0) func();
```

- If `x` is an unsigned field, `i` is assigned the value `2` and `func()` is called.
- If `x` is a signed field, `i` is assigned the value `-2` and `func()` is not called. This is because the range represented by a signed field with two bits is from `-2` to `1`. The value `2` is out of range. Out-of-range assignments are detected at compile time if warning 69 is enabled, or at run time if the translation unit has been compiled with **-check=assignbound** (see "Run-Time Error Checks" on page 159).

Size and Alignment of Bitfields

There is a close relationship between the base type of a field and the space it occupies, which can be summarized in three ways.

First, an individual bitfield can never have more bits than its base type.

Second, an unpacked bitfield must always fit entirely within a memory location that could hold its base type. It cannot cross a boundary that a simple variable of that type cannot cross. Thus, a field of type `char` can be placed in any memory location, but cannot cross a byte boundary. For example:

```
struct {
  char a:4;
  char b:6;
  char c:6;
} w;
```

Variable `w` requires three bytes because fields `a` and `b` do not fit in a single byte. Therefore, four bits of padding are inserted before field `b`. Similarly, the fields `b` and `c` do not fit in a single byte, so two bits of padding are inserted before the field `c`. On the other hand, if we had:

```
struct {
  short a:4;
  short b:6;
  short c:6;
} x;
```

then variable `x` would require only two bytes because the total size of the three fields is less than or equal to the size of a `short`.

Third, the alignment of a bitfield is determined by its base type. Padding may be inserted so that the field and the structure containing it are properly aligned. In the examples above, `w` has the alignment of a `char`, which is usually one byte, and `x` has the alignment of a `short`, which is usually either one or two bytes. Neither `w` nor `x` would be any larger due to alignment. However, in the following example, `y` will have an extra byte of padding added if `short` is 2 byte aligned:

```
struct {
  short a:4;
  short b:6;
  short c:6;
  char byte;  /* padding added after
                 'byte' for alignment */
} y;
```

For information about the size and alignment of C and C++ data types, see “V850 and RH850 Characteristics” on page 26.

Enumerated Types

An `enum` type is treated as one of the integral types. By default, the members of an `enum` type (enumerators) are treated as integral constants of type `int`. If you are compiling a C++ file or using a GNU C dialect (**-gcc** or **-gnu99**) and the values of the enumerators are outside of `int` range, they are treated as a larger integral type. To instruct the compiler to attempt to use the smallest possible predefined type (including unsigned types) that allows representation of all listed values:

Set the **C/C++ Compiler→Data Types→Use Smallest Type Possible for Enum** option to **On** (**--short_enum**).

The ANSI standard does not call for strict type checking with respect to `enum` types. In fact, a variable of enumerated type is considered equivalent to any other integral value. Most operations which are allowed on values of integral types are also allowed on enumerated types, including assignment of a variable or parameter of one enumerated type to a variable or parameter of another type.

Functions with Variable Arguments

Green Hills C and C++ compilers support functions with variable parameters. In ANSI C modes and C++, it is done using the **<stdarg.h>** facility. In K&R C mode, this is done using the **<varargs.h>** facility (though both this mode and the facility are deprecated). The two implementations are different and incompatible. Take care to use the correct facility for the appropriate mode of C.

An important limitation of variable parameters is that the types `char`, `short`, and `float` are not supported. When a function with variable parameters is called, the caller promotes expressions of these types to `int` (for `char` and `short`) and `double` (for `float`).

The <varargs.h> Facility

The **<varargs.h>** facility is deprecated and may be removed in future versions of **MULTI**.

To use the **<varargs.h>** facility, you must perform the following steps:

1. The line `#include <varargs.h>` must appear before the first function definition.
2. The last parameter of a variable argument list function must be named `va_alist`.
3. The last parameter declaration of a variable argument list function must be `va_dcl`.
4. There must not be a semicolon (`;`) between `va_dcl` and the initial left brace (`{`) of the function.
5. There must be a variable declared in the function of type `va_list`.
6. The **<varargs.h>** facility must be initialized at the top of the function by passing the variable of type `va_list` to a call of the macro `va_start`.
7. To obtain the variable arguments to the function, in left-to-right order, the macro `va_arg` is invoked once for each argument. The first argument to the macro `va_arg` is the variable of type `va_list`. The second argument is the type of the current argument of the function. The `va_arg` macro returns the value of the current argument of the function.

8. The **<varargs.h>** facility must be terminated by passing the variable of type `va_list` to a call of the macro `va_end` at the end of the function.

Example 12.2. Using the <varargs.h> Facility

```
#include <varargs.h>
/* Return the sum of a variable number of "int" arguments */
Sum(n, va_alist)
int n;
va_dcl                          /* steps 3 and 4 */
{
  va_list params;               /* step 5 */
  int ret = 0;
  va_start(params);             /* step 6 */
  while (n-- > 0) {
    ret += va_arg(params,int);  /* step 7 */
  }
  va_end(params);               /* step 8 */
  return(ret);
}
```

The <stdarg.h> Facility

The **<stdarg.h>** facility has some additional limitations. First, every function with variable parameters is required to have at least one fixed parameter. Second, a prototype for the function must appear before its first invocation.

To use the **<stdarg.h>** facility, you must perform the following steps:

1. The line `#include <stdarg.h>` must appear before the first function definition.
2. In the function definition, at least one fixed parameter must appear in the parameter list before the ellipsis (`...`).
3. The last parameter in the function's parameter list must be ellipsis (`...`).
4. There must be a variable declared in the function of type `va_list`.
5. The variable argument processing must be initialized by invoking the macro `va_start` at the beginning of the function with two parameters: (1) the variable of type `va_list`, and (2) the last fixed parameter in the function parameter list.

6. To obtain the variable parameters to the function in left to right order, invoke the macro `va_arg` once for each parameter. The first parameter to `va_arg` is the variable of type `va_list`. The second parameter is the type of the current parameter of the function. The `va_arg` macro returns the value of the current parameter of the function.
7. The variable argument processing must be terminated by invoking the macro `va_end` with the variable of type `va_list` as its parameter. If the function passes the `va_list` variable to another function such as `vprintf`, the calling function must invoke `va_end` before returning.

Example 12.3. Using the <stdarg.h> Facility

```
#include <stdarg.h>

/* Return the sum of the parameter, with the first
   parameter being a parameter type string. */
double sum(const char *key, ...)
{
  double ret = 0;
  va_list ap;
  va_start(ap, key);
  while (*key) {
    switch (*key++) {
    case 'i': ret += va_arg(ap, int); break;
    case 'l': ret += va_arg(ap, long); break;
    case 'd': ret += va_arg(ap, double); break;
    case 'p': ret += *(va_arg(ap, int*)); break;
    }
  }
  va_end(ap);
  return ret;
}
```

The asm Statement

The `asm` statement generates in-line assembly code; it can be used anywhere a statement can be used within a function and anywhere a declaration can be used outside of a function. There are two spellings: `__asm` and `asm`. `asm` is supported in both ISO C99 and ANSI C modes (**-c99** and **-ansi**), but not their strict counterparts (**-C99** and **-ANSI**). `__asm` is supported in all modes.

```
__asm ("assembler_string");
```

or

```
asm ("assembler_string");
```

The entire contents of the string will be passed through to the assembly language output file in the same position as it appears in the source file. If the underlying assembler requires a space or tab before the opcode, this tab or space must appear in the string.

See also Chapter 18, “Enhanced asm Macro Facility for C and C++” on page 847.

To control the diagnostic messages associated with `asm` statements (667 and 1546):

Set the **Compiler Diagnostics→C/C++ Messages→Asm Statements** option to one of the following settings:

- **Errors (--asm_errors)**
- **Warnings (--asm_warnings)**
- **Silent (--asm_silent)**

Note

The compiler uses assembly language instructions and directives throughout the file without concern for possible interactions with user `asm` statements. Furthermore, the allocation of variables to registers and memory may vary from one compilation to another. Therefore, the use of the `asm` statement is considered non-portable and may be difficult to maintain.

The Preprocessor

Green Hills C and C++ compilers include a built-in preprocessor. Preprocessing is normally performed in a single pass. The behavior of the preprocessor is affected by the **C/C++ Compiler→C Language Dialect** option.

Preprocessor Output File

By default, preprocessing is performed concurrently with compilation and no intermediate output is generated. To write preprocessed output to a file:

Pass the **-P** option to the driver.

Preprocessed output from a C or C++ file is saved with a **.i** extension, while output from an assembly file is saved with a **.si** extension.

You can also use the **-E** option to direct output to `stdout`. This latter option includes `#line` directives corresponding to the original file.

Extended Characters

English has a small alphabet, which is easily represented in eight bits by the ASCII character set. However, many languages require more space, and character sets that need more than 8 bits can be handled in two ways by ISO C/C++:

- *Multi-Byte Characters* — uses traditional C strings to represent the non-ASCII characters. For example, Kanji characters in the EUC or Shift-JIS format require two consecutive bytes to represent a single Kanji character. In both cases, the first byte has a value outside the normal ASCII range, which allows a program to recognize that the following byte should be treated as the rest of the current Kanji character. Multi-byte characters can be mixed with normal ASCII characters within the same string. Thus the length of a string in bytes does not directly correspond to the number of characters represented by the string. Some characters are one byte and others require more. Hence the term multi-byte.
- *Wide Characters* — are simply fixed-width (16- or 32- bit) characters. Wide characters were introduced by the ANSI/ISO C standard of 1990. The prefix "`L`" is used to mark a character or string literal as wide. The `wchar_t` type matches the type of `L'x'` and `L"hello"[3]`.

The multi-byte string looks just like a traditional string in C, where each element of the string has type `char`. The wide character string uses more space because each element of the string has type `wchar_t`, which occupies either 16 or 32 bits depending on the target and build options.

The advantages of wide characters over multi-byte characters are:

- The length of the string directly corresponds to the number of characters in that string. For example, the third character in a wide character string is always the third element, but in a multi-byte character string it could be one element or two. The character could even begin after the second, third, or fourth byte, depending upon whether there are Kanji or ASCII characters preceding it in the string.
- You can form a single wide character, such as `L'5'`, representing one multi-byte character.

However, wide characters take up more space and, more importantly, they cannot be manipulated by the large number of existing routines that were written for traditional C character strings, requiring instead special wide character functions.

Amendment 1 to the ISO C standard, adopted in 1995 and incorporated into both the ISO C++ standard and the ISO C99 standard, added many new functions to the standard C library in order to provide support for wide characters. These new functions, such as `wprintf()` and `wscanf()`, together with wide-character versions of most of the functions in `string.h` and `ctype.h`, have been added to **libansi.a**.

Note

In addition to requiring the new functions, the standard also modifies the behavior of `printf()` and `scanf()` and their derivatives to support wide character strings and individual wide characters. The Green Hills implementation does not provide these modifications in order to minimize the impact of such changes on the size, speed, and stability of existing applications.

Compiler Support for Multi-Byte Characters

The Green Hills Software C/C++ compilers deal with multibyte character sets in two different places and reject them elsewhere. Multi-byte characters may appear

in comments and in character or string literals, but they may not appear in variable names.

The compiler may not need to know everything about a multibyte character set in order to parse comments or character and string literals which contain multi-byte characters. The compiler *does* need to be able to detect the end of a string or comment. Therefore, if the multibyte character set contains no values which have bytes that match punctuation the compiler is looking for, such as `' \ "`, the compiler will be able to scan the comment or literal without requiring knowledge of the multibyte character set.

Parsing of wide character literals, however, is more difficult. The compiler must know the encoding of the multibyte character set in order to determine which bytes make up each element of type `wchar_t`. For example, `L"hello"` will result in a string of type `wchar_t[6]`, because each of the characters is ASCII (which the compiler understands). If, however, the five bytes in the string literal were not ASCII, the compiler must decide which bytes compose the first `wchar_t` element, and so on.

Applications requiring an unsupported multibyte character set will not be able to use wide character literals with the Green Hills Software C/C++ compilers. Instead, all literals must be written as multi-byte strings and converted using functions such as `mbstowcs()`. Furthermore, if the multibyte character set has bytes which match certain ASCII punctuation characters, multi-byte string literals may not be used either. In either case, it is possible to place multi-byte character strings in external data files and load them into the application at run time. The application can still make full use of the Green Hills C and C++ library routines to manipulate both wide character strings and multi-byte strings even if the compiler cannot parse string literals containing the multibyte character set.

The following list details the limitations to wide character support:

1. When using `wscanf()`, the behavior of the `%[]` modifier is defined only for characters in the range `0` to `0xffff`.
2. Since the multibyte character sets that we support directly do not require state information, our implementation ignores `mbstate_t`. Consequently support for state-dependent encoding is not provided.
3. Locale is ignored. All wide character conversion routines rely on `mbtowc()` and/or `wctomb()`, the behavior of these functions is set according to the **C/C++**

Compiler→Special Tokens→Host and Target Character Encoding option (**-kanji=**). For more information about this option, see “Host and Target Character Encoding” on page 203.

Kanji Character Support

In this manual, *extended characters* refers to the Japanese Kanji, Hiragana, and Katakana character sets and any other characters which are representable in your chosen character encoding (such as UTF-8), but are not members of the basic character set (C99, 5.2.1). To enable extended characters:

Set the **C/C++ Compiler→Special Tokens→Host and Target Character Encoding** option (**-kanji=**). For more information, see “Host and Target Character Encoding” on page 203.

The Green Hills C and C++ compilers provide support for extended characters in comments, strings, and character constants. Their use in variable names or other identifiers is not supported because of the conflicts it would cause with syntactic rules requiring identifiers to begin with a letter or underscore.

If a source file uses extended characters, specify the appropriate character encoding, or the source file might not be parsed correctly. For example, whereas the EUC encoding of a multibyte character uses two bytes outside the normal ASCII range, the Shift-JIS encoding allows normal ASCII characters in the second byte. The compiler must know that Shift-JIS is being used, in order to parse a string literal containing Shift-JIS characters where the second byte has the value `0x5C` (`\`) or `0x22` (`"`).

At link time, the Green Hills driver selects a run-time library containing versions of `wctomb()` and `mbtowc()` appropriate to the character encoding specified on the command line. The EUC and Shift-JIS character encodings share a single implementation.

Note

In Shift-JIS mode, half-width Katakana characters in the range `0xA1-0xDF` are not supported by the wide character functions in the run-time library.

Compiler Limitations

The ANSI standard requires that a conforming compiler implementation accept programs of a certain complexity and size. These requirements are expressed in terms of a list of acceptable limits. The Green Hills compiler meets the requirements. For completeness, we list below all of the limits addressed by the ANSI standard, giving, first, the number which the standard requires and, second in parentheses, the number which the Green Hills compiler supports. The code (U) indicates the compiler is theoretically unlimited.

The code (U > *n*) indicates that the compiler is theoretically unlimited, and it is tested to handle at least the value *n*.

- 15 nesting levels of compound statements, iteration control structures and selection control structures (U > 500)
- 8 nesting levels of conditional inclusion (U > 500)
- 12 pointer, array, and function declarators (in any combination) modifying an arithmetic, structure, union, or incomplete type in a declaration (U > 64)
- 31 nesting levels of parenthesized declarators within a full declarator (U)
- 32 nesting levels of parenthesized expressions within a full expression (U > 200)
- 31 significant characters in an external identifier (U)
- 511 external identifiers in one translation unit (U)
- 127 identifiers with block scope declared in one block (U)
- 1024 macro identifiers simultaneously defined in 1 translation unit (U)
- 31 parameters in one function declaration (U > 100)
- 31 arguments in one function call (U > 100)
- 31 parameters in one macro definition (U > 100)
- 31 arguments in one macro call (U > 100)
- 509 characters in a logical source line (U > 3000)
- 509 characters in a character string literal or wide string literal (after concatenation) (U > 3000)
- 8 nesting levels for `#include` files (U > 500)

- 257 `case` labels for a `switch` statement, excluding those for any nested `switch` statements (U)
- 127 members in a single structure or union (U)
- 127 enumeration constants in a single enumeration (U)
- 15 levels of nested structure or union definitions in a single `struct`-declaration-list (U)

C Implementation-Defined Features

The standard for the C programming language (*ISO/IEC 9899*) leaves certain details of the language unspecified. Throughout the standard, various details are described as "undefined" or "implementation-defined". The following definitions are taken from the standard for these and related terms:

Term	Meaning
implementation-defined behavior	Behavior, for a correct program construct and correct data, that depends on the characteristics of the implementation and that each implementation shall document.
undefined behavior	Behavior, upon use of a nonportable or erroneous program construct, of erroneous data, or of indeterminately valued objects, for which this International Standard imposes no requirements. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).
unspecified behavior	Behavior, for a correct program construct and correct data, for which this International Standard explicitly imposes no requirements.

The standard requires that a conforming implementation provide documentation which describes the behavior in each case where the standard uses the phrase "implementation-defined." This chapter describes the behavior of the Green Hills V850 and RH850 compiler in each such case. Features are listed according to the sequence in which they appear in the 1999 version of the standard, with citations referring to the appropriate section of the standard.

J.3.1 Translation

Each non-empty sequence of whitespace characters, other than newline, is replaced by one space character.

Diagnostic messages all have the form:

```
"filename", line nnn: content_of_message
                                    offending_source_line
```

There are multiple classes of messages. For more information, see “Varying Message Severity” on page 240.

The compiler returns one (indicating failure) if any errors were reported, and zero (indicating success) if there were no errors.

To suppress compiler warnings:

Set the **Compiler Diagnostics→Warnings** option to **Suppress** (**-w**).

J.3.2 Environment

To select the encoding used in translation phase 1 (as well as the encoding used by the standard library at run time), use the driver option **-kanji=**. For more information, see “Host and Target Character Encoding” on page 203.

No further processing occurs after program termination in a stand-alone environment.

In hosted execution environments, two arguments are passed to the function `main`:

```
int main(int argc, char *argv[]);
```

The argument `argc` is the number of valid elements in the `argv` array. The argument `argv` is a null-terminated array of command line words.

In a stand-alone environment, no arguments are passed to the function `main`.

`main()` may be defined with any prototype, but the default startup code expects it to be defined as `int main(int argc, char **argv, char **envp)`.

From the ISO C99 standard defined in *ISO/IEC 9899:1999*, "the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device" (section 7.19.3), and "what constitutes an interactive device is implementation-defined" (section 5.1.2.3). In a hosted environment that supports the `isatty()` system call, such as INTEGRITY, an interactive device is one for which `isatty()` returns a non-zero value. In a stand-alone environment, no streams are fully buffered until `setbuf()` or `setvbuf()` is called to change the buffering. For more information, see “Less Buffered I/O” on page 773.

The set of signals is documented in **<signals.h>**. Signal values that correspond to a computational exception are: `SIGFPE`, `SIGILL`, `SIGSEGV`, `SIGQUIT`, `SIGDRAP`, `SIGIOT`, `SIGABRT`, `SIGEMT`, `SIGBUS`, and `SIGSYS`.

The equivalent of `signal(sig, SIG_DFL)` is executed prior to the call of a signal handler. Default handling is reset if a `SIGILL` signal is received by a handler specific to the signal function.

The only limitation on environment names is that they may not contain the equals character (=) which is used to separate the name from its value.

The method to alter the environment list obtained by a call to the `getenv` function is to call the function `setenv(char *name, char *value)`. This will set the environment variable named by the null-terminated string `name` to a copy of the null-terminated string pointed to by `value`.

`system()` is not supported on stand-alone targets.

J.3.3 Identifiers

Multibyte characters may not appear in identifiers, although universal character names may.

There is no limit to the number of significant initial characters in an identifier.

Case is significant in an identifier with external linkage.

J.3.4 Characters

There are eight bits in a character in the execution character set.

The basic execution character set includes all possible single-byte characters and the multibyte character set includes all possible 4-byte characters.

The escape sequence values are as follows.

Sequence	Value
`\a`	7
`\b`	8
`\f`	12
`\n`	10
`\r`	13
`\t`	9
`\v`	11

`char` is an 8-bit arithmetic type with the usual properties.

The character type `char` is `signed` by default. To control the signedness of `chars` manually, see “Signedness of Char Type” on page 197.

The source and execution character sets are identical and the mapping is direct.

An integer character constant may contain up to four characters. The value is calculated as follows: the least significant byte of the integer is the ASCII code for the rightmost character. The next byte of the integer is the ASCII code for the second character from the right, and so on.

A wide character constant that contains more than one multi-byte character has the same integer value as a wide character constant consisting only of the rightmost multi-byte character.

The C locale is used to convert multi-byte characters into corresponding wide characters (codes) for a wide character constant. The C locale is documented in *X3J11/90-013* under section 7.4.1.1 and in *ISO/IEC 9899:1999* under section 7.11.1.1. Only the C locale is implemented.

Any byte `B` which does not begin a valid multi-byte character is treated as a single-byte multibyte sequence mapping to the value `(unsigned char)B`.

J.3.5 Integers

The table below shows the storage and range of various variable types:

Designation	Size (bits)	Range
`char`	8	If `char` is signed, `-128..127` If `char` is unsigned, `0..255` For information about the signedness of `char`, see "Signedness of Char Type" on page 197.
`signed char`	8	`-128..127`
`unsigned char`	8	`0..255`
`short`	16	`-32768..32767`
`unsigned short`	16	`0..65535`
`int`	32	`-2147483648..2147483647`
`unsigned int`	32	`0..4294967295`
`long`	32 or 64	`-2147483648..2147483647` or `-9223372036854775808..9223372036854775807`
`unsigned long`	32 or 64	`0..4294967295` or `0..18446744073709551615`
`long long`	64	`-9223372036854775808..9223372036854775807`
`unsigned long long`	64	`0..18446744073709551615`

Conversion of an unsigned integer to a signed integer of equal length does not modify the bit pattern of the number. Signed integers are represented in two's complement. Therefore such a conversion on a number with the high order bit set produces a negative number equal to $original\_value - 2^n$, where *n* is the number of bits.

Extended integer types are not supported on stand-alone targets.

Truncation occurs when an integer is converted to a shorter signed integer.

The result of bitwise operations on signed integers is a signed integer whose two's complement representation is the result of applying the bitwise operation on the two's complement representation of the arguments.

The sign of the remainder of integer division will have the same sign as the dividend, if the remainder is not zero.

The right shift of a negative-value signed integral type produces a signed integral value, arithmetically shifted to the right.

J.3.6 Floating-Point

The standard library's floating-point routines are intended to be sufficiently accurate for normal use, but make no guarantees about worst-case accuracy in terms of ULP.

`FLT_ROUNDS` has no implementation-defined values.

`FLT_EVAL_METHOD` has no implementation-defined values.

The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value is to the nearest representable value. In other words, the value is rounded.

The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number is to the nearest representable value.

At compile time, the value for floating constants is chosen using the same method as is used by the standard library's `strtof()`, `strtod()`, or `strtold()` function.

The `FP_CONTRACT` `#pragma` directive is recognized in C99 mode but has no effect on code generation. The compiler may contract floating-point expressions when optimizing for speed or size.

The `FENV_ACCESS` `#pragma` directive is recognized in C99 mode but not obeyed.

There are no implementation-defined floating-point exceptions, rounding modes, environments, or classifications.

The storage requirements and the ranges of the floating-point types are shown below:

Designation	Size (bits)	Range
`float`	32	`1.1754943635e-38` to `3.4028235e+38`
`double`	64	`2.2250738585072015e-308` to `1.7976931348623158e+308`
`long double`	64	`2.2250738585072015e-308` to `1.7976931348623158e+308`

J.3.7 Arrays and Pointers

Casting a pointer to an integer type or vice versa has no effect on the bit pattern of the value. Converting to a smaller type truncates bits, and converting to a larger type sign-extends or zero-extends, depending upon the setting of **--signed_pointers**/**--unsigned_pointers** in the case of pointer-to-integer conversion.

Pointer subtraction yields a result of type `ptrdiff_t`, which is a signed version of `size_t` and has the same size as a pointer.

J.3.8 Hints

The register storage-class specifier is essentially ignored when making decisions about register allocation.

Functions declared as `inline` are generally inlined. When optimizing for size, less inlining is performed. You can disable most inlining with the **–no_inlining** option.

J.3.9 Structures, Unions, Enumerations and Bit-fields

For information about controlling the signedness of bitfields, see “Signedness of Bitfields” on page 647.

Members of structures are padded and aligned as described in “V850 and RH850 Characteristics” on page 26. All structure members except bitfields begin on a byte boundary. A structure is aligned according to the strictest alignment of any of its members. An array member is aligned according to the alignment requirement of its elements. Padding is performed when necessary to ensure that a member or

bitfield begins on its required alignment. There is never any padding before the first member or bitfield, but there is padding after the last member or bitfield if it is needed to ensure that the size of the structure is a multiple of its alignment.

Bitfields may be declared with any standard integer base type (including `signed long long` and `unsigned long long` if supported). They may also be declared with `_Bool` or `enum` base types. When a bitfield is used in an integral context, its value is converted to a standard integer type in the following way. All standard integer types with rank greater than or equal to `int` are ordered from least to greatest rank with signed types preceding unsigned types of the same rank. The value is then converted to the first type in this ordering that can represent all of the values of the original bitfield type. For example, on a target where the `long` type has the same number of bits as the `int` type and the `long long` type is supported, a bitfield value would be promoted as follows. If an `int` can represent all values of the original type, the value is converted to an `int`; otherwise, if an `unsigned int` can represent all values of the original type, the value is converted to an `unsigned int`; otherwise, if a `long long` can represent all values of the original type, the value is converted to a `long long`; otherwise, it is converted to an `unsigned long long`.

A non-packed bitfield must be wholly contained within a storage unit meeting the size and alignment requirements of its declared base type. Padding is inserted between bitfields as required to satisfy this constraint. For example:

```
struct {
    int       a:20;
    int       b:20;
    long long c:20;
};
```

On a target with 32-bit `int`s with 32-bit alignment, there will be 12 bits of padding between `a` and `b` since `b` cannot cross a 32-bit aligned boundary. On a target with 64-bit `long long`s with 64-bit alignment, there will also be 12 bits of padding between `b` and `c` since `c` cannot cross a 64-bit aligned boundary. On a target with 64-bit `long long`s with 32-bit alignment, there will be no padding between `b` and `c` since a `long long` can cross one 32-bit aligned boundary; however, if `c` were declared to contain more than 44 bits, there would then be 12 bits of padding between `b` and `c` since a 32-bit aligned `long long` cannot cross two 32-bit aligned boundaries.

As long as the above constraint is met, multiple bitfields, even those of different base types, may occupy the same storage unit. Within a particular storage unit, bitfields are allocated in increasing memory address order. This means on a big-endian target the first bitfield will occupy the most significant bits of the storage unit, and on a little-endian target the first bitfield will occupy the least significant bits. For example:

```
struct {
    int   a:10;
    char  b:4;
    short c:5;
};
```

Assuming that `int`s are 32 bits, `short`s are 16 bits, `char`s are 8 bits, and the alignment of each type matches its size, there will be no padding between `a` and `b`, but there will be 2 bits of padding between `b` and `c`. All three fields will occupy one 4-byte storage unit, with `a` occupying the first byte, `a` and `b` sharing the second byte, and `c` occupying the third byte.

For information about controlling the size and type of enumerations, see “Enumerated Types” on page 650.

J.3.10 Qualifiers

A volatile access occurs when a volatile object's value is either read or modified. If the volatile object is in memory, the compiler generates load and store instructions for each access. The compiler might still optimize away an access to an automatic (local) variable. As required by the standard, though, on return from `setjmp`, an automatic volatile variable has the value it had as of the corresponding call to `longjmp`.

J.3.11 Preprocessing Directives

No `#pragma` directives allow header names.

Header names are passed through to the operating system unchanged. For example, `#include "dir\x30"` refers to a file named **dir\x30**, not to a file named **dir0**.

The value of a single character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Such a character constant can have a negative value.

If the file is included using the form `#include "file"`, the directory containing the file in which the `#include` directive occurs is searched first. If the directory does not contain the file, or if the file is included using the form `#include <file>`, a list of user-specified directories is searched. For information about specifying include directories, see "Using Your Own Header Files and Libraries" on page 70.

A header name must correspond to a single preprocessing token. There is no pasting of adjacent string literals.

There is no nesting limit for `#include` processing.

The `#` operator does not escape the backslash in `\u` when it stringizes a universal character name.

For a list of the recognized `#pragma` directives, see "Pragma Directives" on page 749.

The `__DATE__` and `__TIME__` macros always have a time available.

J.3.12 Library Functions

The contents of the Green Hills libraries are documented in "Green Hills Standard C Library Functions" on page 782.

Given the following file **test.c**:

```
#include <stdio.h>
#include <assert.h>

int main()
{
    assert(1==2);
    return 0;
}
```

the program would print the following to `stderr` and then call `abort()`.

```
test.c, line 6: Assertion failed: "1==2"
```

The floating-point status flags stored by `fegetexceptflag()` are documented in **<fenv.h>**.

`feraiseexcept()` does not raise any exceptions that were not explicitly requested.

No strings other than `""` and `"C"` may be passed as the second argument to `setlocale()`.

`float_t` and `double_t` are defined as `float` and `double`, respectively, when `FLT_EVAL_METHOD` is less than zero.

In the case of a domain error, mathematics functions return an appropriate value, usually a positive or negative zero, NaN, or infinity.

Only the functions `exp()`, `pow()`, and `tan()` detect underflow and set `errno` to `ERANGE`.

When `fmod()` has a second argument of zero, a domain error occurs and zero is returned.

When a remainder function has a second argument of zero, a domain error occurs and the first argument is returned.

The base-2 logarithm of the modulus used by `remquo` functions is the size of `int` in bits (32).

When a `remquo` function has a second argument of zero, a domain error occurs and the first argument is returned.

Signal handling does not happen on stand-alone targets.

`NULL` expands to `((void*)0)` for the C language. For C++, the `NULL` macro expands to either `0` or `0L`.

The last line of a text stream does not require a terminating newline character.

Space characters written out to a text stream immediately before a newline character appear when the stream is read back.

No null characters are appended to data written to a binary stream.

The file position indicator of an append mode stream is initially positioned at the end of the file.

A write on a text stream causes the associated file to be truncated beyond that point.

Unbuffered, fully buffered, and line buffered modes are all provided in hosted environments. Embedded environments provide a limited form of buffering which is essentially unbuffered, but which provides the benefits of buffering within operations such as `fwrite()` and `printf()`.

A zero length file can exist.

The rules for composing a valid filename differ between systems.

The same file can be opened multiple times.

To select the encoding used in the encoding of files, use the driver option **-kanji=** (see “Host and Target Character Encoding” on page 203).

Removing an open file has unpredictable results. In most cases the application program will only have access to a portion of the file as it was before it was deleted. Once the application exits, the file will cease to exist entirely even if the application was writing to the file.

The effect of renaming a file to a name of a file which already exists varies between systems. On some systems it is undetected and the old file is lost. On others, the rename fails and all files remain unchanged.

An open temporary file is not removed if the program terminates abnormally.

The semantics of `freopen()` depend on the file system, but generally any file can be reopened with any mode.

Any NaN is printed as `nan`. Infinity is printed as `inf` (for positive infinity) or `-inf` (for negative infinity).

The `%p` format descriptor in `printf()` behaves exactly as `%lx`.

If the - is neither the first character nor the last character in the scan list for `%[` conversion in the `fscanf` function, all characters in the range beginning with the character preceding the - up to and including the character following the - are

added to the scan set. If the character following the - lexically precedes the character before the -, then the - and the character following it are ignored.

The `%p` format descriptor in `scanf()` behaves exactly as `%lx`.

The functions `fgetpos()` and `ftell()` set `errno` to `EBADF` on failure.

When a string representing a NaN is converted by `strtod()`, `strtof()`, `strtold()`, `wcstod()`, `wcstof()`, or `wcstold()` any `n-char` or `n-wchar` sequence between parentheses is ignored.

The above functions do not set `errno` to `ERANGE` when underflow occurs.

The functions `calloc()` and `malloc()` return a valid pointer if the size requested is zero. When you request a size of zero with the `realloc()` function, it returns `NULL` if you pass it a valid pointer, or a valid pointer if you pass it `NULL`.

When `abort()` is called, the buffers within the **stdio** library for open files are not flushed. The files are closed and exist. Temporary files are not deleted.

The `abort()` function in the stand-alone library is equivalent to calling `raise(SIGABRT)` followed by `exit(EXIT_FAILURE)`. It does not call C++ destructors for global objects or functions registered with `atexit()`.

The low-order eight bits of the argument passed to `exit()` are returned as the status.

The `exit()` function in the stand-alone library is equivalent to calling the functions registered by `atexit()`, followed by `_Exit(`*`status`*`)`.

`system()` is provided by the operating system; it generally returns the result of the executed command.

The era for the `time()` function's Local Specific Behavior is midnight, January 1, 1970 (GMT).

The replacement string for the `%Z` specifier to `strftime()` and `wcsftime()` is one of `AST`, `EST`, `CST`, `MST`, `PST`, `AKT`, `HIT`, `MET`, `EET`, `WET`, or three space characters, depending on the return value of `__gh_timezone()`.

`fesetround()` and `fegetround()` are recognized but have no effect.

The macros specified in **<float.h>**, **<limits.h>**, and **<stdint.h>** are documented in their respective header files.

The sizes of the basic types are specified in "J.3.5 Integers" on page 664. See also "Data Type Macros" on page 740.

For the layout of `struct` and `union` types, see "Structure Packing" on page 28.

On a 32-bit target, the range of times representable in a value of type `time_t` is from Dec 13 20:45:52 1901 (`-2147483648` or `0x80000000`) to Jan 19 03:14:07 2038 (`2147483647` or `0x7fffffff`). The precision is one second.

The following table explains the return values of the listed functions for certain characters:

Function	Characters for which non-zero value is returned
`isalnum()`	`'A'..'Z'`, `'a'..'z'`, and `'0'..'9'`
`isalpha()`	`'A'..'Z'` and `'a'..'z'`
`iscntrl()`	`0..31` and `127`
`islower()`	`'a'..'z'`
`isprint()`	`32..126`
`isupper()`	`'A'..'Z'`

J.4 Locale-Specific Behavior

The characters in the execution set in addition to those required by the C standard are listed in "J.3.4 Characters" on page 663.

There are no shift states in the encoding of multi-byte characters.

The direction of printing is left to right.

The decimal point character is a period (`.`).

The printing characters are those in the range `0x20` to `0x7E`, inclusive.

The control characters are those in the range `0x00` to `0x1F`, inclusive, and the single character `0x7F`.

`isalpha()`, `isupper()`, `islower()`, and `isdigit()` return the expected values. `isblank()` returns true for `0x09` and `0x20`. `isspace()` returns true for `0x09`, `0x0a`, `0x0b`, `0x0c`, `0x0d`, and `0x20`. `ispunct()` returns true for any printing character not covered by the other functions. `iswfoo(wc)` returns the same value as `isfoo((char)wc)`.

Only the C locale is implemented. The C locale is documented in *X3J11/90-013* under section 7.4.1.1 and in *ISO/IEC 9899:1999* under section 7.11.1.1.

For a detailed description of the C locale's values related to `strftime()`, see *ISO/IEC 9899:1999*, section 7.23.3.4, paragraph 7.

The program is run in the local time zone and Daylight Savings Time.

No non-standard sequences are supported by the numeric conversion functions.

The collation sequence of the execution character set is the natural numerical order.

There is little formatting style to the string returned by `strerror()`. The first character is always capitalized and the string does not end with a period.

The function `perror()` prints a message in this format:

`argument``: `*`contents_of_message`*

where *argument* is the parameter passed to `perror` and *contents_of_message* is a predefined message associated with the current value of `errno`.

The strings returned by `strerror` for the various values of `errno` are listed below:

00: `"Error 0 (no error)"`
01: `"Not owner"`
02: `"No such file or directory"`
03: `"No such process"`
04: `"Interrupted system call"`
05: `"I/O error"`
06: `"No such device or address"`
07: `"Arg list too long"`
08: `"Exec format error"`
09: `"Bad file number"`
18: `"Cross-device link"`
19: `"No such device"`
20: `"Not a directory"`
21: `"Is a directory"`
22: `"Invalid argument"`
23: `"File table overflow"`
24: `"Too many open files"`
25: `"Not a typewriter"`
26: `"Text file busy"`
27: `"File too large"`

10: "No children"
11: "No more processes"
12: "Not enough core"
13: "Permission denied"
14: "Bad address"
15: "Block device required"
16: "Mount device busy"
17: "File exists"
28: "No space left on device"
29: "Illegal seek"
30: "Read-only file system"
31: "Too many links"
32: "Broken pipe"
33: "Argument too large"
34: "Result too large"
35: "Non standard error"

The standard library's `towctrans()` provides only what is required by the Standard. `towupper(wc)` returns `(unsigned char)toupper((char)wc)`, and `towlower(wc)` returns `(unsigned char)tolower((char)wc)`.

Only the classifications required by the Standard are supported by `iswctype()`.

Character Testing and Case Mapping

The execution character set in its natural sequence is the ASCII character set.

00 nul	01 soh	02 stx	03 etx	04 eot	05 enq	06 ack	07 bel
08 bs	09 ht	0A nl	0B vt	0C np	0D cr	0E so	0F si
10 dle	11 dc1	12 dc2	13 dc3	14 dc4	15 nak	16 syn	17 etb
18 can	19 em	1a sub	1b esc	1c fs	1d gs	1e rs	1f us
20 sp	21 !	22 "	23 #	24 $	25 %	26 &	27 '
28 (	29)	2A *	2B +	2C ,	2D -	2E .	2F /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3A :	3B ;	3C <	3D =	3E >	3F ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4A J	4B K	4C L	4D M	4E N	4F O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5A Z	5B [	5C \	5D]	5E ^	5F _
60 `	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6A j	6B k	6C l	6D m	6E n	6F o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w

78 x	79 y	7A z	7B {	7C |	7D }	7E ~	7F del

The `strftime()` function converts `%c`, `%x`, and `%X` as follows:

`"%c"`	is equivalent to	`"%a %b %d %H:%M:%S %Y"`
`"%x"`	is equivalent to	`"%a %b %d, %Y"`
`"%X"`	is equivalent to	`"%H:%M:%S"`

Additional Implementation Information

Defining a static function with the reserved name of a standard library function results in undefined behavior. For more information, see the C99 standard, section 7.1.3, paragraph 2.

The `strtof()`, `strtod()`, and `strtold()` functions return the smallest non-zero value with the same sign as the correct value. They then set `errno` to `ERANGE`.

Attributes

The Green Hills compiler supports the `__attribute__` keyword, which can be used to annotate program entities with pertinent information that is not handled by Standard C or C++. The syntax for `__attribute__` is the same as that supported by the GNU compiler, but some GNU attributes are not supported by Green Hills compilers and some Green Hills attributes are not supported by the GNU compiler. If an attribute name is not listed in this section, it is not supported by Green Hills, even if it is recognized by the compiler.

`alias`
: Annotating the declaration of a function *foo* with `__attribute__((alias("`*bar*`")))` causes the compiler to refer to *foo* by the name *bar* in the generated assembly or object file.

  Aliases containing special symbols such as `.` and `$` might not be handled correctly. Aliases that are treated as multiple tokens by the assembler, such as `bar+1`, are not escaped.

`aligned`
: Annotating the definition of a function or variable with `__attribute__((aligned(`*x*`)))` causes the compiler to force the start of

that entity onto an *x*-byte boundary. In the case of local automatic variables, *x* must not exceed the stack alignment enforced by the current calling convention (usually 4, 8, or 16 bytes).

Annotating a type definition with `__attribute__((aligned(x)))` causes the compiler to align every variable and field of that type to an *x*-byte boundary. For example:

```
typedef int IntVector[16] __attribute__((aligned(8)));

struct S {
    int i;
    IntVector iv; // will be preceded by 4 bytes of padding
};

void foo() {
    static IntVector ivs; // 8-byte-aligned
    IntVector iva; // 8-byte-aligned if allowed by calling convention
}
```

`at`

Annotating a field definition with `__attribute__((at(x)))` causes the compiler to issue diagnostic 1947 if the field does not indeed start at offset *x* (in bytes) from the beginning of the current struct type. For example:

```
struct S {
    union {
        float f;
        long l;
    } u;
    struct S *next __attribute__((at(4)));
    // will compile silently in the case that u is 4 bytes long
    // and next does not need to be preceded by any padding bytes,
    // but will produce an error if long is 8 bytes
    // or if pointers require 8-byte alignment
};
```

`common`, `nocommon`

Annotating the definition of an uninitialized global variable with `__attribute__((common))` forces it into commons, even when **--no_commons** is in effect. Annotating the definition with `__attribute__((nocommon))` forces it to be defined as a non-common symbol, even when **--commons** is in effect. For more information about common symbols, see the **C/C++ Compiler→C/C++ Data Allocation→Allocation of Uninitialized Global Variables** option in "C/C++ Data Allocation" on page 200.

`deprecated`

Annotating an entity with `__attribute__((deprecated))` causes any future use of that entity (for example, calling a deprecated function or declaring a variable of a deprecated type) to issue a warning. Related diagnostics are 1215 and 1441.

`entry_exit_log`

Annotating a function with `__attribute__((entry_exit_log(1)))` causes the compiler to instrument the function for entry/exit logging, regardless of compiler options.

Annotating a function with `__attribute__((entry_exit_log(0)))` causes the compiler not to instrument the function for entry/exit logging, regardless of compiler options.

For more information, see "Enabling Function Entry/Exit Logging" on page 59.

`externally_visible`

Annotating a function or variable declaration with `__attribute__((externally_visible))` tells the compiler that there may be unseen references to the symbol even when the code is compiled with **-Owholeprogram** (see "Wholeprogram Optimizations" on page 310).

`final`

Annotating the declaration of a C++ virtual member function with `__attribute__((final))` tells the compiler that although this function might override a virtual member function of a parent class, it will never be overridden by any child of this class. This allows the compiler to optimize a virtual call into a non-virtual call, in cases where the actual type of the object being invoked can be deduced at compile time.

Annotating a class type definition with `__attribute__((final))` is equivalent to annotating every virtual member function of that class with the same attribute.

`format`

The `format` attribute annotates a function prototype for argument checking. Green Hills implements argument checking for several different standard library functions.

`__attribute__((format(printf,`*m*`,`*n*`)))` means that this function behaves like `printf()`; the *m*th argument is a format string and the (possibly empty)

set of arguments starting with the *n*th are the values to be matched with the format specifiers in the format string. If *n* is zero, then no matching is performed, but the format string is still checked to see if each individual specifier is valid.

The format string must conform to the ISO C99 specification for `printf()`; for example, `%d` must expect an argument of type `int`. Related diagnostics are 181, 224, 225, 226, and 1932.

`__attribute__((format(scanf,`*m*`,`*n*`)))` means that this function behaves like `scanf()`; the *m*th argument is a format string and the (possibly empty) set of arguments starting with the *n*th are the values to be matched with the format specifiers in the format string. If *n* is zero, then no matching is performed, but the format string is still checked to see if each individual specifier is valid.

The format string must conform to the ISO C99 specification for `scanf()`; for example, `%d` must expect an argument of type `int *`. Related diagnostics are 181, 224, 225, 226, and 1932.

`__attribute__((format(memset,`*x*`,`*y*`)))` means that this function takes arguments similar to `memset()`; the *x*th argument is a pointer to an object or array of objects, and the *y*th argument is a count of bytes. The compiler will check that the *y*th argument is of the form *n*`*sizeof(`*e*`)` where *n* is an arbitrary expression and *e* has the same type as pointed to by the *x*th argument. Related diagnostics are 1956, 1990, and 1991.

The compiler does not check whether the *x*th argument is writeable; the function is not expected to behave like `memset()` in that way. For example:

```
// Build with --diag_warning=1956,1990,1991 to see the diagnostics.
void mem_zero(void *p, int n) __attribute__((format(memset,1,2)));
void *realloc(void *p, int n) __attribute__((format(memset,1,2)));

void f(int *a) {
    mem_zero(a, 4); // produces diagnostic 1990
    mem_zero(a, 4*sizeof(char)); // produces diagnostic 1991
    a = realloc(a, 1); // produces diagnostics 1956 and 1990
}
```

`__attribute__((format(memcpy,`*x*`,`*y*`,`*z*`)))` means that this function takes arguments similar to `memcpy()` or `memcmp()`; the *x*th and *y*th arguments are pointer to objects or arrays of objects, and the *z*th argument is a count of bytes. The compiler will check that the *x*th and *y*th arguments are of the same type (or that at least one of them is a pointer to void or a character type); the compiler

will also check that the *z*th argument is of the form `n*sizeof(e)` where *n* is an arbitrary expression and *e* has the same type as pointed to by the *x*th and *y*th arguments.

The compiler does not check whether the *x*th argument is writeable; the function is not expected to behave like `memcpy()` in that way.

`format_arg`

Annotating a function prototype with `__attribute__((format_arg(m)))` tells the compiler that the function might be used to preprocess a format string for passing to a function that is similar to `printf()`. The compiler will allow a call to this function anywhere that a format string argument would normally be expected, as long as the *m*th argument to that call is itself (recursively) a valid format string. For example:

```
// Build with -Wformat --diag_warning=181,1932 to see the diagnostics.
#include <stdio.h>
const char *bad(const char *x);
const char *good(const char *x) __attribute__((format_arg(1)));
void f() {
    printf(bad("hello %d\n"), 42); // produces diagnostic 1932
    printf(bad("hello %s\n"), 42); // produces diagnostic 1932
    printf(good("hello %d\n"), 42); // silently accepted
    printf(good("hello %s\n"), 42); // produces diagnostic 181
}
```

`ghs_noprofile`

Annotating a function with `__attribute__((ghs_noprofile))` indicates that the function is not expected to be covered when a **-coverage=** option is enabled. MULTI excludes this function and any code paths containing calls to it from coverage calculations.

`nonnull`

Annotating a function prototype with `__attribute__((nonnull(m)))` tells the compiler that the *m*th argument should never be the constant `NULL`. The compiler will produce diagnostic 1583 if `NULL` is ever passed in this argument position.

`noreturn`

Annotating a function prototype with `__attribute__((noreturn))` tells the compiler that this function never returns to its caller, and therefore any code following the function call may be considered dead code and optimized away. For example, this attribute can be used on the standard library functions `exit()`, `abort()`, and `longjmp()`.

`nothrow`

Annotating a C++ function prototype with `__attribute__((nothrow))` tells the compiler that this function never throws an exception. This is different from providing an empty throw-specification, because a throw-specification always causes extra run-time code to be generated. `__attribute__((nothrow))` is a lightweight way to tell the compiler that no exception-handling code need be generated for this function. However, Green Hills does not guarantee that the exception-handling code will actually be removed.

`pure`

Annotating a function prototype with `__attribute__((pure))` tells the compiler that this function is *pure*: the state of the world is the same before and after a call to this function. A pure function may read memory, throw a C++ exception, or even enter an infinite loop, but a pure function may not modify global memory (for example, it may not write to a global variable, or write through a pointer). Marking a function as pure may improve the compiler's ability to perform alias analysis optimizations.

`section`

Annotating a function or variable definition with `__attribute__((section("`*name*`")))` causes the compiler to place that function or variable in a user-defined section with the specified name. The name must not be identical to a predefined section name such as `.text` or `.data`. The effect is similar to that of `#pragma ghs section data="`*name*`"`, except that only a single variable or function is affected. For more information about this directive, see "Assigning Data to Custom Program Sections in C" on page 78.

`sentinel`

Annotating a function prototype with `__attribute__((sentinel(`*m*`)))` tells the compiler that this variadic function uses a sentinel value to find the end of its variable argument list. The compiler will check that, in each call to this function, the *m*th argument from the end of the argument list (where 0 represents the last argument) is a constant null pointer. For example:

```
#include <stdlib.h>
#include <stdarg.h>

void print_result(int);
void __attribute__((sentinel(1))) for_each(int *p, ...)
{
    typedef int (*FuncType)(int);
    FuncType f;
    int sum = *p;
```

```
        va_list ap;
        va_start(ap, p);
        while ((p = va_arg(ap, int*)) != NULL)
            sum += *p;
        f = va_arg(ap, FuncType);
        f(sum);
    }
    int a,b,c,d,e;
    void f() {
        for_each(&a, &b, &c, NULL, print_result);
        for_each(&a, &b, &c, &d, &e, NULL, print_result);
        for_each(&a, &b, &c, NULL); // gives warning 1462
    }
```

`stackcheck`

Allows you to enable stack checking on a per-function basis. To enable stack checking for a function, use one of the following forms:

```
__attribute__((stackcheck))
__attribute__((stackcheck(1)))
```

If you are using the **-stack_check** option and want to disable stack checking for a function, use:

```
__attribute__((stackcheck(0)))
```

`strong_fptr`

Annotating a function pointer `typedef` with `__attribute__((strong_fptr))` marks the type as a strong function pointer and adds restrictions to how objects of that type may be used in order to enhance static call graph analysis. For more information, see “The gstack Utility Program” on page 560.

Legal operations for strong function pointers are:

- assignment of a function name to a strong function pointer object (as long as other C rules allow it)
- assignment of a strong function pointer object to any object of the exact same type
- calls as with any other C function pointer
- assignment of the return value of `malloc()` (or similar, such as `calloc()`) to a type that points to a strong function pointer

For example:

```
typedef                                int (*fptr_t)(void);
typedef __attribute__((strong_fptr)) int (*sfptr_t)(void);

fptr_t  fp1;
sfptr_t sfp1, sfp2;
sfptr_t *psfp;

int func(void);

sfp1 = func;  /* Legal - assign 'func' to a strong function pointer of
                 type 'sfptr_t' */

sfp2 = sfp1; /* Legal - assignment to an object of the same type */

(*sfp1)();   /* Legal - function call through the strong_fptr */

psfp = (sfptr_t*)malloc(sizeof(sfptr_t) * 10);
             /* Legal - assignment of the return value of malloc to a pointer
                to strong function pointer */
```

Operations that are not legal:

- assignment of a strong function pointer value to an object that is not a strong function pointer
- assignment of a non-strong function pointer value that is not a function name and not `NULL` to a strong function pointer object
- assignment of a strong function pointer value to a strong function pointer object of a different type

For example:

```
typedef __attribute__((strong_fptr)) int (*sfptr1_t)(void);
typedef __attribute__((strong_fptr)) int (*sfptr2_t)(void);

sfptr1_t sfp1;
sfptr2_t sfp2;
int      (*fp3)(void);
...
fp3  = sfp1;   /* Illegal - strong function pointer to non-strong */
sfp2 = fp3;    /* Illegal - non-strong function pointer to strong */
sfp1 = sfp2;   /* Illegal - different strong function pointer types */
```

Use of a strong function pointer type inside another type construct, such as a struct, causes the containing type to take on properties similar to the above. For example, it is illegal to cast a pointer to a struct to a different struct type if either has a member that is a strong function pointer.

Annotating a struct or C++ class with this attribute creates a unique strong function pointer type for each function pointer field of the structure, provided that the field does not already have a strong function pointer type. For example:

```
typedef __attribute__((strong_fptr)) int (*sfptr_t)(void);

struct __attribute__((strong_fptr)) S {
    /* strong function pointer types automatically created for fields
       'op1', 'op2', and 'op3' */
    void (*op1)(int, int);
    void (*op2)(int);
    int  (*op3)(void);

    /* 'op4' already has a strong function pointer type */
    sfptr_t op4;

    /* non-function pointer fields are not affected */
    int  i, j, k;
};
```

A struct or C++ class is considered to have strong function pointers if:

- any of its direct fields are strong function pointers, or
- any of its fields are another struct or C++ class that contains strong function pointers, or
- (for C++ classes) any of its base classes contain strong function pointers

Having a field that is a pointer to a strong function pointer or a struct or class containing a strong function pointer does not cause a struct or C++ class to be treated as having strong function pointers. The following are considered to have strong function pointers:

```
typedef __attribute__((strong_fptr)) int (*sfptr_t)(void);

struct S1 {
   sfptr_t fp;        /* directly contains a strong function pointer */
};

struct S2 {
   struct S1 s1;      /* field is a struct with a strong function pointer */
};

struct __attribute__((strong_fptr)) S3 {
   int (*myfp)(void); /* struct marks all function pointers as strong */
};
```

And in C++:

```
class Base {
   sfptr_t fp;
```

```
};

class Derived : public Base {  /* base class contains strong function
                                  pointer */
   //...
};
```

However, the following are not considered to have a strong function pointer:

```
struct Z1 {
   sfptr_t *pfp;     /* just a pointer to a strong function pointer */
};

struct Z2 {
   struct S2 *ps2;  /* just a pointer to a struct with a strong
                       function pointer */
};
```

A struct or C++ class can be marked with `__attribute__((strong_fptr))` as long as:

- no field members are a struct or class type that contain a function pointer that is not strong
- (for C++ classes) no base class contains a function pointer that is not strong

For example:

```
struct P1 {
   int (*myfp)(void);
};

struct __attribute__((strong_fptr)) P2 {
   struct P1 p1;     /* Illegal - P1 contains non-strong function pointer */
};
```

And in C++:

```
class Base {
   int (*myfp)(void);
};

class __attribute__((strong_fptr)) Derived
    : public Base {  /* Illegal - Base contains non-strong function pointer */
    //...
};
```

The following operations are allowed on structs and C++ classes that contain a strong function pointer:

- a struct pointer can be assigned to `void *` (and vice versa), but it is user error to assign a struct pointer of one object to `void *` and then later cast

it to a different object. The compiler does not issue any errors for this illegal case.

- a pointer to the type of the first field of the struct can be assigned to the struct pointer as long as there are no other fields with strong function pointers.

For example:

```
typedef __attribute__((strong_fptr)) int (*sfptr_t)(void);
struct S1 {
   sfptr_t fp;
};
struct S2 {
   struct S1 s1;
};

void      *pv;
struct S1 *ps1;
struct S2 *ps2;
struct S3 *ps3;
...

pv = ps1;                   /* Legal - assign struct pointer to 'void *' */
ps1 = pv;                   /* Legal - assign 'void *' to struct pointer */

ps2 = (struct S2 *)ps1;     /* Legal - assign struct pointer to struct
                               whose first field has the same type */

ps3 = pv;                   /* Illegal - assign 'void *' to different
                               struct type (no compiler error given) */

ps3 = (struct S3 *)ps1;     /* Illegal - assign struct pointer to different
                               struct type */
```

`__attribute__((strong_fptr))` is not legal for unions.

The attribute may also be used with the syntax `__attribute__((strong_fptr("relax")))` to define a strong function pointer type for which the type rules are relaxed. In addition to the operations that are legal for the normal form of the attribute, the following operations are allowed:

- assignment of a relaxed strong function pointer value to an object that is not itself a strong function pointer type and does not contain any strong function pointer types

- assignment of a value that is not itself a strong function pointer type and does not contain any strong function pointer types to a relaxed strong function pointer object
- assignment of a strong function pointer value (with either form of type rules) to a relaxed strong function pointer object

The relaxed form of the attribute is useful in circumstances where you cannot change every value assigned to a strong function pointer object to use the same type. Because operations involving types that are not strong function pointers nullify the benefit of the attribute for **gstack**, we recommend that you use the standard form over the relaxed form whenever possible.

If you declare an object (variable, parameter, field of a struct, return type, etc.) to be of a `strong_fptr` type, you must use the same `strong_fptr` type in all translation units where the object is declared. Otherwise, the results are undefined.

`transparent_union`

Annotating the definition of a union type with `__attribute__((transparent_union))` marks that union type as *transparent*. In a function call where the formal parameter type is a transparent union type and the actual argument type is the same type as one of the members of that union, the compiler will silently generate code to accept the function call with the natural semantics. For example:

```
#include <stdio.h>
union __attribute__((transparent_union)) U {
    unsigned int ui;
    float f;
};
unsigned int bit_representation(union U u) {
    return u.ui;
}
int main() {
    printf("%x", bit_representation(3.14f));
    return 0;
}
```

`__attribute__((transparent_union))` is not supported in C++ mode, because it would interfere with function overload resolution.

`unused`

Annotating a function or variable definition with `__attribute__((unused))` tells the compiler that it is all right for this function or variable to have been defined but not referenced in the current translation unit; that is, it suppresses

the warnings usually given at end of scope for unreferenced entities. Related diagnostics are 177 and 826.

Annotating a type definition with `__attribute__((unused))` is equivalent to annotating every variable of that type with the same attribute.

`used`

Annotating a function or variable definition with `__attribute__((used))` tells the compiler that this function or variable might be used in a hidden or target-specific way; for example, it might be referenced from inline assembly code. As with `__attribute__((unused))`, it suppresses the warnings usually given at end of scope for unreferenced or unread variables. Related diagnostics are 177, 550, and 826.

`warn_unused_result`

Annotating a function declaration with `__attribute__((warn_unused_result))` causes the compiler to issue a warning whenever the function is called and the return value discarded. Related diagnostics are 1611 and 1767. Diagnostic 1768 is given whenever any function return value is discarded without an explicit cast to `void`.

`weak`

Annotating a function or variable declaration with `__attribute__((weak))` makes the declared symbol weak. For more information, see the documentation about `#pragma weak` *`foo`* in “Pragma Directives” on page 749.

Chapter 13

Green Hills C++

Contents

This chapter describes the Green Hills implementation of C++, including aspects of the language particular to our compilers.

The needs of C++ users vary widely, depending on a number of factors, including the type of target application and environment, the foreign libraries used, compatibility with other C++ compilers, and the trade-offs users make in regard to the C++ feature set and library support they require. To meet such a diverse set of needs, Green Hills Software supports a form of "scalable" C++. You can specify language and library levels to obtain everything from a small and efficient Embedded C++ compiler and library, to full ISO Standard C++.

Specifying a C++ Language Dialect

To specify a C++ dialect:

Set the **C/C++ Compiler→C++ Language Dialect** option to one of the available settings. For a list of driver options, see "C++ Language Dialect" on page 166.

Several dialects are available, depending on which OS you are using for your project:

- The *International Standard ISO/IEC 14882:2003* dialect (see "Standard C++" on page 692).
- The *Annotated C++ Reference Manual (ARM)* dialect (see "Standard C++ with ARM Extensions" on page 698).
- The GNU dialect (see "GNU C++" on page 699).
- Embedded C++ (see "Embedded C++" on page 695).
- Extended Embedded C++ (see "Extended Embedded C++" on page 697).

Note

The Green Hills Tools do not support mixing C++ language dialects. Use the same dialect when compiling and linking all C++ files in your project.

Specifying C++ Libraries

Depending on how you set **C/C++ Compiler→C++ Language Dialect** and **C/C++ Compiler→C++ Exception Handling**, the Green Hills Tools use the following default for **C/C++ Compiler→C++ Libraries**:

C++ Language Dialect	C++ Exception Handling	Default C++ Libraries
Standard C++ **--STD**, **--std**, **--arm**, **--g++**	On **--exceptions**	C++ with exceptions **--stdle**
	Off **--no_exceptions**	C++ without exceptions **--stdl**
Extended Embedded C++ **--ee**	On **--exceptions**	EEC++ with exceptions **--eele**
	Off **--no_exceptions**	EEC++ without exceptions **--eel**
Embedded C++ **--e**	On **--exceptions**	EC++ with exceptions **--ele**
	Off **--no_exceptions**	EC++ without exceptions **--el**

To link against a different set of Green Hills libraries, set the **C/C++ Compiler→C++ Libraries** option manually. You must specify an option that is less full-featured than the default (and therefore lower in the table). The Green Hills Tools do not support mixing C++ libraries. Use the same library when compiling and linking all C++ files in your project.

For example, if you want to write code in C++ that uses exceptions while restricting your use of library modules to those offered in EEC++, use the following options:

```
--std --exceptions --eele
```

The **C/C++ Compiler→C++ Libraries** option also sets the default `#include` paths so that the tools use the correct header files for the selected libraries.

Note

We recommend that you do not mix code that handles exceptions with libraries that do not handle exceptions, or vice versa, as you may get unexpected results when running your program.

If you do not want to link against any Green Hills standard libraries, see "Link in Standard Libraries" on page 287.

Standard C++

This dialect is defined by the *International Standard ISO/IEC 14882:2003* (except as noted in this chapter). It also includes additional features and changes developed after the standard. To specify use of this dialect:

Set the **C/C++ Compiler→C++ Language Dialect** option to either:

- **Standard C++ (Violations give Errors)** (**--STD**), or
- **Standard C++ (Violations give Warnings)** (**--std**) [default]

Some of the features in the standard are disabled by default. In order to obtain the exact dialect specified by the standard, you need to set these additional options as follows:

- Set the **C/C++ Compiler→C++ Exception Handling** option to **On** (**--exceptions**).
- Set the **C/C++ Compiler→C++→Templates→Recognition of Exported Templates** option to **On** (**--export**).
- Set the **Advanced→C/C++ Compiler Options→Advanced C++ Options→Distinct C and C++ Functions** option to **On** (**--c_and_cpp_functions_are_distinct**).
- Set the **C/C++ Compiler→C++→Keyword Support→Instantiate Extern Inline** option to **On** (**--instantiate_extern_inline**).
- Set the **C/C++ Compiler→C++→Templates→Implicit Source File Inclusion** option to **Off** (**--no_implicit_include**). This setting is implied by **-export**.

By default, the **--STD** option enables **C/C++ Compiler→C++→Templates→Dependent Name Processing** (**--dep_name**) and

Advanced→C/C++ Compiler Options→Advanced C++ Options→Distinct C and C++ Functions (**--c_and_cpp_functions_are_distinct**). For more information about **--dep_name**, see "Namespace Support" on page 712.

Extensions Accepted in Normal C++ Mode

The following extensions are accepted in all modes (except when strict ISO C violations are diagnosed as errors):

- A `friend` declaration for a class may omit the `class` keyword:

```
class B;
class A {
  friend B;       // Should be "friend class B"
};
```

- Constants of scalar type may be defined within classes:

```
class A {
  const int size = 10;
  int a[size];
};
```

- In the declaration of a class member, a qualified name may be used:

```
struct A {
  int A::f();      // Should be "int f()"
};
```

- "Anonymous classes" and "anonymous structures" are allowed as long as they have no C++ features (e.g., no static data members or member functions and no nonpublic members) and have no nested types other than other anonymous classes, structures, or unions. For example:

```
struct A {
  struct {
    int i, j;
  };         // Legal -- references to A::i and A::j
             // are allowed
};
```

- If you use the **--restrict** option, the type qualifier `restrict` is allowed for reference and pointer-to-member types and for nonstatic member functions. For more information about this option, see “restrict Keyword Support” on page 205.
- Implicit type conversion between a pointer to an `extern "C"` function and a pointer to an `extern "C++"` function is permitted. For example:

```
extern "C" void f();  // f's type has
                      // extern "C" linkage
void (*pf)() = &f;    // pf points to an
                      // extern "C++" function
                      // error unless implicit
                      // conversion is allowed
```

  For more information, see “Distinct C and C++ Functions” on page 285.
- In **--arm** mode, a `?` operator whose second and third operands are string literals or wide string literals can be implicitly converted to `char *` or `wchar_t *`, respectively. In C++, string literals are `const`. There is a deprecated implicit conversion that allows conversion of a string literal to `char *`, dropping to `const`. That conversion, however, applies only to simple string literals. Allowing it for the result of a `?` operation is an extension.

```
char *p = x ? "abc" : "def";
```

Note

For more information about Standard C++, we recommend Stroustrup's *The C++ Programming Language*.

Embedded C++

The Embedded C++ dialect (EC++) is intended as a stable, simple, and efficient version of Standard C++, intended to produce an open, worldwide standard. EC++ is not a new language specification that competes with existing Standard C++; it is a pure subset for the practical user of C++. The EC++ library is smaller and more efficient than the Standard C++ library. For more information about EC++, visit the Embedded C++ Web site [http://www.caravan.net/ec2plus/]. For a feature comparison with Standard C++, see "Features Supported by Each C++ Dialect" on page 697.

To specify use of this dialect:

Set the **C/C++ Compiler→C++ Language Dialect** option to **Embedded C++ (--e)**.

Differences Between Standard C++ and Embedded C++

In addition to the features and omissions mentioned in "Features Supported by Each C++ Dialect" on page 697, the following features behave differently in Embedded C++ than in Standard C++.

std::string

In C++, `string` is a typedef for:

```
basic_string<char, char_traits<char>, allocator<char> >
```

In EC++, none of these template classes exist, and `string` is provided as a non-template class. The EC++ library does not provide any wide character version of string (there is no `wstring`).

Complex Numbers

In C++, the `<complex>` header defines a template class `complex<T>` with specializations for `float`, `double`, and `long double`. In EC++, this template

class does not exist, and the <complex> header defines two non-template classes float_complex and double_complex, with all the same member functions as the standard complex<float> and complex<double>, respectively. The EC++ library does not provide any equivalent to complex<long double>.

Streams and Buffers

In C++, the standard library provides a large number of "stream" and "buffer" classes — filebuf, streambuf, stringbuf, istream, ostream, ifstream, and ofstream — each of which is implemented as a specialization of an underlying template class named basic_*class*. In EC++, none of these underlying template classes exist, but the EC++ library provides non-template versions of the stream classes with the same interfaces as the standard C++ versions. The EC++ library does not provide any wide character streams or buffers (there is no wistream, wstringbuf, etc.).

The standard C++ classes iostream and strstream require multiple inheritance, and therefore are not provided by EC++.

The only standard iostream objects provided by EC++ and EEC++ are cin and cout. The cerr and clog objects are omitted for simplicity, and the wide iostream objects wcin, wcout, wcerr, and wclog are omitted along with all of the wide stream and buffer types.

Extended Embedded C++

Extended Embedded C++ is a hybrid of Standard C++ and Embedded C++, created in a joint effort by Green Hills Software and P. J. Plauger. It is often referred to as EEC++ or ESTL (Embedded C++ with the Standard Template Library). It is not an official standard. For a feature comparison with Standard C++ and EC++, see "Features Supported by Each C++ Dialect" on page 697.

To specify use of this dialect:

Set the **C/C++ Compiler→C++ Language Dialect** option to **Extended Embedded C++** (**--ee**).

Features Supported by Each C++ Dialect

The following table compares the feature set of Standard C++, Extended Embedded C++ (EEC++), and Embedded C++ (EC++). Features that are not listed in this table can be individually controlled through Builder and driver options. For more information about features marked with EC++, see "Differences Between Standard C++ and Embedded C++" on page 695.

Feature	Standard	EEC++	EC++
Exception support	Optional	Optional	Optional
Complex numbers	C++	EC++	EC++
Streams and buffers	C++	EC++	EC++
`string`	C++	EC++	EC++
`wchar_t`	Yes	Yes	Yes
`cin`, `cout`	Yes	Yes	Yes
Template support	Yes	Yes	No
Standard container classes	Yes	Yes	No
Namespace support	Yes	Yes	No
`std::` namespace	Yes	Yes	No
`bitset<>`	Yes	Yes	No

Feature	Standard	EEC++	EC++
`<algorithm>`, `<functional>`, `<memory>`	Yes	Yes	No
`mutable` keyword	Yes	Yes	No
`static_cast<>`, `const_cast<>`, `reinterpret_cast<>`	Yes	Yes	No
Multiple inheritance	Yes	No	No
`dynamic_cast<>`	Yes	No	No
`<locale>`	Yes	No	No
`<limits>`	Yes	No	No
`wcin`, `wcout`, etc.	Yes	No	No
`cerr`, `clog`	Yes	No	No
`iostream`, `strstream`	Yes	No	No
`<typeinfo>`	Yes	No	No

Standard C++ with ARM Extensions

This dialect is ARM-compliant C++ as defined by *The Annotated C++ Reference Manual* (ARM) by Ellis and Stroustrup (Addison-Wesley, 1990), including templates, exceptions, and the anachronisms listed in Chapter 18 of that book. This is essentially the same language defined by the language reference for Cfront version 3.0*x*, with the addition of exceptions.

To specify use of this dialect:

Set the **C/C++ Compiler→C++ Language Dialect** option to **Standard C++ with ARM Extensions (--arm)**.

GNU C++

This dialect closely emulates the C++ language supported by the GNU compiler.

To specify use of this dialect:

Set the **C/C++ Compiler→C++ Language Dialect** option to **GNU C++** (**--g++**).

GNU C++ Extensions

This dialect enables the extensions listed below, together with those in "GNU C/C++ Extensions" on page 633.

- If an explicit instantiation is preceded by the keyword `extern`, no (explicit or implicit) instantiation is created for the indicated specialization.
- A special keyword `__null` expands to the same constant as the literal "`0`", but is expected to be used as a null pointer constant.
- When dependent name processing is disabled, names from dependent base classes are ignored only if another name would be found by the lookup.

```
const int n = 0;
template <class T> struct B {
  static const int m = 1; static const int n = 2;
};
template <class T> struct D : B<T> {
  int f() { return m + n; } // B::m + ::n in g++ mode
};
```

- When doing name lookup in a base class, the injected class name of a template class is ignored.

```
namespace N {
  template <class T> struct A {};
}
struct A {
  int i;
};
```

```
struct B : N::A<int> {
  B() { A x; x.i = 1; } // g++ uses ::A, not N::A
};
```

- The injected class name is found in certain contexts in which the constructor should be found instead.

```
struct A {
  A(int) {};
};
A::A a(1);
```

- A difference in calling convention is ignored when redeclaring a typedef, even when **--c_and_cpp_functions_are_distinct** is specified.

```
typedef void F();
extern "C" {
  typedef void F(); // Accepted in GNU C++ mode (error otherwise)
}
```

- The macro `__GNUG__` is defined identically to `__GNUC__` (i.e., the major version number of the GNU compiler version that is being emulated).
- **GNU 3.3 ONLY** A friend declaration may refer to a member typedef.

```
class A {
  class B {};
  typedef B my_b;
  friend class my_b;
};
```

- An inherited type name can be used in a class definition and later redeclared as a typedef.

```
struct A { typedef int I; };
struct B : A {
  typedef I J;      // Refers to A::I
  typedef double I; // Accepted in g++ mode
};                  // (introduces B::I)
```

- In a catch clause, an entity may be declared with the same name as the handler parameter.

```
try { }
catch(int e) {
  char e;
}
```

- A template argument list may appear following a constructor name in a constructor definition that appears outside of the class definition:

```
template <class T> struct A {
  A();
};
template <class T> A<T>::A<T>(){}
```

- **GNU 3.3 ONLY** A constructor need not provide an initializer for every nonstatic const data member (but a warning is still issued if such an initializer is missing).

```
struct S {
  int const ic;
  S() {} // Warning only in GNU C++ mode (error otherwise)
};
```

- Exception specifications are ignored on function definitions when support for exception handling is disabled (normally, they are only ignored on function declarations that are not definitions).
- A friend declaration in a class template may refer to an undeclared template.

```
template <class T> struct A {
    friend void f<>(A<T>);
};
```

- A function template default argument may be redeclared. A warning is issued and the default from the initial declaration is used.

```
template<class T> void f(int i = 1);
template<class T> void f(int i = 2) {}
int main() {
  f<void>();
}
```

- A definition of a member function of a class template that appears outside of the class may specify a default argument.

```
template <class T> struct A { void f(T); };
template <class T> void A<T>::f(T value = T()) {}
```

Template Instantiation

The C++ language includes the concept of templates. A template is a description of a class or function that is a model for a family of related classes or functions. For example, you can write a template for a `Stack` class, and then use a stack of integers, a stack of floats, and a stack of some user-defined type. In the source, these might be written `Stack<int>`, `Stack<float>`, and `Stack<X>`. From a single source description of the template for a stack, the compiler can create instantiations of the template for each of the types required.

The instantiation of a class template is always done as soon as it is needed in a compilation. However, the instantiations of template functions, member functions of template classes, and static data members of template classes (hereafter referred to as *template entities*) are not necessarily done immediately, for several reasons:

- You would like to end up with only one copy of each instantiated entity across all the object files that make up a program. (This of course applies to entities with external linkage.)
- The language allows you to write a specialization of a template entity, that is, a specific version to be used in place of a version generated from the template for a specific data type. (You could, for example, write a version of `Stack<int>`, or of just `Stack<int>::push`, that replaces the template-generated version; often, such a specialization provides a more efficient representation for a particular data type.) While compiling a reference to a template entity, the compiler cannot know if a specialization for that entity will be provided in another compilation. Thus, it cannot do the instantiation automatically in any source file that references it.
- The language also dictates that template functions that are not referenced should not be compiled, and that, in fact, such functions might contain semantic errors that would prevent them from being compiled. Therefore, a reference to a template class should not automatically instantiate all the member functions of that class.

It should be noted that certain template entities are always instantiated when used (e.g., inline functions).

The Green Hills C++ compiler provides two methods of template instantiation, which are described in the following.

Prelinker Template Instantiation

Prelinker instantiation, selected with the **--no_link_once_templates** option, is the default mode of template instantiation. To explicitly enable it:

Set the **C/C++ Compiler→C++→Templates→Link-Once Template Instantiation** option to **Off** (**--no_link_once_templates**). For more information about this option, see "Templates" on page 210.

The Green Hills C++ prelinker instantiation method works as follows:

1. The first time the source files of a program are compiled, no template entities are instantiated. However, template information files are generated and contain information about entities that could have been instantiated in each compilation. These template information files have a **.ti** suffix.
2. When the object files are linked together, a program called the *prelinker* is run. It examines the object files, looking for references and definitions of template entities, and for the added information about entities that could be instantiated.
3. If the prelinker finds a reference to a template entity for which there is no definition anywhere in the set of object files, it looks for a file that indicates that it could instantiate that template entity. Upon finding such a file, the prelinker assigns the instantiation to it. The set of instantiations assigned to a given file is recorded in an associated instantiation request file (with a **.ii** suffix).
4. The prelinker then executes the compiler again to recompile each file for which the **.ii** file was changed. The original compilation options (saved in the **.ti** file) are used for recompilation.
5. When the compiler compiles a file, it reads the **.ii** file for that file and obeys the instantiation requests therein. It produces a new object file containing the requested template entities (and everything else already in the object file). The

compiler also receives a definition list file, which lists all the instantiations for which definitions already exist in the set of object files. If during compilation the compiler has the opportunity to instantiate a referenced entity that is not on that list, it performs the instantiation. It passes back to the prelinker (in the definition list file) a list of instantiations that it has "adopted" in this way, so the prelinker can assign them to a file. This adoption process allows rapid instantiation and assignment of instantiations referenced from new instantiations, and reduces the need to recompile a given file more than once during the prelinking process.

6. The prelinker repeats steps 3-5 until there are no more instantiations to be adjusted.
7. The object files are linked together.

After the program has been linked correctly, the **.ii** files contain a complete set of instantiation assignments. From then on, whenever source files are recompiled, the compiler will consult the **.ii** files and do the indicated instantiations as it does the normal compilations. That means that, except in cases where the set of required instantiations changes, the prelink step from then on will find that all the necessary instantiations are present in the object files and no adjustments will need to be made to the instantiation assignments. This is true even if the entire program is recompiled.

If the programmer provides a specialization of a template entity somewhere in the program, the specialization will be seen as a definition by the prelinker. Since that definition satisfies whatever references there might be to that entity, the prelinker will see no need to request an instantiation of the entity. If the programmer adds a specialization to a program that has previously been compiled, the prelinker will notice that too and remove the assignment of the instantiation from the proper **.ii** file.

The **.ii** files should not, in general, require any manual intervention. There are two exceptions.

1. If the following are true:
    - A definition is changed in such a way that some instantiation no longer compiles (it gets errors), and
    - A specialization is added in another file, and
    - The first file is being recompiled before the specialization file and is getting errors,

You must delete the **.ii** file for the file getting the errors to allow the prelinker to regenerate it.

2. If you use multiple major versions of the tools to build your project (for example, MULTI 4 and MULTI 5), we recommend that you use multiple build directories. If you do not use multiple build directories, you might get errors when building your project in one release when **.ti** files reference configuration options that only exist in the other release. If you get these errors, remove the appropriate **.ti** and **.ii** files.

If the prelinker changes an instantiation assignment, it will issue a message such as:

```
C++ prelinker: A<int>::f() assigned to file test.o
```

Instantiations are normally generated as part of the object file of the translation unit in which the instantiations are performed. But when *one instantiation per object* is used, each instantiation is placed in its own object file. This mode is useful when building libraries that need to include copies of the instances referenced from the library. If each instance is not placed in its own object file, it may be impossible to link the library with another library containing some of the same instances.

To enable one instantiation per object:

Set the **C/C++ Compiler→C++→Templates→One Instantiation Per Object** option to **On** (**--one_instantiation_per_object**). For more information about this option, see “Templates” on page 210.

Note

If you use one instantiation per object and you have a template that calls a static function, you might not get debug information for that function.

Link-Once Template Instantiation

An alternative method of template instantiation is known as link-once template instantiation. In this method, non-export templates that are used in a given source file are always instantiated during compilation. Because a template can be used by multiple source modules, some convention must be used to ensure that the definitions of such templates in multiple object files does not result in multiply defined symbols

at link time. The templates are instantiated into specially named link-once sections as if they had been declared with `__linkonce` (see "ANSI C Extensions" on page 623).

To enable link-once template instantiation:

Set the **C/C++ Compiler→C++→Templates→Link-Once Template Instantiation** option to **On** (**--link_once_templates**). For more information about this option, see "Templates" on page 210.

Because link-once template instantiation repeatedly compiles templates, link-once template instantiation may result in longer compile times, although the initial build of a project may be faster since prelink-time template instantiation will rebuild objects during the initial prelink phase. Link-once template instantiation is often used in build environments such as **clearmake**, where some aspect of prelink-time template instantiation is undesirable, although the prelinker may still need to instantiate export templates.

Instantiation Pragma Directives

Instantiation `#pragma` directives can be used to control the instantiation of specific template entities or sets of template entities. This is sometimes used as an alternative to Link-Once Template Instantiation in environments where minimizing the amount of prelinking is desirable. There are three instantiation `#pragma` directives:

- `#pragma instantiate` *`fn`* — Instructs the compiler to instantiate *fn*.
- `#pragma can_instantiate` *`fn`* — Instructs the compiler to consider *fn* for instantiation. This `#pragma` is often used in conjunction with automatic instantiation to indicate potential sites for instantiation if the template entity turns out to be required.
- `#pragma do_not_instantiate` *`fn`* — Instructs the compiler to not instantiate *fn*. This `#pragma` is often used to suppress the instantiation of an entity for which a specific definition will be supplied.

Each of the above instantiation `#pragma` directives takes an argument, *fn*, which may be one of the following:

- A template class name (e.g., `A<int>`)

- A template class declaration (e.g., `class A<int>`)
- A member function name (e.g., `A<int>::f`)
- A static data member name (e.g., `A<int>::i`)
- A static data declaration (e.g., `int A<int>::i`)
- A member function declaration (e.g., `void A<int>::f(int, char)`)
- A template function declaration (e.g., `char* f(int, float)`)

A `#pragma` directive in which the argument is a template class name is equivalent to repeating the directive for each member function and static data member declared in the class. When instantiating an entire class a given member function or static data member may be excluded using the `#pragma do_not_instantiate` directive. For example:

```
#pragma instantiate A<int>
#pragma do_not_instantiate A<int>::f
```

The template definition of a template entity must be present in the compilation for an instantiation to occur. If an instantiation is explicitly requested by use of the `#pragma instantiate` directive, and no template definition is available, or a specific definition is provided, an error is issued.

```
template <class T> void f1(T);  // No body provided
template <class T> void g1(T);  // No body provided
void f1(int) {}  // Specific definition
void main()
{
  int i;
  double d;
  f1(i);
  f1(d);
  g1(i);
  g1(d);
}
#pragma instantiate void f1(int)  // error - specific
                                  // definition
#pragma instantiate void g1(int)  // error - no body
                                  // provided
```

`f1(double)` and `g1(double)` will not be instantiated (because no bodies were supplied) but no errors will be produced during the compilation (if no bodies are supplied at link time, a linker error will be produced).

A member function name (for example, `A<int>::f`) can only be used as a `#pragma` argument if it refers to a single user-defined member function (that is, a function that is not overloaded). Compiler-generated functions are not considered, so a name may refer to a user-defined constructor even if a compiler-generated copy constructor of the same name exists. Overloaded member functions can be instantiated by providing the complete member function declaration. For example:

```
#pragma instantiate char* A<int>::f(int, char*)
```

The argument to an instantiation directive may not be a compiler-generated function, an inline function, or a pure virtual function.

Implicit Inclusion

Implicit inclusion permits the compiler to assume that if it needs a definition to instantiate a template entity declared in a **.h** file it can implicitly include the corresponding **.cc** file to get the source code for the definition. To enable this feature:

Set the **C/C++ Compiler→C++→Templates→Implicit Source File Inclusion** option to **On** (**--implicit_include**).

For example, if a template entity `ABC::f` is declared in file **xyz.h**, and an instantiation of `ABC::f` is required in a compilation but no definition of `ABC::f` appears in the source code processed by the compilation, the compiler searches for an appropriately suffixed C++ source file whose base name is **xyz**; if the file is found, the compiler processes it as if the file were included at the end of the main source file.

To find the template definition file for a given template entity, the compiler needs to know the full pathname of the file in which the template was declared and whether the file was included using the system include syntax (e.g., `#include <file.h>`). This information is not available for preprocessed source code containing `#line` directives. Consequently, the compiler will not attempt implicit inclusion for source code containing `#line` directives.

The following suffixes are searched for: `.c`, `.C`, `.cpp`, `.CPP`, `.cxx`, `.CXX`, and `.cc`.

Care should be taken when writing files that may be implicitly included. For instance, if the file **xyz.cc** is implicitly included since it contains a definition of a template declared in **xyz.h**, it is possible that more than one module will include it. In fact, most likely any module that includes **xyz.h** will end up implicitly including **xyz.cc**. In addition, if **xyz.cc** defines a global symbol (a non-template function or a variable), every module which (implicitly) includes it will get that symbol definition. This can lead to multiply defined symbol errors from the linker. This can be averted by having only template definitions in files that can be implicitly included. An implicitly included header file should be organized much the same way as any other header file.

Since a file is automatically pulled in when the implicit inclusion is on, the compiler driver or the builder do not need to separately compile it; just like a regular header file does not need to be separately compiled.

If a source file is unexpectedly compiled via implicit inclusion, you may see strange errors during compilation or linking. We recommend that you do not enable implicit inclusion when you start new projects written in standard C++.

Exported Templates

Exported templates are templates declared with the keyword `export`, which is not recognized by default. To enable this feature:

Set the **C/C++ Compiler→C++→Templates→Recognition of Exported Templates** option to **On** (**--export**).

Note

This option requires and will automatically enable **C/C++ Compiler→C++→Templates→Dependent Name Processing** (**--dep_name**).

Exported templates and implicit inclusion (see “Implicit Inclusion” on page 708) are mutually exclusive. Enabling **C/C++ Compiler→C++→Templates→Recognition of Exported Templates**

(**--export**) will automatically disable **C/C++ Compiler→C++→Templates→Implicit Source File Inclusion** (**--no_implicit_include**)

Exporting a class template is equivalent to exporting each of its static data members and each of its non-inline member functions. An exported template is special because its definition does not need to be present in a translation unit that uses that template. In other words, the definition of an exported (non-class) template does not need to be explicitly or implicitly included in a translation unit that instantiates that template. For example, the following is a valid C++ program consisting of two separate translation units:

```
// File 1:
#include <stdio.h>
static void trace() { printf("File 1\n"); }

export template <class T> T const & min(T const &, T const &);
int main() {
  trace();
  return min(2, 3);
}

// File 2:
#include <stdio.h>
static void trace() { printf("File 2\n"); }

export template <class T> T const & min(T const &a,
                                        T const &b) {
  trace();
  return a < b ? a : b;
}
```

Note that these two files are separate translation units: one is not included in the other. That allows the two functions `trace()` to coexist (with internal linkage).

Note

We recommend that you do not use `export` templates; they have been removed in the C++11 standard.

Multiple and Virtual Inheritance

By default, when involving multiple and/or virtual inheritance, the Green Hills C++ compiler has a limitation on the maximum byte offset of a base class within a derived class. If the offset is larger than the maximum value that fits in the `unsigned short` type, virtual function calls and RTTI might not work properly. For example:

```
class BIG_BASE {
public:
        char c[70000];
        virtual void foo();
};


class TWO_BASE {
public:
        int i;
        virtual void bar();
};


// offset of TWO_BASE is sizeof(BIG_BASE), which doesn't fit
// into 'unsigned short' on most architectures - diagnostic
// is given
class DERIVED : public BIG_BASE, public TWO_BASE {
public:
        // ...
};
```

In this type of situation, an error message is generated.

Where only single class inheritance is involved, the offset of a base class is always 0, so this limitation does not apply.

To remove this limitation, you can pass the **--large_vtbl_offsets**. For more information, see “Virtual Function Table Offset Size” on page 209.

Namespace Support

Namespaces are enabled by default. To disable this feature:

Set the **C/C++ Compiler→C++→Namespaces→Namespace Support** option to **Off** (**--no_namespaces**).

When looking up names in a template instantiation, some names must be found in the context of the template definition, while others can also be found in the context of the template instantiation.

The Green Hills C++ compiler implements two different instantiation lookup algorithms:

- The algorithm mandated by the standard (“Dependent Name Lookup” on page 712).
- The algorithm that existed before the dependent name lookup was implemented (“Lookup Using the Referencing Context” on page 713).

To enable dependent name lookup:

Set the **C/C++ Compiler→C++→Templates→Dependent Name Processing** option to **On** (**--dep_name**).

Dependent Name Lookup

When using dependent name lookup, the Green Hills C++ compiler implements the instantiation name lookup rules that are specified in the standard. This processing requires that non-class prototype instantiations are performed, which requires that the code be written using the **typename** and **template** keywords that are required by the standard.

Lookup Using the Referencing Context

When you are not using dependent name lookup, the compiler uses a name lookup algorithm that approximates the two-phase lookup rule of the standard. This is done in such a way that is more compatible with existing code and legacy compilers.

When a name is looked up as part of a template instantiation but is not found in the local context of the instantiation, the name is looked up in a synthesized instantiation context that includes both names from the context of the template definition and names from the context of the instantiation.

For example:

```
namespace N {
   int g(int);
   int x = 0;
   template <class T> struct A {
      T f(T t) {return g(t);}
      T f() {return x;}
   };
}
namespace M {
   int x = 99;
   double g(double);
   N::A<int> ai;
   int i = ai.f(0);     // N::A<int>::f(int) calls
                        // N::g(int)
   int i2 = ai.f();     // N::A<int>::f() returns 0
                        //   (= N::x)
   N::A<double> ad;
   double d = ad.f(0); // N::A<double>::f(double)
                        // calls M::g(double)
   double d2 = ad.f(); // N::A<double>::f() also
                        // returns 0 (= N::x)
 }
```

The lookup of names in template instantiations does not conform to the rules in the standard in the following respects:

- Although only names from the template definition context are considered for names that are not functions, the lookup is not limited to those names visible at the point at which the template was defined.

- Functions from the context in which the template was referenced are considered for all function calls in the template. Functions from the referencing context should only be visible for “dependent” functions calls.

Argument Dependent Lookup

By default, the Green Hills C++ compiler performs argument-dependent (Koenig) lookup, as specified by the standard. Functions made visible by using argument-dependent lookup overload with those made visible by normal lookup.

Setting the **--g++** option can affect argument-dependent lookup in some cases. For instance, normally Koenig lookup is suppressed when normal lookup of an unqualified name finds a block-scope function declaration that is not a using-declaration, but in GNU mode Koenig lookup will continue despite such a declaration.

```
namespace M {
   struct A {};
   void g(A, double);
   void h(A, double);
   A operator+(A, double);
}
void g(M::A, int);
M::A operator+(M::A, int);
void f() {
   void h(M::A, int);
   M::A a1;
   g(a1, 1.0); // calls M::g(M::A, double)


   a1 + 1.0;   // calls M::operator+(M::A, double)
   h(a1, 1.0); // calls h(M::A, int) unless --g++ is set,
               // in which case M::h(M::A, double)
               // is called
}
```

Linkage

C++ accepts the `extern "`*`language`*`"` storage class specifier in order to achieve linkage between C++ and C. The full syntax is as follows:

```
extern "language" {
  declarations
}
```

or

```
extern "language" declaration;
```

where *language* may be `C` or `C++`.

Language	Effect on Resulting Code
`C`	C++ does not alter the procedure name as it usually would when confronted with an overloaded function.
`C++`	Uses C++ naming rules. Function names are always mangled according to C++ linkage specifications. This is the default linkage.

Note that the `extern "language"` directive only affects the external names of functions so that the compiler will apply the appropriate function naming rules. This directive does not modify the type or number of arguments of a function, or its return type. Normal C++ type checking rules are not altered by this directive.

For more information about using C++ with C, see Chapter 17, “Mixing Languages and Writing Portable Code” on page 827.

Post Processing in C++

In C or C++, *global objects* (or non-local static objects) are those objects which are declared outside of the scope of any function and are available throughout the entire program. C++ objects which are instances of a class type have mechanisms for automatic construction/initialization and destruction/cleanup through the use of *constructors* and *destructors*. Constructors are functions which are automatically called by the compiler when an object is created. Destructors are called when an object is deleted. This implies some implementation-specific behavior for global objects which may vary from C++ system to C++ system.

Global objects, such as those in libraries (e.g. `cin`, `cout`, and `cerr`) must be constructed and initialized for the entire program. This means that the constructor functions must be called as soon as the program begins. The compiler has no knowledge of external global objects contained in other modules or libraries except for an `extern` declaration. This is not enough information for the compiler to be

able to properly ensure that these calls are performed. The linker resolves the global constructor and destructor information. The VxWorks environment is unique in that the module load/unload functions invoke the global constructors/destructors, or else the user executes them manually.

When using the **--munch** option, after an executable has been produced, the Green Hills C++ compiler driver calls an **nm** utility (such as **gnm**) to find all global symbols. The output of the **nm** utility is sent to the post-link program **cxxmunch**. **cxxmunch** searches for all global constructor and destructor calls and generates a C module that will execute these calls appropriately at program startup and exit. The driver then invokes the compiler and assembler to produce another object module. Following that, the linker is invoked to relink the new constructor/destructor object with the original object modules and libraries. This produces the fully processed C++ executable, which has all of the appropriate constructor and destructor calls. The driver makes all of this processing completely transparent. However, if you choose not to use the driver provided by Green Hills, you are responsible for calling the post-link program after producing an executable; otherwise the program may not run correctly.

The C++ decode Utility

The **decode** utility is provided with Green Hills C++. Its syntax is as follows:

decode [-u]

This utility is a C++ name "demangler". It reads the input from `stdin` and writes output to `stdout`. Anything that looks like mangled names in the input are demangled. Everything else is passed through unchanged. This makes the program suitable, for example, as a filter for the output of a linker; error messages in general are passed through unaltered, but mangled names are demangled. For example:

```
ld: Undefined symbol
    f__Fif
```

is changed to:

```
ld: Undefined symbol
    f(int, float)
```

The decode utility takes only a single option: **-u**. When used, **-u** specifies that external names have an added leading underscore.

C++ Implementation-Defined Features

- **1.3.2: Diagnostic message** — For a complete list of the C++ compiler diagnostic messages, see the *MULTI: C and C++ Compiler Error Messages* book (not available in print).
- **1.3.5: Implementation-defined behavior** — Documentation is provided in this section.
- **1.4: Implementation compliance** — The standard Green Hills distribution is built around a hosted implementation with the standard set of available libraries.
- **1.7: The C++ memory model** — The size of a byte is 8 bits.
- **1.9: Program execution** — The minimum requirements are all fulfilled.
- **2.1: Phases of translation** — Mapping of file characters to the basic source character set is performed via identity mapping. Non-empty sequences of white-space characters (other than newline characters) are retained during preprocessing. Sources of translation units containing definitions of templates that are being instantiated are required to be available.
- **2.2: Character sets** — ASCII encoding is used for all characters.
- **2.8: Header names** — See “Instructing the Compiler to Search for Your Headers” on page 70.
- **2.13.2: Character literals** — The **C/C++ Compiler→Special Tokens→Host and Target Character Encoding** option enables recognition of multi-byte character Kanji sequences. When this option is disabled, or when unrecognized sequences are encountered, up to four chars are generated, initializing an `int` value, the least significant byte of which is initialized to the rightmost c-char of the multi-byte character literal. If there are less than four c-chars, the most significant bytes will be 0. Character literals outside of the implementation-defined range are truncated to the [0,255] range. Universal characters are truncated in the same way as `\xhhh` characters.
- **2.13.3: Floating literals** — Same as the C compiler (see “C Implementation-Defined Features” on page 660).

- **2.13.4: String literals** — Duplicate string literals are merged by default. Enable the **C/C++ Compiler→C/C++ Data Allocation→Uniquely Allocate All Strings** option to create a unique instance of every string.
- **3.6.1: Main function** — `main()` is required for stand-alone projects, and all operating systems except for VxWorks. The following additional `main()` definitions are supported:
  - `int main(int argc, char *argv[], char *env[])`

    (where `*env[]` is a null-terminated array of environment variables.)
  - A return type of `void` is also permitted with a warning.

  The linkage of `main()` is external.
- **3.6.2: Initializations of non-local objects** — Dynamic initialization of objects of namespace scope is performed by `_main()`, except for VxWorks projects which use the `_ctors` array and for Native Linux targets which use the `.init_array` section. All static constructors within one compilation unit are executed in the order in which their definition appears within the translation unit. Constructors in different compilation units are executed in link-line order.
- **3.9: Types** — For data type sizes and alignments, see "V850 and RH850 Characteristics" on page 26.
- **3.9.1: Fundamental types** — `char` and `signed char` hold equivalent values by default. For more information, see the **C/C++ Compiler→Data Types→Signedness of Char Type** option. `float` and `double` are represented in the IEEE format.
- **3.9.2: Compound types** — Pointers are represented as plain memory addresses in the natural way for the target.
- **4.7: Integral conversions** — Unrepresentable values are truncated to a value that fits into the destination type using modulo arithmetic.
- **4.8: Floating-point conversions** — Same as the C compiler (see "C Implementation-Defined Features" on page 660).
- **4.9: Floating-integral conversions** — Same as the C compiler (see "C Implementation-Defined Features" on page 660).
- **5.2.8: Type identification** — `typeid` expression results are only of static type `std::type_info`. No derived classes are provided.

- **5.2.10: Reinterpret_cast** — Mapping remains the same. Mapping of pointers to addresses and vice versa uses `int` or larger type.
- **5.3.3: Sizeof** — For information about the sizes of fundamental types, see "V850 and RH850 Characteristics" on page 26.
- **5.6: Multiplicative operators** — Remainder from the division of a negative and non-negative operand is the same sign as the first operand.
- **5.7: Additive operators** — `ptrdiff_t` is a typedef for `int` on 32-bit targets, or `long long` on 64-bit targets.
- **5.8: Shift operators** — The result of `E1 >> E2` where `E1` has a signed type and a negative value is negative.
- **7.1.5.2: Simple type specifiers** — Type `char` is `signed` by default. Bitfields are `signed` by default.
- **7.2: Enumeration declarations** — Enumerations have a type of `int`, unless the **C/C++ Compiler→Data Types→Use Smallest Type Possible for Enum** option is enabled.
- **7.4: The asm declaration** — `asm` declarations are unmodified by the compiler. For more information, see Chapter 18, "Enhanced asm Macro Facility for C and C++" on page 847.
- **7.5: Linkage specifications** — Routines declared with `extern "C"` will not have name mangling performed on them. The default mode, `extern "C++"`, performs name mangling. No other language linkages are supported. Mangling of function names, type names, and static data member names is performed by the C++ compiler.
- **8.5.3: References** — In the circumstances described, a temporary is generated, `rvalue` is copied into it, and the reference is bound to the temporary.
- **9.6: Bit-fields** — A bitfield is allocated at the current offset, unless it crosses an offset which is a multiple of the size of the base type, in which case it is padded up to the first multiple of the size of the base type. Plain bitfields are unsigned unless **C/C++ Compiler→Data Types→Signedness of Bitfields** is set to **Signed**.
- **12.2: Temporary objects** — Are generated for `f(X(2))`, but the result will be stored directly into `b`.
- **14: Templates** — Only C and C++ linkages are supported.
- **14.7.1: Implicit instantiation** — Recursive instantiations are limited to 64.

- **15.5.1: The terminate() function** — The stack is unwound before `terminate()` is called.
- **15.5.2: The unexpected() function** — Unless reset by user, will call `terminate()`.
- **16.1: Conditional inclusion** — Numeric values for character literals in `#if` and `#elif` controlling expressions is identical to `char` literals in regular expressions.
- **16.2: Source file inclusion** — See "Instructing the Compiler to Search for Your Headers" on page 70.
  - `#include <`*`h-char-sequence`*`>` — searches the system header directories. All identifiers in *h-char-sequence* will be resolved.
  - `#include "`*`q-char-sequence`*`"` — searches the current directory, and then all include directories. Sequences in *q-char-sequence* will be treated as a string literal, and will not be resolved.
  - Mapping between the sequence and the external source filename — is performed through the identity function.
  - `#include` preprocessor directives can be nested to arbitrary depth.
- **16.6: Pragma directive** — See "Pragma Directives" on page 749.
- **16.8: Predefined macro names** — For a list of the required preprocessor macros and their values, see "Macro Names Required by ANSI C and C++" on page 734.
- **17.4.4.5: Reentrancy** — All subroutines are reentrant provided that the target system has been configured with the appropriate lock routines in subdirectory **src/libsys**.
- **17.4.4.8: Restrictions on exception handling** — See the *Dinkum C++ Library Reference* (included with your distribution, at ***install_dir*/scxx/html/index.html**)
- **18.1: Types** — `NULL` is defined as `(0)`.
- **18.3: Start and termination** — `exit()` returns the value of argument `status`:
  - `EXIT_SUCCESS` — has a value of `0`.
  - `EXIT_FAILURE` — has a value of `1`.
- **18.4.2.1: Class bad_alloc** — `what()` returns an empty string.
- **18.5 to 18.6** — Various class definitions. See *Dinkum C++ Library Reference*.

- **20.1.5: Allocator requirements** — Match those in the standard. Certain operations on containers and algorithms will behave differently if allocator instances compare non-equal. Instances of `std::allocator` always compare equal.
- **21.1.3.1: struct char_traits<char>** — Types `streampos` and `streamoff` are typedefs for `fpos<mbstate_t>` and `long`, respectively. Type `wstreampos` is a typedef for `streampos`.
- **22.2: Standard locale categories** — Various type definitions. See *Dinkum C++ Library Reference*.
- **22.2.5.1.2: time_get virtual functions** — Two-digit year numbers are supported. Numbers in the range [0,68] represent years 2000 to 2068. Numbers in the range [69,136] represent years 1969 to 2036. See *Dinkum C++ Library Reference*.
- **23.2 to 23.3** — Various type and class definitions. See *Dinkum C++ Library Reference*.
- **26.2.8: complex transcendentals** — `pow(0,0)==(1,0)`
- **26.3: Numeric arrays** — Various type and class definitions. See *Dinkum C++ Library Reference*.
- **27.1.2: Positioning Type Limitations** — In Green Hills standard libraries, `traits::pos_type` and `traits::off_type` are typedefs for `streampos` and `streamoff` respectively.
- **27.4.1: Types** — `streamoff` is a typedef for `long`.
- **27.4.2.4: ios_base static members** — The synchronization flag is ignored. Streams are always synchronized.
- **27.4.4.3: basic_ios iostate flags functions** — On failure, the object to be thrown is constructed with a string argument that is the first applicable of `ios_base::badbit set`, `ios_base::failbit set`, or `ios_base::eofbit set`.
- **27.7.1.3: Overridden virtual functions** — `basic_streambuf::setbuf()` has no effect.
- **27.8.1.4: Overridden virtual functions** — `basic_filebuf::setbuf()` with non-zero arguments invokes the C library function `setvbuf` with the given arguments, scaled appropriately, and with a mode argument of `_IOFBF`.

- **B: Implementation quantities** — In order to guard against infinite recursion, the compiler places limits on the following quantities:
  - Nesting levels for `#include` files (10 recursive inclusions of the same file; no general limit)
  - Recursively nested template instantiations (64).

  Associated diagnostics include 3 and 456.
- **C.1.9.16.8: Predefined names** — `__STDC__` is defined as `0`.
- **C.2.2.3: Macro NULL** — Is defined as `(0)`.
- **D.6: Old iostreams members** — See 21.1.3.1 for `streampos` and `streamoff` typedefs.

Deprecated C++ Headers

The Green Hills C++ libraries provide all 51 of the headers specified by the *International Standard ISO/IEC 14882:2003*, as well as the common non-standard headers `<hash_map>`, `<hash_set>`, and `<slist>`.

The following non-standard header files are provided only for the support of legacy C++ code. These header files are deprecated, and may be removed in a future release:

```
<fstream.h>
<iomanip.h>
<iostream.h>
<new.h>
<rope>
<stdiostream.h>
<stl.h>
<strstream.h>
```

Chapter 14

Coding Standards

Contents

Coding standards are sets of rules that enforce a stricter coding standard than C or C++. The Green Hills tools ship with support for some widely-adopted standards (such as MISRA), and provide a feature called *coding standard profiles* to help you create your own. This chapter explains the standards that ship with the tools, how to enable them, and how to create your own standards.

GHS Standard Mode

GHS Standard Mode is a collection of compiler warnings and errors that enforces a stricter coding standard than regular C and C++. It was designed to aid you in avoiding common pitfalls when designing complex programs. Green Hills uses this standard internally, and has found it provides good protection against common mistakes and bad coding style, without causing too many false positives. We recommend that you use this standard unless you are required to use MISRA for your project.

Several versions of the standard may be available, allowing you to use the latest version or standardize on a previous one. The diagnostics enabled by this standard are listed in Appendix A, “Coding Standard Profile Reference” on page 891.

The 2010 version of GHS Standard Mode is implemented through the ***install_dir*/defaults/coding_standards/ghstd2010.csp** coding standard profile. To use this profile:

Set the **Compiler Diagnostics→Coding Standard Profile** option to `ghstd2010` (**--coding_standard=ghstd2010**), or use the **--ghstd=** option (see “Green Hills Standard Mode” on page 235).

MISRA C 1998

The MISRA C 1998 Standard is a set of guidelines for the C programming language created by the Motor Industry Software Reliability Association (MISRA). These guidelines are designed to enforce good practices in the development of embedded automotive systems. Complete details can be found in the *Guidelines For The Use Of The C Language In Vehicle Based Software, Motor Industry Software Reliability Association, April 1998* book. For more information, see the MISRA Web site [http://www.misra.org.uk/].

If the compiler is invoked with options enabling both MISRA 1998 rules and MISRA 2004 rules, the compiler does not directly enforce any MISRA 1998 rules. Rather, the compiler enforces the MISRA 2004 equivalents (if they exist) for any enabled MISRA 1998 rules.

There are 127 rules, classified into two categories, as follows:

- 93 *required* rules (designated **[R]** below) — mandatory requirements placed on the programmer that will, by default, generate an error if enabled and breached.
- 34 *advisory* rules (designated **[A]** below) — suggestions to the programmer that will, by default, generate a warning if enabled and breached.

The MISRA rules are organized into 17 groups, each focusing on a particular C language topic (for example: types, functions, expressions, the C preprocessor, etc). Each group has an associated option below. For information about each group and individual rules, see "MISRA C 1998" on page 188.

The Green Hills Tools provide options to control the severity of messages that alert you to violations of the rules. Most of the messages are given at compile time, although some are generated at run-time. Because run-time checking can adversely affect code performance, the tools provide the **--no_misra_runtime** option to disable it.

Some MISRA rules are enforced by disabling toolchain features such as recognition of certain language extension keywords. A violation of one of MISRA rules results in an error if the rule is enabled, regardless of the diagnostic severity specified for that rule. These MISRA rules are not affected by `#pragma ghs startnomisra` or `#pragma ghs endnomisra` (see "Green Hills Extension Pragma Directives" on page 753).

Some MISRA rules change other command line controllable options as documented in the following section. The MISRA rules override explicit settings of the other command line options, even if these options appear later on the command line than the MISRA option.

Enforcing MISRA C 1998 Rules

A single driver option allows you to enable any or all of the MISRA rules. It may be used in any of the following forms (where *n* is a rule number from the list below):

- **--saferc=all** — Enables checking of all the rules
- **--saferc=none** — Disables checking of all the rules
- **--saferc=*n*[,*n1*…]** — Enables checking of a comma-separated list of rules
- **--saferc=*n*-*n1*** — Enables checking of a range of rules
- **--saferc=all,-7** — Enables checking of all the rules except rule 7.
- **--saferc=7–10** — Enables checking of rules 7 though 10.
- **--saferc=7,10** — Enables checking of rules 7 and 10.

MISRA C 2004

The MISRA C 2004 Standard was created by the Motor Industry Software Reliability Association (MISRA). It is a set of guidelines for the C programming language that is designed to enforce good practices in the development of embedded automotive systems. You can find complete details in the *Guidelines For The Use Of The C Language In Critical Systems, Motor Industry Software Reliability Association, October 2004* book. For more information, see the MISRA Web site [http://www.misra.org.uk/].

There are 141 rules, classified into two categories, as follows:

- 120 *required* rules (designated **[R]** below) — mandatory requirements placed on the programmer that will, by default, generate an error if enabled and breached.
- 21 *advisory* rules (designated **[A]** below) — suggestions to the programmer that will, by default, generate a warning if enabled and breached.

The MISRA rules are organized into 21 groups, each focusing on a particular C language topic (for example: types, functions, expressions, the C preprocessor, etc.). For information about each group and individual rules, see “MISRA C 2004” on page 168.

Enforcing MISRA C 2004 Rules

A single driver option allows you to enable any or all of the MISRA rules. It may be used in any of the following forms (where *n* is a rule number):

- **--misra_2004=all** — Enables checking of all the rules.
- **--misra_2004=none** — Disables checking of all the rules.
- **--misra_2004=*n*[,*n1*…]** — Enables checking of a comma-separated list of rules.
- **--misra_2004=-*n*[,-*n1*...]** — Disables checking of a comma-separated list of rules.
- **--misra_2004=*n*-*n1*** — Enables checking of rules *n* through *n1*.
- **--misra_2004=-*n*-*n1*** — Disables checking of rules *n* through *n1*.

For a list of all supported MISRA C 2004 rules, see "MISRA C 2004" on page 168.

When the **--diag_*=** options and **--misra_2004=** options are both used in a single project, both options are passed to the compiler and both are used.

The compiler first uses the **--misra_2004=** options to determine whether certain MISRA behavior is enabled. If a MISRA rule is not enabled based on the **--misra_2004=** options, the compiler applies the **--diag_*=** options to determine the severity of the message.

If a MISRA rule is enabled based on the **--misra_2004=** options, the compiler follows MISRA-specific logic to determine if a message should be given, and then usually applies the **--diag_*=** options to determine the severity and suppression of the message. It is not always possible to change the severity of a MISRA rule using **--diag_*=**.

Coding standard profiles, such as Green Hills Standard Mode, are implemented using the same mechanism as **--diag_*=** and therefore have a similar interaction with **--misra_2004=**.

The following example demonstrates the use of these MISRA options:

Example 14.1. Using MISRA Builder and Driver Options

--misra_2004=all,-2.3 — Enables checking of all rules except rule 2.3.

--misra_2004=2.3-4.2 — Enables checking of rules 2.3 through 4.2.

--misra_2004=2.3,4.2 — Enables checking of rules 2.3 and 4.2.

--misra_2004=-2.3,-4.2 — Disables checking of rules 2.3 and 4.2.

MISRA C 2004 Implementation

The Green Hills tools provide options to control the severity of messages that alert you to violations of MISRA C 2004 rules. Most of the messages are given at compile time, although some are generated at run-time. Because run-time checking can adversely affect code performance, an option is provided to disable it.

Most of the rules that generate messages at compile time are implemented though compiler checks that are only enabled when the corresponding MISRA rule is enabled. Some of the rules specify constraints that the compiler already has checks for. In these cases, the severity of the diagnostic message will be elevated to the severity specified for the corresponding MISRA rule if it exceeds the default severity for that message.

In some cases a MISRA rule has been implemented using a number of distinct compile time checks that each generate unique diagnostic messages when they are violated. This means that one item that violates one MISRA rule might cause multiple diagnostics to be issued if it violates more than one of the checks used to implement the rule. The advantage of this approach is that by modifying the severity of or suppressing individual diagnostic messages, the user can allow a deviation from a portion of a MISRA rule without disabling the entire rule. For example, if you have **Rule 8.1** enabled but want to allow static functions to be defined without a separate prototype declaration, you can specify **--diag_remark=1828** or **--diag_suppress=1828** to disable the portion of **Rule 8.1** that requires static function definitions to have separate prototype declarations.

In the introduction to section 6.15, "Switch statements," MISRA-C:2004 defines a restricted switch statement syntax. Some of the syntax restrictions are enforced as part of the rules contained in section 6.15. The remainder of the syntax restrictions are enforced by the compiler by default if any of the rules from section 6.15, 15.1-15.5, are enabled. Because all of the rules in section 6.15 are required rules, the diagnostic severity for the restricted switch statement syntax is set to the required

rule severity level by default. The compiler diagnostics issued for these syntax restrictions include 1705, 1706, 1708, 1709, and 1821.

The compiler toolchain enforces some MISRA rules by disabling features or changing logic that is not specific to MISRA rule checking. If you enable checking for one of these rules, and your code violates that rule, you will receive an error regardless of the specified MISRA diagnostic level. Because the compiler toolchain does not issue the error due to the MISRA rule directly, the error message will not contain a reference to that rule. These MISRA rules are not affected by `#pragma ghs startnomisra` or `#pragma ghs endnomisra` (see "Green Hills Extension Pragma Directives" on page 753).

For example, you might enable checking for MISRA required rule 1.1, which enables strict ANSI C by disabling recognition of certain non-standard keywords. Because the compiler can no longer recognize those keywords, it cannot parse code that violates the rule. As a result, the compiler will issue a non-discretionary error even if you have used `--misra_req=silent` to downgrade and hide MISRA required rule errors.

Some MISRA rules are enforced through preprocessor directives in the standard headers provided by Green Hills and are only enforced when the standard headers are used.

Some MISRA rules change other command line controllable options as documented in the following section. The MISRA rules override explicit settings of the other command line options, even if these options appear later on the command line than the MISRA option.

Creating Your Own Coding Standards with Coding Standard Profiles

Coding standard profiles are a convenient way to specify your own coding standards. You can use coding standard profiles to:

- print custom prefixes in front of diagnostics.
- group diagnostics together under tags, which you can use in place of diagnostic numbers for various options and `#pragma` directives.
- easily distribute a single coding standard to several developers.

The following sections explain how to use the preset coding standard profiles that ship with the Green Hills tools, as well as write your own.

Specifying a Coding Standard Profile

The Green Hills tools ship with preset coding standard profiles located in ***install*/defaults/coding_standards**:

- **ghstd2010.csp** — see " GHS Standard Mode" on page 892
- **misra1998.csp** — (provided as an example; does not fully implement MISRA 1998) see "MISRA C 1998" on page 895

You can also create your own (see "Coding Standard Profile Syntax" on page 731).

To specify a coding standard profile for use with your project:

Set the **Compiler Diagnostics→Coding Standard Profile** (**--coding_standard=**) option (see "Coding Standard Profile" on page 236).

After specifying a coding standard profile, you can use the tags defined by that profile in place of diagnostic numbers for options such as **--diag_warning** and **--diag_error**, and in `#pragma ghs nowarning` (see "Varying Message Severity" on page 240 and "Green Hills Extension Pragma Directives" on page 753).

Note

The MISRA C 2004 standard is available through a set of driver options. For more information, see "MISRA C 2004" on page 168.

Coding Standard Profile Syntax

Coding standard profiles are text files. By convention, they have a **.csp** extension. The file is divided into two sections:

1. **Tag Section** — Associates diagnostic messages with tags.
2. **Severity Section** — [optional] Associates tags with a default severity level.

Each line in the tag section has the following syntax:

diagnostic "*prefix*" *tag* [*tag*]…

- *diagnostic* — A valid diagnostic number, or zero (in which case the line has no effect). If the diagnostic number is out of range, the compiler issues a warning and the line is ignored. For a list of diagnostic numbers, see *MULTI: C and C++ Compiler Error Messages*.
- *prefix* — A string that prints before each diagnostic message of this type.
- *tag* [*tag*]… — A whitespace-separated list of tags, terminated by the end of the line or the start of a comment (`//`). Each tag must follow the rules for C identifiers.

Each line in the severity section has the following syntax:

severity tag [*tag*]…

- *severity* — Must be one of `diag_suppress`, `diag_remark`, `diag_warning`, or `diag_error`.
- *tag* [*tag*]… — A whitespace-separated list of tags specified in the tag section, terminated by the end of the line or the start of a comment (`//`).

Comments begin with the sequence `//` and continue until the end of the line.

For example, if you have the following program **test.c**:

```
#define foo @
void bar();;
```

and the following coding standard profile **my_profile.csp**:

```
// Extra semicolons are bad
7   "My Standard 1.1" my_tag my_tag_1_1
381 "My Standard 1.2" my_tag my_tag_1_2
7   ""                silent_tag
diag_warning my_tag
diag_suppress silent_tag
```

the program compiles cleanly without additional options:

```
> ccv850 -c test.c
```

When you use the **--coding_standard** option to specify your standard, the compiler uses the severity levels defined in the standard's severity section. If the same diagnostic has multiple tags, the compiler uses the last-defined entry of the severity section that specifies one of the tags, and uses the prefix defined for that tag. For example, in **my_profile.csp**, the last-defined severity for diagnostic 7 is `diag_suppress silent_tag`, so it does not appear by default:

```
> ccv850 -c test.c --coding_standard=my_profile.csp
"test.c", line 2: warning #381-D: My Standard 1.2: extra ";" ignored
  void bar();;
             ^
```

Specify a tag for the **--diag_suppress**, **--diag_remark**, **--diag_warning**, or **--diag_error** option to override the levels defined in the severity section:

```
> ccv850 -c test.c --coding_standard=my_profile.csp --diag_warning=my_tag
"test.c", line 2: warning #7-D: My Standard 1.1: unrecognized token
  #define foo @
              ^
"test.c", line 2: warning #381-D: My Standard 1.2: extra ";" ignored
  void bar();;
             ^
```

If the compiler uses a tag for a diagnostic whose prefix is an empty string, no additional characters are printed in standard error output:

```
> ccv850 -c test.c --coding_standard=my_profile.csp --diag_warning=silent_tag
"test.c", line 1: warning #7-D: unrecognized token
  #define foo @
              ^
"test.c", line 2: warning #381-D: My Standard 1.2: extra ";" ignored
  void bar();;
             ^
```

If you pass a tag that does not exist in any specified profile, the compiler issues a warning:

```
> ccv850 -c test.c --coding_standard=my_profile.csp --diag_warning=bad_tag
Warning #1966-D: unused coding standard tag or malformed error number bad_tag
```

Chapter 15

Macros, Pragma Directives, and Intrinsics

Contents

Predefined Macro Names

The preprocessor defines a number of macro names, depending on the compiler and language you use and the builder and driver options you specify. To write code that compiles differently depending on which macro names the preprocessor defines, use the `#ifdef` preprocessor directive.

The following sections contain lists that cover all of the preprocessor predefined macro names.

Macro Formats

The standards for C and C++ require that all compiler-predefined macros begin with one or two underscore characters. Prior to the ANSI standard, some C preprocessors defined macros without leading underscores (such as `ghs` or `unix`) that could conflict with identifiers in your program.

Each Green Hills macro is defined with two leading underscores and two trailing underscores in all language modes, unless stated otherwise.

Macros marked with an asterisk (`*`) are also defined by default without the two trailing underscores.

When using C or C++ modes other than Strict ANSI C, you can instruct the preprocessor to generate versions of macros marked with `*` that do not have a leading or trailing underscore by enabling the **Advanced→Preprocessor Options→Definition of Unsafe Symbols** (**--unsafe_predefines**) option.

All predefined macros have the value `1`, unless stated otherwise.

Macro Names Required by ANSI C and C++

The standards for C and C++ require the preprocessor to define certain macro names. These macro names and their values are shown in the table below.

`__cplusplus`

The value of this macro is:

- `199711L` — in ANSI C++
- `1` — in other modes of C++ (EC++, EEC++, ARM, GNU)
- Undefined in C

`__DATE__`

A string literal representing the date of the current compilation, which is 11 characters long, and in the form: *mmm dd yyyy*. For example: `Apr 01 2001`

`__FILE__`

A string literal representing the name of the current source file (for example, `file.c`). For the base file, `__FILE__` includes the pathname. If the compiler reaches the file through an `#include` directive, `__FILE__` contains the pathname (if any) from the include path through which the file was found. If the include path is relative, it is appended to the pathname of the base file.

`__LINE__`

An integer literal representing the line number in the current source file.

`__STDC__`

The value of this macro is:

- `0` — in C++ and ANSI C mode
- `1` — in Strict ANSI C mode and C99 mode
- Undefined in K&R C

`__STDC_VERSION__`

The value of this macro is:

- `199901L` — in C99 mode
- `199409L` — in ANSI C and GNU C
- Undefined in C++ and K&R C

`__STDC_HOSTED__`

The value of this macro is:

- `1` — in C++ and all modes of C other than K&R C
- Undefined in K&R C

`__TIME__`

A string literal representing the local time of the current compilation, which is 8 characters long, and in the form: *hh:mm:ss*. For example: `"11:46:51"`

Language Identifier Macros

These macros identify the language used:

Primary Language Macros

`__LANGUAGE_ASM__` Language is assembly.	*
`__LANGUAGE_C__` Language is C.	*
`__LANGUAGE_CXX__` Language is C++.	*

C Language Macros

`__GNUC__` GNU C or C++ mode (**-gcc** or **--g++**).
`__GNUC_GNU_INLINE__` GNU C or GNU C99 mode (**-gcc** or **-gnu99**) when GNU-style inlining is used.
`__GNUC_STDC_INLINE__` GNU C or GNU C99 mode (**-gcc** or **-gnu99**) when GNU99-style inlining is used.
`__Japanese_Automotive_C__` Japanese Automotive C (see“Japanese Automotive C Extensions” on page 643).
`__PROTOTYPES__` All modes of C except K&R mode. All modes of C++.
`__STRICT_ANSI__` Language is Strict ANSI C (**-ANSI**) or Strict C99 (**-C99**). Undefined in C++.

C++ Language Macros

`__c_plusplus` Language is C++ (this is for backwards compatibility with some older C++ implementations). When writing new code, use `__cplusplus`.

`__embedded_cplusplus` Language is Embedded or Extended Embedded C++.
`__EMBEDDED_CXX` Language is Embedded C++ (for example, option **--e**).
`__EMBEDDED_CXX_HEADERS` The Embedded C++ header files and libraries are used (for example, options **--el** or **--ele**).
`__EXTENDED_EMBEDDED_CXX` Language is Extended Embedded C++ (ESTL) (for example, option **--ee**).
`__EXTENDED_EMBEDDED_CXX_HEADERS` The Extended Embedded C++ header files and libraries are used (for example, options **--eel** or **--eele**).
`__STANDARD_CXX` Language is Standard C++.
`__STANDARD_CXX_HEADERS` The Standard C++ header files and libraries are used (for example, options **--stdl** or **--stdle**).

Green Hills Toolchain Identification Macros

`__EDG__` C or C++ compilers.
`__ghs__` * Any Green Hills Software compiler or preprocessor.
`__ghs_asm` Indicates that the Green Hills assembler is in use.
`__GHS_REVISION_DATE` The date of the compiler, represented as a string similar to `__DATE__`.
`__GHS_REVISION_VALUE` Represents the date of the compiler build in the `time_t` format.
`__GHS_VERSION_NUMBER` Represents the version of the toolchain as a six-digit decimal integer. For example, in MULTI , Compiler 2012.0, this macro would be defined as `201200`.
`__gnu_asm` Indicates that the GNU assembler is in use.

`__unix_asm` Indicates that the UNIX assembler is in use.

V850 and RH850-Specific Predefined Macro Names

The following table lists predefined macros specific to the V850 and RH850 processor family. For a complete list of individual processor macros, see "V850 and RH850 Processor Variants" on page 47.

`__V800__` * Renesas V800 Series processor family.
`__V800_ignore_callt_state_in_interrupts__` * `CTPSW` and `CTPC` registers are not saved by compiler-generated interrupt prologues.
`__V800_no_callt__` * `callt` instructions are not generated.
`__V800_r20has255__` * Register `r20` always contains the value 255.
`__V800_r21has65535__` * Register `r21` always contains the value 65535.
`__V800_registermode__=`*n* * Represents registers reserved for user. *n* is set to one of the following values: • 22: `r15–r24` are reserved for user • 26: `r17–r22` are reserved for user • 32: No registers are reserved for user
`__V800_reserve_r2__` * Register `r2` is reserved for user.
`__V850__` * `__V851__` Both of these symbols are defined for any `v850` family processor

`__V850__`
`__V850e__`
One of these symbols is defined depending on whether the processor belongs to the `v850e` family.

Note

For a list of individual processor macros, see "V850 and RH850 Processor Variants" on page 47.

Endianness Macros

One of these symbols is always defined to specify the endianness of the target processor.

`__BIG_ENDIAN__` Big Endian byte order.
`__LITTLE_ENDIAN__` Little Endian byte order.

Data Type Macros

These macros specify the size and signedness of certain data types.

Sizes

`__CHAR_BIT` Defined as the size of `char` in bits.
`__FUNCPTR_BIT` Defined as the size of a function pointer in bits.
`__INT_BIT` Defined as the size of `int` in bits.
`__LONG_BIT` Specifies the size of `long` in bits.
`__LLONG_BIT` Specifies the size of `long long` in bits. Defined only if the `long long` type is supported.
`__PTR_BIT` Specifies the size of a pointer in bits.
`__REG_BIT` Specifies the size of an integer register in bits.

`__SHRT_BIT` Specifies the size of `short` in bits.
`__WCHAR_BIT` Specifies the size of `wchar_t` in bits.

Signedness

`__Field_Is_Signed__`
`__Field_Is_Unsigned__`
Bitfields are signed or unsigned.

`__Ptr_Is_Signed__`
`__Ptr_Is_Unsigned__`
Pointers are signed or unsigned.

`__Char_Is_Signed__`
`__SIGNED_CHARS__`
`__Char_Is_Unsigned__`
`__CHAR_UNSIGNED__`
Type `char` is signed or unsigned.

`__WChar_Is_Signed__`
`__WChar_Is_Unsigned__`
Type `wchar_t` is signed or unsigned.

Floating-Point Macros

These macros specify the manner in which floating-point operations are performed.

`__DOUBLE_HL__` * Passed if the high word of a double comes at the lower address in memory. This is usually the case for big endian targets, although certain targets behave differently.
`__IeeeFloat__` * IEEE-754 floating-point format. This symbol is always defined.

Floating-Point Mode Macros

One of these symbols is always defined, unless full hardware floating point is enabled.

`__NoFloat__`	*
No floating point mode.	
`__SoftwareDouble__`	*
Double precision floating point uses integer instructions.	
`__SoftwareFloat__`	*
All floating point is done with integer instructions.	

MISRA Macros

These macros relate to the MISRA C variant.

`__MISRA_`*`i`*
MISRA diagnostic levels. *i* is either 8 or 118–127. The macro is defined as one of the following: • 0: Silent • 1: Warn • 2: Error For more information about the MISRA rules, see “MISRA C 1998” on page 188.
`__ONLY_STANDARD_KEYWORDS_IN_C`
MISRA rule No. 1

Memory Model Macros

These macros relate to various different forms of memory model.

Position Independent Code and Data Macros

`__ghs_pic` Position Independent Code (PIC) is enabled. This macro is defined as one of the following: • 1: Deprecated Green Hills behavior. In this mode, static or global initializers to position-independent objects were not supported. The compiler will warn for any unsupported position independent initializers. • 2: Normal Green Hills behavior. Special start-up code is invoked to fix up static or global initializers to position-independent objects.	*
`__ghs_pid` Position Independent Data (PID) is enabled. This macro is defined as one of the following: • 1: Deprecated Green Hills behavior. In this mode, static or global initializers to position-independent objects were not supported. The compiler will warn for any unsupported position independent initializers. • 2: Normal Green Hills behavior. Special start-up code is invoked to fix up static or global initializers to position-independent objects.	*

Small Data Area Macros

`__ghs_sda` Defined if Small Data Area optimization is supported. If this macro is not defined, instances of `#pragma ghs startsda` will be ignored with a warning.	*
`__ghs_sda_threshold` Defined as the size, *n*, of the SDA threshold (as specified by the **-sda=*n*** option).	*

Tiny Data Area Macros

`__ghs_tda` Defined if Tiny Data Area optimization is supported. If this macro is not defined, instances of `#pragma ghs starttda` will be ignored with a warning.	*

Zero Data Area Macros

`__ghs_zda`	*
Defined if Zero Data Area optimization is supported. If this macro is not defined, instances of `#pragma ghs startzda` will be ignored with a warning.	
`__ghs_zda_threshold`	*
Defined as the size, *n*, of the ZDA threshold (as specified by the **-zda=*n*** driver option).	

Global Register Macros

`__GlobalRegisters`
Corresponds to the **-globalreg=*n*** option, where *n* is a non-negative positive integer.

Global Variable Macros

`__GHS_NOCOMMONS__`	*
Uninitialized global variables in C do not exhibit `COMMON` variable behavior (corresponds to the **--no_commons** driver option).	

Alignment and Packing Macros

These macros specify the size of data alignment and packing.

`__ghs_alignment`
Alignment of the most-aligned type, which is either `double`, `long`, or `long long`.
Has a value of 1, 2, 4, or 8.
`__ghs_max_pack_value`
Represents the largest value that may be passed to `#pragma pack(n)`.
`__ghs_packing`
Represents value specified with the **-pack=*n*** option on the command line. Has a value of 1, 2, 4, or 8. If the macro name is not defined, the **-pack=** option was not specified on the command line. This value is not changed by the `#pragma pack(n)` directive.

Current File and Function Macros

These macros identify the current file and function. See also `__FILE__` in "Macro Names Required by ANSI C and C++" on page 734.

`__BASE__` A string literal representing the name of the current source file, without the directory path.
`__FULL_DIR__` A string literal representing the full pathname of the directory of the current source file.
`__FULL_FILE__` A string literal representing the full pathname of the current source file.
`__FUNCTION__` A string literal representing the current function name. The compiler does not resolve this macro when you use the **-E** or **-P** options.
`__PRETTY_FUNCTION__` In C++, a string literal representing the fully qualified function name (for example, `mynamespace::myclass::method`). Note that overloaded functions with different signatures might have the same fully qualified name. In C, `__PRETTY_FUNCTION__` behaves the same as `__FUNCTION__`. The compiler does not resolve this macro when you use the **-E** or **-P** options.

Object Format Macros

These macros specify the object format.

`__COFF__` * Object file type is COFF.
`__ELF__` * Object file type is ELF.

Optimization Predefined Macro Names

These macros relate to the optimization strategy employed.

`__GHS_Inline_Memory_Functions` Inlining of C Memory Functions (see "Inlining of C Memory Functions" on page 303).

`__GHS_Inline_String_Functions` Inlining of C String Functions is enabled (see “Inlining of C String Functions” on page 303).
`__GHS_Optimize_Inline` Two-pass inlining is enabled see “Two-Pass Inlining (Intermodule Inlining)” on page 298).

C++ Macros

These macros relate to various features of C++.

`__ARRAY_OPERATORS` Defined when array new and delete are enabled (that is, `operator new[]` and `operator delete[]`).
`_BOOL` Defined when `bool` is recognized as a basic type (for example, as with the option **--bool**).
`__EDG_IMPLICIT_USING_STD` Defined when the standard namespace `std` is implicitly used (as with the option **--using_std**).
`__EDG_RUNTIME_USES_NAMESPACES` Indicates that the C++ Run-time libraries use namespaces.

`__EXCEPTION_HANDLING`
`__EXCEPTIONS`
Indicates C++ compiler is running in a mode that allows exception handling.

`__NAMESPACES` Indicates C++ namespaces are accepted.
`__RTTI` Indicates Run-Time Type Identification code accepted.
`_WCHAR_T` Defined if `wchar_t` is a keyword.

Operating System-Specific Macros

These macros identify the target operating system.

ThreadX Macros

`__THREADX` ThreadX real-time operating system.
`TX_ENABLE_EVENT_LOGGING` ThreadX event logging enabled.

Miscellaneous Macros

`__COUNTER__` The value of the `__COUNTER__` macro increases by one each time it is expanded. On the first expansion, its value is `0`.

Deprecated Macros

These macros are deprecated and may not be supported in future versions of MULTI.

`__Char_Is_Signed__` Type `char` is `signed char`.
`__Char_Is_Unsigned__` Type `char` is `unsigned char`.
`__Int_Is_32` Type `int` is 4 bytes.
`__Int_Is_64` Type `int` is 8 bytes. (The macro `__Int_Is_64` is an old style macro, kept for backward compatibility. The new style macros (like `__INT_BIT=`*n*) should be used when possible.)
`__msw` * Any version of Microsoft Windows (deprecated).
`__LL_BIT` Describes the size of `long long` in bits.
`__LL_Is_64` Type `long long` is 8 bytes (when `long long` is an allowed type).
`__Long_Is_32` Type `long` is 4 bytes.

`__Long_Is_64` Type `long` is 8 bytes.
`__Ptr_Is_32` Pointers are 4 bytes.
`__Ptr_Is_64` Pointers are 8 bytes.
`__Reg_Is_32` CPU has 32-bit registers.
`__Reg_Is_64` CPU has 64-bit registers.
`__WChar_Is_Int__` Type `wchar_t` are `int` or `unsigned int`.
`__WChar_Is_Long__` Type `wchar_t` are `long` or `unsigned long`.
`__WChar_Is_Short__` Type `wchar_t` is `short` or `unsigned short`.

Pragma Directives

`#pragma` directives allow individual compiler implementations to add special features to C programs without changing the C language. Programs that use `#pragma` directives stay relatively portable, although they make use of features not available in all ANSI C implementations.

According to the ANSI standard, the `#pragma` directives that are not recognized by the compiler should be ignored. This requirement may help when porting code between compilers because one compiler will recognize and process a certain `#pragma` directive, while a different compiler (which does not need or recognize that `#pragma` directive) will, by default, generate a warning and ignore it.

If a `#pragma` directive which would ordinarily be recognized is misspelled, the compiler will not recognize it and will, by default, generate a warning and ignore it. You can increase the severity of these messages to errors, or suppress them, as follows:

Set the **Compiler Diagnostics→C/C++ Messages→Unknown Pragma Directives** option to **Errors** (**--unknown_pragma_errors**) or **Silent** (**--unknown_pragma_silent**).

The majority of Green Hills proprietary `#pragma` directives begin with the keyword `ghs` to differentiate them from other implementations. The compiler considers any `#pragma` beginning with the `ghs` keyword to be recognized.

If the compiler recognizes a `#pragma` directive, it will perform some limited syntax checking. `#pragma` directives that fail these checks will, by default, generate warnings, and be ignored. You can increase the severity of these messages to errors, or suppress them, as follows:

Set the **Compiler Diagnostics→C/C++ Messages→Incorrect Pragma Directives** option to **Errors** (**--incorrect_pragma_errors**) or **Silent** (**--incorrect_pragma_silent**).

General Pragma Directives

The general `#pragma` directives are listed in the following table:

`#pragma alignfunc (`*n*`)` Instructs the compiler to make the next function *n*-byte aligned, where *n* is 1, 2, 4, 8, 16, 32, 64, or 128.
`#pragma alignvar (`*n*`)` Instructs the compiler to make the next variable *n*-byte aligned, where *n* is a power of 2 and no greater than 128. If the next variable is a non-static local variable, this directive is only effective up to the stack alignment maintained by the ABI.

`#pragma asm`
 	
`#pragma endasm`
 	
These directives mark the beginning and end of a section of code which is to be copied literally to the assembly language output file. Any macros or preprocessing directives will be processed, but no processing of comments will be performed. The construct `#pragma endasm` may not contain any comments or `\` escapes, although it may contain spaces or tabs before and after the `#` character, as long as the total length of the line does not exceed 255 characters.
 	
Note that this directive is not supported with C++ source code.

`#pragma ident "`*string*`"` Inserts *string* into the output object file's `.comment` section.
`#pragma inline `*function-list* This `#pragma` is deprecated. Instructs the compiler to consider each function in the comma-separated *function-list* as a candidate for inlining. Note that you cannot force the compiler to inline any function.
`#pragma instantiate `*fn* `#pragma can_instantiate `*fn* `#pragma do_not_instantiate `*fn* These directives can be used to control the instantiation of a template function, *fn*. For full documentation, see “Template Instantiation” on page 702.
`#pragma intvect `*intfunc integer_constant* Instructs the compiler to insert a jump to the function *intfunc* at the address specified by *integer_constant*. For more information, see “Interrupt Routines” on page 111.

`#pragma once`

If this `#pragma` is placed at the beginning of a header file, the compiler will only include the file once, and will ignore subsequent `#include` statements that reference the file.

`#pragma pack (`*n*`)`

`#pragma pack ()`

`#pragma pack (push {, `*name*`} {, `*n*`})`

`#pragma pack (pop {, `*name*`} {, `*n*`})`

Instructs the compiler to lay out subsequent `class`, `struct`, and `union` definitions in memory with a maximum alignment of *n*, where *n* is a power of two and less than or equal to the value of `__ghs_max_pack_value`.

The second format, where *n* is not specified, reverts the maximum alignment to the default value, or that specified by the Builder or driver.

The third and fourth formats allow you to push and pop maximum alignment values, and to specify a *name* for the pushed value, so that it can subsequently be popped back to by name.

`#pragma __printf_args`

`#pragma __scanf_args`

These directives should be used immediately before a function declaration or definition.

If the function has a variable number of arguments and the last specified argument is of type `char *`, the function is treated as a variadic function like `printf` or `scanf`.

The compiler will expect that the last specified argument is a character string, with `printf` or `scanf` formatting sequences (such as `"%s"`), and will check the extra arguments to ensure that the type and number match the type and number specified in the format string.

`#pragma unknown_control_flow (`*function-list*`)`

Instructs the compiler to anticipate an *unexpected flow* from each function in the comma-separated *function-list*. Such a function would be `setjmp`. This information is used in various optimizations.

`#pragma weak foo`

Defines *foo* as a *weak symbol*. This `#pragma` may be used in either of the following ways:

- If *foo* is declared as an external reference in *filename*:

  ```
  #pragma weak foo
  extern int foo;
  ```

  and *foo* is not defined in any other file, then the address of *foo* is set to zero (or, in the case of projects that use PIC or PID, `0` plus the adjustment for the `text` or `data` segment offset). This will prevent an `undefined symbol` linker error.

- If *foo* is defined in *filename*:

  ```
  #pragma weak foo
  int foo = 1;
  ```

  and is linked with a non-weak definition of *foo* in another module, then that other definition will have priority, and the definition in *filename* will be treated as if it were an external reference. If all definitions are weak and there is more than one definition, it is undefined which definition is used.

By default, the linker does not pull in object files from libraries to resolve undefined weak symbols or to override weak definitions with stronger library definitions. To force the linker to resolve undefined weak symbols, use the **-extractweak** linker option.

`#pragma weak foo = bar`

This `#pragma` has the same effect as `#pragma weak foo`, except that (subject to the same conditions) the address of *foo* is set to the address of *bar*. It requires that *foo* is not defined in the current module, although it could be an external reference. *bar* must be defined at the outermost level of the current module. *bar* may not be a common symbol (see the **C/C++ Compiler→C/C++ Data Allocation→Allocation of Uninitialized Global Variables** option in "C/C++ Data Allocation" on page 200).

This `#pragma` may be used to provide a backwards-compatible alias to a symbol which has been renamed in a new version of a program, as follows:

```
#pragma weak oldname=name
int name()
{
  return 5;
}
```

If any old software depends on the existence of *oldname*, it will continue to work as if the function is called *oldname*. If there are no longer any references to *oldname*, then the symbol effectively disappears and can be reused.

When aliasing variables, behavior is undefined if you use more than one way to reference the actual variable.

Green Hills Extension Pragma Directives

The Green Hills extension `#pragma` directives are listed in the following table:

`#pragma ghs alias` *`newsym oldsym`* Creates a new global symbol *newsym* with the same address as *oldsym*. *newsym* cannot be defined in the current module or the behavior is undefined. *oldsym* must be defined at the outermost level of the current module. *oldsym* cannot be a common symbol (see the **C/C++ Compiler→C/C++ Data Allocation→Allocation of Uninitialized Global Variables** option in "C/C++ Data Allocation" on page 200). `#pragma ghs alias` *`newsym oldsym`* is similar to `#pragma weak` *`newsym=oldsym`* except that in the latter case, *newsym* is defined as a weak symbol. When aliasing variables, behavior is undefined if you use more than one way to reference the actual variable.

`#pragma ghs callmode=near|far|default`
 	
`#pragma ghs near`
 	
`#pragma ghs far`
 	
These `#pragma` directives control the call mode used to call subsequent functions.
 	
For more information about these directives, and other methods for controlling function call modes, see "Near and Far Function Calls" on page 84.

`#pragma ghs check=(`*`check-list`*`)`

Controls various run-time checks. Any run-time checks enabled either at the point of a function declaration (the prototype or the first unprototyped use) or at the point of a function definition will cause those run-time checks to be enabled for that function. Use the following words in the *check-list*.

Memory Checks. These checks do not affect other checks:

- `defaultmemory` — **Revert** to the **check=memory** or **check=nomemory** option as specified by the Builder or driver (see below).
- `memory` — **Memory Allocation (Intensive)** check

Other Checks. These checks do not affect memory checks:

- `all` — **All** checks.
- `assignbound` — **Assignment Bounds** check
- `bounds` — **Array Bounds** check
- `default` — **Revert** to the checks specified by the Builder or driver (see below).
- `nilderef` — **Nil Pointer Dereference** check
- `none` — **No** checks.
- `return` — **Return Without Value** check
- `switch` — **Case Label Bounds** check
- `watch` — **Write to Watchpoint** check
- `zerodivide` — **Divide by Zero** check

To turn off any of these checks, prepend `no` to the word. For example, to turn on all checks except **Divide by Zero**, use the following directive:

```
#pragma ghs check=(all,nozerodivide)
```

These directives require that you link in the Green Hills library **libind.a**.

For information about using run-time checks, see the **Debugging→Run-Time Error Checks** option in “Debugging Options” on page 156, and the documentation about viewing memory allocation information in the *MULTI: Debugging* book.

`#pragma ghs extra_stack (`*`n`*`)`

Specifies the extra stack usage of a particular function that is due to constructs such as Variable Length Arrays (VLAs), `alloca()` calls, and inline assembly, whose stack usages are not statically computed. This directive should appear immediately before the definition of the function it describes, and is used to provide **gstack** with information that is necessary to accurately compute the maximum stack usage for the program. For more information, see “Annotating C Functions for gstack” on page 563.

`#pragma ghs entry_exit_log=`*`option`*

`#pragma ghs entry_exit_log push` *`option`*

`#pragma ghs entry_exit_log pop`

Enable or disables function entry/exit (FEE) logging on a function-by-function basis, where *option* has one of the following values:

- `on` instruments subsequent functions, regardless of compiler options
- `off` disables instrumentation of subsequent functions, regardless of compiler options
- `default` instruments subsequent functions based on the specified compiler options

The `push pop` versions of this directive allow you to push the option state onto a stack and pop back to the previous option state. This behavior is useful for changing FEE logging in header files where you probably do not want to modify the including files's state. For example:

```
#pragma ghs entry_exit_log=on
// The state is: (0) on
void foo1() { } // FEE logged

#pragma ghs entry_exit_log push off
// The state is: (0) on (1) off
void foo2() { } // Not FEE logged

#pragma ghs entry_exit_log=default
// The state is: (0) on (1) default
void foo3() { } // FEE logged depending on default logging level

#pragma ghs entry_exit_log pop
// The state is: (0) on
void foo4() { } // FEE logged
```

For more information, see “Enabling Function Entry/Exit Logging” on page 59.

`#pragma ghs function [exporttda] tdata="`*`section-name`*`"`

`#pragma ghs function tda=default`

`#pragma ghs function`

Re-allocates data that would ordinarily be placed in a TDA section to the user-defined *section-name*.

The optional `exporttda` parameter, marks any appropriate functions as `exporttda`.

The second and third forms revert allocation to the TDA section.

`#pragma ghs inlineprologue`

`#pragma ghs noinlineprologue`

These directives control the inlining of function prologue and epilogue code.

`#pragma ghs inlineprologue` tells the compiler to inline the prologue and epilogue methods. This is useful when the prologue or epilogue routines are not callable.

`#pragma ghs noinlineprologue` lifts this restriction from the compiler and selects the most efficient method, given the optimization settings and the registers that must be saved.

This behavior can also be controlled with the **Advanced→Target Options→Function Prologues** Builder option and associated driver option (see "Target Options" on page 249).

`#pragma ghs interrupt[(enabled|nonreentrant)]`

Instructs the compiler to make the encompassing function, or the next function if in the file scope, an interrupt handling function.

The optional parameter `enabled` instructs the compiler to enable interrupts in the interrupt handler's prologue.

The optional parameter `nonreentrant` instructs the compiler to assume the interrupt handler cannot be interrupted, resulting in a smaller and faster interrupt handler.

`#pragma ghs io` *`identifier addr`*

Instructs the compiler to make any subsequent declaration of a variable with the name *identifier* a special I/O-variable, corresponding to IO port *addr*.

`#pragma ghs max_instances (`*`n`*`)`

Specifies the maximum number of instances of a function that can exist on the stack at a given time. This directive should appear immediately before the definition of the function it describes, and is used to provide **gstack** with an upper bound on possible recursions so it can accurately compute the maximum stack usage for the program. For more information, see "Annotating Recursive Functions" on page 570.

`#pragma ghs max_stack (`*`n`*`)`

Specifies the maximum stack usage for a function that is defined inside a `#pragma asm` block within a C file. A declaration for the function should be added outside the block, and this directive should come immediately before the declaration. This directive is used to provide **gstack** with information that is necessary to accurately compute the maximum stack usage for the program. For more information, see "Annotating Assembly Functions for gstack" on page 562.

`#pragma ghs nofloat interrupt`

Instructs the compiler to prevent the encompassing function or, if in file scope, the next function (which must be separately identified as an interrupt function) from saving and restoring floating-point registers. This pragma has no effect if the function prologue is not inline.

`#pragma ghs noprologue`

Instructs the compiler to omit register save and restore code. That is, all registers are treated as temporary, caller-save registers.

In addition, a return instruction is not generated.

`#pragma ghs noshorteload`

This `#pragma`, which must be placed directly before a structure definition, prevents the compiler from optimizing that structure with *short loads*.

`#pragma ghs nowarning` *n*

`#pragma ghs endnowarning`

Temporarily suppress diagnostic number *n*. You may use multiple and/or nested `#pragma ghs nowarning` directives, but each should have a matching `#pragma ghs endnowarning`.

A coding standard profile tag *tag* may be given in place of the diagnostic number *n*; this temporarily suppresses all discretionary diagnostics associated with that tag. For more information about coding standard profiles, see Chapter 14, "Coding Standards" on page 723.

To display error and warning numbers with diagnostic messages, enable the **Compiler Diagnostics→C/C++ Messages→Varying Message Severity→Display Error Message Numbers** option (**--display_error_number**).

Note: The location where `#pragma ghs nowarning` must be is dependent on the implementation of the warning in question, and is not always at the position where the warning is shown in the diagnostic output.

`#pragma ghs O`*string*

`#pragma ghs ZO`

`#pragma ghs O`*string* turns on optimizations, while `#pragma ghs ZO` turns them off. The optional *string* can contain any or all of the following letters:

- `L` —Loop optimizations
- `M` — Memory optimizations
- `S` — Small (but Slow) optimizations

Note: You can enable optimizations by omitting *string* from the directive.

For information about Builder and driver optimization options, see "Optimization Options" on page 145. For descriptions of many of our optimizations, see Chapter 4, "Optimizing Your Programs" on page 293.

`#pragma ghs reference` *sym*

Creates a reference to symbol *sym*, which can force the symbol to be pulled in from a library. It can also be used to prevent an unused function called *sym* from being deleted by the "Deletion of Unused Functions" optimization.

`#pragma ghs revertoptions` Disables all options set via `#pragma ghs` directives and returns to the defaults specified by the Builder or driver.

`#pragma ghs safeasm`
 	
`#pragma ghs optasm`
 	
These directives mark the next hand-written assembly language entity (for example, `#pragma asm`, `asm` macro, etc.) as "safe" for linker optimization analysis. Normally, hand-written assembly entities disqualify a module for linker optimization. If the entity is preceded by either of these directives, that entity does not disqualify the module for linker optimization. In order for a module to be considered "safe", each assembly entity must be marked with one of these pragma directives.
 	
`#pragma ghs optasm` marks the entity as safe and allows optimizations to occur within the entity.
 	
`#pragma ghs safeasm` marks the entity as safe for some optimizations, but unsafe for others. If you want the final executable to contain the exact assembly code you have written, do not use this `#pragma` directive.
 	
Only use these directives to mark assembly that does not contain any "unsafe" code. Unsafe code includes, but is not limited to, branches and control flow structures not normally generated by the compiler, and any code that violates the standard calling convention.

`#pragma ghs section` *`secttype`*`="`*`sectname`*`"` Instructs the compiler to allocate to *sectname* code or data that would normally be allocated to *secttype*. *sectname* must not contain spaces, quotes, parentheses, or wildcards. For more information, see "Custom Program Sections" on page 77.

`#pragma ghs start`*`data-type`*

`#pragma ghs end`*`data-type`*

These directives mark the beginning and end of a section of code in which all data should be allocated to a particular type of data section, where *data-type* is one of the following:

- `data` — normal data sections.
- `sda` — *small data area* sections (see "Special Data Area Optimizations" on page 87).
- `tda` — *tiny data area* sections (see "V850 Tiny Data Area (TDA) Optimization" on page 96).
- `zda` — *zero data area* sections (see "Special Data Area Optimizations" on page 87).

These directives will force the compiler to re-allocate data items to special data areas even if the **Target→Text and Data Placement→Special Data Area** or **Target→Text and Data Placement→V850 Tiny Data Area** option has not been enabled. The directives will also override any threshold which has been set with these options in order to control the maximum size of data items which can be allocated to a special data area.

You cannot nest these directives; each `#pragma ghs start`*`data-type`* must be completed by a matching `#pragma ghs end`*`data-type`* before another can be used.

`#pragma ghs start_externally_visible`

`#pragma ghs end_externally_visible`

These directives mark the beginning and end of a section of code where all symbols should be treated by optimizations as externally visible. Any functions or variables defined in this section will be considered visible outside of the source during Wholeprogram Optimizations. You can similarly specify that symbols are externally visible using the **-external=** and **-external_file=** options, or by using the `externally_visible` GNU attribute. For more information about the use of this `#pragma`, see "Wholeprogram Optimizations" on page 310.

`#pragma ghs startnocoverage`

`#pragma ghs endnocoverage`

These directives mark the beginning and end of a section of code such that any function defined in this section will not be instrumented to collect block coverage information.

For more information about coverage profiling, see "Enabling Instrumented Coverage or Performance Profiling" on page 58.

`#pragma ghs startnoinline`

`#pragma ghs endnoinline`

These directives mark the beginning and end of a section of code such that any function defined in this section will not be considered for inlining. Even inlined C++ member functions or functions marked with the keywords `__inline` or `inline` will not be inlined. Template functions and member functions of template classes are not affected.

For more information about controlling inlining, see "Inlining Optimizations" on page 294.

`#pragma ghs startnomisra`
 	
`#pragma ghs endnomisra`
 	
These directives mark the beginning and end of a section of code in which the compiler does not perform MISRA checking.
 	
For more information, see "MISRA C 2004" on page 168.

`#pragma ghs startnostrongfptr`
 	
`#pragma ghs endnostrongfptr`
 	
These directives mark the beginning and end of a section of code in which the compiler does not analyze conversions to and from `strong_fptr` types. When using these directives, you must make sure the call graph properly reflects the behavior of the code in order for **gstack** to produce correct results.
 	
For more information, see "Working With Recursive Clusters" on page 565.

`#pragma ghs static_call` *routine* Specifies that the encompassing function (or the next function to be defined, if used in the file scope), calls *routine*. If the caller is defined inside a `#pragma asm` block within a C file, a declaration for the function should be provided and the directive should be specified immediately before the declaration. Specify `0` instead of *routine* to indicate that no additional calls are made. This directive provides **gstack** with information that is necessary to accurately compute the maximum stack usage for the program. For more information, see "Annotating C Functions for gstack" on page 563 and "Annotating Assembly Functions for gstack" on page 562.

`#pragma ghs struct_min_alignment(`*n*`)`
 	
`#pragma ghs struct_min_alignment()`
 	
Instructs the compiler to lay out subsequent `class`, `struct`, and `union` definitions in memory with a minimum alignment of *n* bytes.
 	
The second format, where *n* is not specified, reverts the minimum alignment to the default value of 1.
 	
The `#pragma` should not be active before including headers, as the structures referenced by the header need to be consistent between the module including the header and any other modules on which the header depends.
 	
The **C/C++ Compiler→Alignment and Packing→Packing (Maximum Structure Alignment)** (`-pack=`*n*) and `#pragma pack(`*n*`)` directive override this pragma directive.

Conditional Pragma Directives

This feature is deprecated and may be removed in future versions of MULTI.

Any of the Green Hills extension `#pragma` directives can be made conditional upon whether you are generating debugging information or are optimizing your executable. To make a directive conditional, use the syntax:

```
#pragma ghs [condition] pragma-name
```

where the optional *condition* parameter is one of:

- `ifdebug` — Enable the `#pragma` only if debugging information is being generated (see **Debugging→Debugging Level** in "Debugging Options" on page 156).
- `ifnodebug` — Enable the `#pragma` only if debugging information is not being generated.
- `ifoptimize` — Enable the `#pragma` only if optimizations are enabled (see **Optimization→Optimization Strategy** in "Optimization Options" on page 145).
- `ifnooptimize` — Enable the `#pragma` only if optimizations are not enabled.

For example, to enable the **Assignment Bounds** and **Unallocated Memory** run-time checks only if optimizations are not enabled, enter `#pragma ghs check=(check-list)` as follows:

```
#pragma ghs ifnooptimize check=(assign,memory)
```

_Pragma Operator

The `_Pragma` preprocessing operator from C99 can be used in all language modes. It is useful in cases when the effect of a directive is desired within a macro.

For example, the following can be used to define a macro `DECL_ALIGN_BUF`:

```
#define PRAGMA(x) _Pragma(#x)
#define ALIGNVAR(align) PRAGMA( alignvar(align) )
#define CHANGE_SEC(sec, name) PRAGMA( ghs section sec = name )
/* Define a buffer in section "sect" with the given name, size, and
 * alignment
 */
#define DECL_ALIGN_BUF(name, size, align, sect) \
ALIGNVAR(align)                                 \
CHANGE_SEC(bss, sect)                           \
char name[size];                                \
CHANGE_SEC(bss, default)
```

The macro can be invoked as:

```
DECL_ALIGN_BUF(mybuf, 80, 8, ".mybss")
```

This macro invocation has the effect of:

```
#pragma alignvar(8)
#pragma ghs section bss=".mybss"
char mybuf[80];
#pragma ghs section bss=default
```

Intrinsic Functions

Intrinsic functions perform certain tasks which are difficult or inefficient to write in a high-level language.

Intrinsics are only available on processors that support the underlying instructions. Consult your processor's reference manual to determine which instructions are available.

The name of a C or C++ intrinsic function begins with two underscores (__). It usually generates optimized inline code, often using special instructions that do not correspond to standard C and C++ operations.

Prototypes for the functions can be found in ***install_dir*/include/v800/v800_ghs.h**, and can be included as follows:

```
#include <v800_ghs.h>
```

`void *__builtin_return_address(unsigned int level);` Returns the return-address of the current function. The *level* argument must be set to `0`. This function cannot be used in interrupt procedures.

`void __DI(void);`
`void __EI(void);`
These functions disable and enable interrupts and can bracket a critical section of code that must not be interrupted.

`void __RIR(unsigned int key);` Takes a *key* previously returned by `__DIR()` or `__EIR()` and restores interrupts to the state represented by that key. This function, together with `__DIR()` and `__EIR()` can be used to safely enable or disable interrupts and later restore them to their original state when that state is not known at compile time.

`unsigned int __DIR(void);`
`unsigned int __EIR(void);`
Like `__DI()` and `__EI()`, these functions disable and enable interrupts. In addition, they return an `unsigned int` key that represents the state of interrupts prior to their operation. For more information, see the documentation for `__DI()` and `__EI()`.

```
unsigned int __GETSR(void);
```

```
void __SETSR(unsigned int val);
```

The `__GETSR` function returns the current value of the *Program Status Word* (`PSW`).

The `__SETSR` function takes a 32-bit integer argument and stores it into this register.

If you are using a V850E2R or later, take caution when using these intrinsics because they do not ensure that BSEL is set to any particular system register bank.

`unsigned int __STSR(unsigned int, unsigned int);` (Only on the RH850 and later processors)

`void __LDSR(unsigned int, unsigned int, unsigned int);` (Only on the RH850 and later processors)

`unsigned int __STSR(unsigned int);` (Only for processors pre-dating the RH850)

`void __LDSR(unsigned int, unsigned int);` (Only for processors pre-dating the RH850)

The `__STSR()` function returns the current value of the system register corresponding to the number specified by its argument, which must be a constant immediate. On the RH850 and later processors it also takes a second argument which specifies the register group number, which must be a constant immediate.

The `__LDSR()` function takes a 32-bit integer argument (as its last argument) and stores it into the system register corresponding to the number specified by its first argument, which must be a constant immediate. On the RH850 and later processors it also takes an additional second argument which specifies the register group number, which must be a constant immediate.

```
int __CAXI(int *, int, int);
```

The `__CAXI()` function maps to the V850E2R and later's `CAXI` instruction, which performs a compare and exchange for interlock. 32-bits of data are loaded from the memory pointed to by the first argument to the function, and is compared to the 32-bits of data provided in the second argument. If the two values are equal, the third argument is stored back to the memory pointed to by the first argument. Regardless of the result of the compare, the original contents of the memory pointed to by the first argument are returned. This intrinsic does not provide a way to determine the outcome of the comparison.

Arithmetic Operation Instructions

```
unsigned int __MULUH(unsigned int a, unsigned int b);
```

```
signed int __MULSH(signed int a, signed int b);
```

The `__MULUH(a,b)` function takes as arguments two 32-bit unsigned integers *a* and *b* and returns the high 32-bit half of their 64-bit unsigned product.

The `__MULSH(a,b)` function takes as arguments two 32-bit signed integers *a* and *b* and returns the high 32-bit half of their 64-bit signed product.

```
int __ADD_SAT(int a, int b);

int __SUB_SAT(int a, int b);
```

The `__ADD_SAT(a,b)` function takes as arguments two 32-bit integers *a* and *b* and returns the sum (`a+b`), saturated as defined by the underlying microprocessor architecture.

The `__SUB_SAT(a,b)` function takes as arguments two 32-bit integers *a* and *b* and returns the difference (`a-b`), saturated as defined by the underlying microprocessor architecture.

```
int __SCH0R(int);

int __SCH1R(int);

int __SCH0L(int);

int __SCH1L(int);
```

These functions return the position of the first zero (`__SCH0R()`, `__SCH0L()`) or one (`__SCH1R()`, `__SCH1L()`) encountered in their 32-bit integer argument when searching from either from right-to-left (`__SCH0R()`, `__SCH1R()`) or left-to-right (`__SCH0L()`, `__SCH1L()`) respectively. If no such value is found, zero is returned. If the first bit examined matches, then 1 is returned, with 2 returned for the second bit, and so forth. These intrinsics are only available with the V850E2 and later processors.

The following example demonstrates their use:

```
#include "v800_ghs.h"
int x;
void foo(void)
{
    x = __SCH0L(0x80000000); /* returns 2 */
    x = __SCH0L(0xFFFFFFFE); /* returns 32 */
    x = __SCH1R(0x80000000); /* returns 32 */
    x = __SCH1R(0xFFFFFFFE); /* returns 2 */
    x = __SCH0R(0xFFFFFFFF); /* returns 0 */
}
```

Built-In Intrinsics

The following intrinsics do not require a prototype or header file:

`__builtin_constant_p(`*arg*`)` Evaluates to true if the argument is recognized by the compiler as representing a known constant value. *arg* may be an integer or string constant; complex expressions may or may not be considered constant, depending on optimization settings. This intrinsic may help you optimize away unused cases when a known constant expression can be handled differently depending on its value. Unlike most other **gcc** syntax, it does not require you to pass the **-gcc** mode option.
`__ghs_noprofile_func()` Acts like an empty function that is always inlined and has the `ghs_noprofile` attribute. This intrinsic does not affect generated code, but causes MULTI to exclude code paths containing calls to it from coverage calculations.

Static Assertions

Static assertions provide a mechanism to perform assertions that are evaluated and reported at compile time rather than run time. Static assertions have no effect if their tested condition evaluates to a non-zero value. If the condition evaluates to zero, an error is displayed along with the string literal passed as the second argument. The syntax for static assertions is:

```
_Static_assert( condition, message )
static_assert( condition, message )
```

where *condition* must be an integer constant expression that can be evaluated at compile time, and *message* must be a string literal. For example:

```
static_assert(1==0, "Assertion Failed");
_Static_assert(1==0, "Assertion Failed");
```

The `_Static_assert` keyword is recognized in both `C` and `C++`, but results in an error in strict dialects (**--ANSI**, `--C99`, and `--STD`).

In C, `static_assert` is not a keyword, but there is a macro definition in **assert.h** that is preprocessed to `_Static_assert` in permissive C dialects. This macro is not defined in strict C dialects.

In C++, `static_assert` is recognized as a keyword in permissive C++ dialects, and can be used in the same way as `_Static_assert`. It is not a keyword in strict C++ dialects.

Chapter 16

Libraries and Header Files

Contents

This chapter describes the standard language libraries provided with the Green Hills tools for V850 and RH850. Libraries are files which contain collections of object files.

Green Hills provides libraries to implement language features, such as the Standard C libraries, the C++ libraries, and various aspects of run-time support. These libraries, and any libraries that you create, are linked into the program image by the linker, **elxr**.

Header files provide declarations and macros that may be used while programming. Many of the functions in the standard libraries are declared in header files. In C and C++, header files are referenced in the source code by using a `#include` directive, which is read by the compiler's preprocessor.

By convention, libraries have a **.a** extension, and are named in the form **lib*name*.a**. C language header files generally have a **.h** extension, and C++ headers either have a **.h** extension or no extension.

This chapter documents Green Hills libraries. For information about creating and maintaining your own libraries, see Chapter 10, “The ax Librarian” on page 477.

For more information, see:

- P. J. Plauger, *The Standard C Library* (describes the C89 library)
- Harbison & Steele, *C: A Reference Manual (5th Ed.)* (describes the C99 library)
- B. Stroustrup, *The C++ Programming Language (3rd Ed.)*
- P. J. Plauger, *Dinkum C++ Library Reference* (included with your distribution, at ***install_dir*/scxx/html/index.html**)
- P. J. Plauger, *Dinkum Abridged Library Reference* (for EC++ and EEC++; included with your distribution, at ***install_dir*/ecxx/html/index.html**).

Note

Your Green Hills compiler license permits you to make unlimited distributions of programs linked with the Green Hills library object code without charge. Distribution of Green Hills library source code or object code is not permitted.

The Green Hills Header Files and Standard Libraries

Your Green Hills tools for V850 and RH850 provide a number of directories that contain header files and standard libraries for the C and C++ languages, together with a number of other low-level system libraries.

C and C++ Header File Directories

Depending upon the language you are compiling, the driver selects one or more of the following directories for header files which are specified in `#include` preprocessor directives:

- **include/v800** — Contains header files for V850 and RH850 extensions for C and C++.
- **ansi** — Contains Standard C library header files.
- **scxx** — Contains Standard C++ library header files.
- **eecxx** — Contains Extended Embedded C++ library header files.
- **ecxx** — Contains Embedded C++ library header files.

C and C++ Header Files

Some of the files in the **ansi** header file directory are for internal use only. These internal header files should not be included in your code directly, because future releases may eliminate, reorganize, or rename them. If you include any of them directly, a fatal error explains which header file should be included in your code instead:

```
Please include <string.h> instead of ghs_str.h
Please include <string.h> instead of ghs_mem.h
Please include <stddef.h> instead of ghs_null.h
Please include <stdarg.h> instead of ghs_valist.h
Please include <stddef.h> instead of ghs_wchar.h
```

In the header file directories for C++ (**scxx**, **eecxx**, and **ecxx**), files that begin with `x` or `y` are internal header files that should not be included in your code directly.

Library Directories

Depending upon the target architecture that you specify and whether or not you are using INTEGRITY, the driver selects one of the several directories (inside ***install_dir*/lib**) for libraries to resolve references. Each directory contains a library set optimized for use with a specific type of target CPU in the V850 and RH850 processor family.

Libraries

Each library directory contains a version of the libraries listed in this section, built for the particular target architecture. Library names may contain one or more suffixes that begin with an underscore (_), depending on the options you pass to the driver:

- All C++ libraries have the suffix `_xvtbl` when **--large_vtbl_offsets** is used.
- The Standard C++ library and `wchar_t` libraries have the suffix `_u16` or `_s32` depending on the setting of **-wchar_u16** or **-wchar_s32** (except on INTEGRITY).
- The Standard C++ library, **libmath**, and **libind** have a floating-point suffix depending on the floating-point mode:

-fhard	`_fp`	Hardware floating point
-fsoft	`_sf`	Software floating point
-fsingle	`_sd`	Hardware single, software double
-floatsingle	`_sngl`	Hardware single used for all floating point types

C++ libraries are searched before the C libraries if there is a C++ source file on the driver command line, if you use the C++ driver (**cxv850**), or if you specify **-language=cxx** (see also "Specifying C++ Libraries" on page 691). They are searched in the following order:

- One of the following C++ libraries is used, depending on selected options:
  - **libscnoe** — Standard C++ Library without exceptions.
  - **libsce** — Standard C++ Library with exceptions.
  - **libeecnoe** — Extended embedded C++ Library without exceptions.
  - **libeece** — Extended embedded C++ Library with exceptions.

  - **libecnoe** — Embedded C++ Library without exceptions. This library is a subset of **libeecnoe**.
  - **libece** — Embedded C++ Library with exceptions.
- One of the following run-time support libraries is used, depending on selected options:
  - **libsedgnoe** — Run-time support for standard C++, without exceptions.
  - **libsedge** — Run-time support for standard C++, with exceptions.
  - **libedgnoe** — Run-time support for embedded and extended embedded C++, without exceptions.
  - **libedge** — Run-time support for embedded and extended embedded C++, with exceptions.

Standard C Libraries are searched in the following order (unless you pass the **-nostdlib** driver option):

- **libwchar*** — Standard C library functions that use `wchar_t`.
- **libnoflt** — Provides versions of `printf`, `scanf`, and other functions, which exclude floating point support, and so are smaller than those in **libansi**. This library is only used if the deprecated **-nofloatio** driver option is specified.
- **libfmalloc** — Provides versions of `malloc`, `realloc`, and `free` that are optimized for small blocks. This library is searched before **libansi**, and is used unless the **-no_fast_malloc** driver option is specified. For more information, see “Use Small Block Malloc” on page 131.
- **libdbmem** — Provides a version of `malloc` with memory-checking. This library is selected when you pass either the **-check=alloc** or **-check=memory** driver options. This library is searched in place of **libfmalloc**.
- **libansi** — Standard C library. This is the primary C library.
- **libutf8**, **lib8bit**, **libwc**— Low-level C library functions that recognize `wchar_t` encodings. Only one of these libraries is used, depending on the value of the **-kanji=** option:
  - **-kanji=utf8** — **libutf8**
  - **-kanji=none** — **lib8bit**
  - otherwise — **libwc***

  This library is searched after **libansi**.

- **libmath** — Standard Math library. This library is searched after **libutf8**, **lib8bit**, or **libwc**.

* On INTEGRITY, **libwchar** is contained in **libansi** and **libwc** is contained in **libind**.

Low-level System Libraries are searched in the following order (unless you pass the **-nostdlib** driver option):

- **libind** — Language-independent library, containing support routines for features such as software floating point, run-time error checking, C99 complex numbers, and some general purpose routines of the ANSI C library.
- **libstartup** — Run-time environment startup routines. The source code for the modules in this library is provided in the **src/libstartup** directory (see "Low-Level Startup Library libstartup.a" on page 817).
- **libsys** — Run-time environment system routines. The source code for the modules in this library is provided in the **src/libsys** directory (see "Low-Level System Library libsys.a" on page 818).
- **libarch** — Target-specific run-time support. Any file produced by the Green Hills Compiler may depend on symbols in this library.

Note

Some functions in low-level system libraries are for use only by the compiler or linker, and may not follow standard calling conventions. The compiler and linker take this into account when generating or modifying calls to these functions.

Other Libraries:

- **libmulti** — Supports procedure calls from the MULTI Debugger command line. This library is selected when you pass the **-G** driver option.

Advanced Library Topics

Less Buffered I/O

The Green Hills ANSI C library includes a *Standard I/O Package* ("**stdio**"), which provides formatted I/O using `printf()` and `scanf`, character I/O using `getc()` and `putc()`, and other features.

If I/O is completely *unbuffered*, every character read or written requires a system call. This unbuffered I/O is slow, but it ensures that I/O is not delayed until the buffer is full or lost by being left in a buffer if the application exits abnormally.

As a consequence, in traditional implementations, most I/O performed in **stdio** is *fully buffered*. This improves performance by increasing the average number of characters written per system call, but requires additional space for the code to manage the buffers, as well as for the buffers themselves. Often, several kilobytes of code are added to the application, because the **stdio** package invokes `malloc()` and other routines to manage the buffering.

In order to provide a reasonable compromise between fully buffered and unbuffered I/O, the Green Hills C library defaults to a *less buffered I/O* mode. Rather than allocate a buffer for each file via `malloc()` (which is used as long as the file is open), buffering is performed within `fprintf()`, `fwrite()`, `fputs()`, `puts()`, `printf()`, `vprints()`, `vfprintf()`, and their wide character forms. Because of this approach, none of the I/O functions call `malloc()`, possibly reducing the size of the overall program by a few kilobytes.

During a single call, any characters written to one of these functions are buffered in a buffer on the stack. One function call often requires only one output system call. After the function completes, all characters are written to the file. No characters are left in a buffer after the function call, eliminating both the risk of output loss and the need to flush buffers when a file is closed or the program exits. No input routines other than `fread()` are buffered by the less buffered I/O method. Files are not closed upon program termination, except as noted below.

You can enable full buffering of any file by calling either `setbuf()` with a buffer of at least `BUFSIZ` bytes or `setvbuf()` with a buffer of at least *n* bytes, where *n* is the fourth parameter to `setvbuf()`. For example:

```
char buffer[BUFSIZ];
setbuf(f, buffer);

char buffer[8196];
setvbuf(f, buffer, _IOFBF, sizeof(buffer));
```

In this case, characters are only written to the file when the buffer is full or when `fflush()` or `fclose()` is called. Upon normal termination of the program, if `setbuf()` or `setvbuf()` has been called at any time, all open files will be flushed and then closed.

You can enable line buffering by calling `setvbuf()` with `_IOLBF` as the third parameter. Line buffering is the same as full buffering with the exception that `fflush()` is called at the end of `fputs()` and all forms of `printf()`. For example:

```
char buffer[256];
setvbuf(f, buffer, _IOLBF, sizeof(buffer));
```

If the second parameter to `setvbuf()` is `NULL`, all buffering is disabled and the third and fourth parameters are ignored.

All files are processed in exactly the same manner, whether they are opened by default (**stdin**, **stdout**, and **stderr**) or by `fopen()`.

Buffering of C++ I/O Streams

The C++ I/O streams are implemented on top of the C **stdio** files. When you use C++ I/O streams, the constructors for these streams enable full buffering for files, and line buffering for `stdout` and `stderr`.

To disable this buffering, call `pubsetbuf()`, `setbuf()`, or `setvbuf()` before outputting anything onto either of the I/O streams. For example:

```
cout.rdbuf()->pubsetbuf(0, 0);
```

or

```
setbuf(stdout, NULL);
```

If you call one of these functions after outputting anything onto an I/O stream, buffering will remain unchanged, in accordance with the C specification for `setvbuf()`.

64-bit Integer Arguments and printf

The type `long long` was an extension to ANSI C which has become official in the ISO C99 standard. Since this type is larger than either `int` or `long`, it requires special handling in the `printf` and `scanf` routines.

As in earlier versions of `printf` and `scanf`, a single '`l`' modifier indicates an argument of type `long`. For example:

```
printf("large number: %ld", 0x7fffffffL);
```

However, when the '`l`' modifier is repeated, an argument of type `long long` is indicated. For example:

```
printf("very large number: %lld", 0x7fffffffffffffffLL);
```

For backwards compatibility with older implementations, `printf` and `scanf` also allow the '`L`' modifier to indicate an argument of type `long long`, but this is non-standard and is not recommended.

Memory Allocation and malloc()

`malloc()`, `calloc()`, `free()`, and the C++ `new` and `delete` operators are all examples of dynamic memory allocation. Source code for the Green Hills implementation of dynamic memory allocation is not provided. Allocation requests are limited to `LONG_MAX - 127` bytes; larger requests may fail, even if there is enough available memory.

Before debugging memory allocation issues, ensure that your implementation of **ind_heap.c** is appropriate for your system. Then, check that your code does not do any of the following:

- write before the beginning or after the end of a piece of dynamic memory
- access memory that has been deallocated by calling `free()` or the C++ `delete` operator
- deallocate memory more than once
- deallocate memory that was not dynamically allocated (global, static, or local memory).

For help debugging dynamic memory allocation issues, see “Enabling Run-Time Memory Checking” on page 65 and Chapter 5, “The DoubleCheck Source Analysis Tool” on page 333.

Using Thread-Safe Library Functions

A *thread-safe* or *reentrant* function executes correctly in a multithreaded environment, even if the function is called by multiple threads concurrently or if it is interrupted or suspended and then resumed. A *thread* is a single task running in a multitasking environment in which several tasks share the same memory space.

Some Green Hills library functions are not inherently thread-safe. The libraries provide four mechanisms for an application to achieve thread-safe behavior. The following sections describe the four mechanisms and, where necessary, explain how to port them to a multithreaded environment.

- “Using a Lock to Ensure Critical Code Is Thread-Safe” on page 777
- “Using File Locks to Ensure I/O Is Thread-Safe” on page 778
- “Allocating Space in Each Thread for Static Data” on page 779
- “Using Alternate Thread-Safe Library Functions” on page 781

In some cases, the library provides more than one way to achieve thread-safe behavior. For example, you can make calls to `strtok()` thread-safe either by allocating space in each thread for the static data it uses, or by changing the code to use the inherently thread-safe `strtok_r()` function. The appropriate choice is left up to you.

For applications that require maximum I/O performance, Green Hills provides versions of some I/O library functions that are not thread-safe. These functions have names ending in `_unlocked`, such as `putc_unlocked()` and `getc_unlocked()`. Applications using these functions must guarantee that no two threads concurrently access the same file.

Porting to a new multithreaded environment involves customizing the following functions, located in **src/libsys/ind_lock.c**, **src/libsys/ind_errn.c**, and **src/libsys/ind_thrd.c**.

`void __ghsLock(void);` Acquires global lock. Blocks until the lock becomes available.
`void __ghsUnlock(void);` Releases global lock.
`void __gh_lock_init(void);` Initializes the global lock data structure before it is used.
`void __ghs_flock_file(void *addr);` Acquires per-file lock ***addr***. Blocks until the lock becomes available.
`void __ghs_funlock_file(void *addr);` Releases per-file lock ***addr***.
`int __ghs_ftrylock_file(void *addr);` Attempts to acquire per-file lock ***addr***. Returns `0` on success and nonzero if the lock could not be acquired. May return `-1` if this behavior cannot be implemented. This function must not block.

`void __ghs_flock_create(void **addr);`
`void __ghs_flock_destroy(void *addr);`
Initializes and cleans up per-file lock data structure. The `__ghs_flock_create()` implementation may store a pointer in *addr*. That pointer is passed to the other per-file lock calls.

`void *__ghs_GetThreadLocalStorageItem(int specifier);` Retrieves pointer to specified per-thread data item. May return `NULL` if the data item is not allocated for each thread.

For information about the default implementation of per-thread storage and locks in Green Hills supported operating systems, see the appropriate RTOS User's Guide (*INTEGRITY Libraries and Utilities User's Guide* or *u-velOSity User's Guide*).

Using a Lock to Ensure Critical Code Is Thread-Safe

Code that accesses resources such as global data structures, hardware devices, or memory shared between two or more threads is *critical code*. These shared resources may become corrupted when two or more threads write to the same data structures simultaneously, or when one thread reads data while another thread is writing to that same data.

To make critical code thread-safe, you must implement a global, recursive lock mechanism that allows a function to completely execute critical code before that code is accessed by another thread. Green Hills library functions call `__ghsLock()` before critical code and `__ghsUnlock()` after critical code. The `__ghsLock()` function must acquire the global lock or block the calling thread if any other thread has already acquired the global lock. The `__ghsUnlock()` function releases the global lock. These functions operate recursively. If `__ghsLock()` is called *n* times, the lock is released only after `__ghsUnlock()` is called *n* times.

Using File Locks to Ensure I/O Is Thread-Safe

You can also implement recursive locking for file I/O to ensure thread safety. If multiple threads are performing I/O on the same file concurrently, the file may become corrupted. This section discusses three possible implementations of file locking, from the simplest to the most flexible.

The simplest file lock implementation uses the global lock by having `__ghs_flock_file()` call `__ghsLock()` and `__ghs_funlock_file()` call `__ghsUnlock()`. However, this implementation may cause slow performance if one thread holds the global lock until it completes relatively slow I/O while other threads are attempting to use library functions that contain critical sections.

Another file lock implementation choice is to use a separate global lock for I/O. This avoids locking the entire library during I/O. However, performance concerns could still arise if other threads are attempting to perform I/O and must wait for the single global I/O lock.

The most flexible file lock implementation choice is to create a separate lock for each file. To implement a lock for each file, have `__ghs_flock_create()` initialize a lock, and use the argument to store a pointer to the lock data structure. The `__ghs_flock_file()` and `__ghs_funlock_file()` functions have the pointer passed to them, so they can locate the specific lock for each file. Finally, the `__ghs_flock_destroy()` function should clean up and deallocate the lock.

Allocating Space in Each Thread for Static Data

This section describes how you can make certain library functions thread-safe by allocating separate copies of the data structures they use in each thread. The data structures are then referred to as *per-thread data* or *thread local storage*.

The additional space needed to make all such library functions completely thread-safe is several hundred bytes per thread. Therefore, the decision of which, if any, of these data structures are allocated for each thread is left to the implementation. In many cases, a more practical solution would be to use alternate library functions that are thread-safe. For more information, see "Using Alternate Thread-Safe Library Functions" on page 781.

An implementation could allow some library functions to operate on global data. It could either restrict all such function calls to a single thread or allow the application to synchronize access by using a mutual exclusion mechanism around the function call and the use of the return value. For example, if two threads wanted to call `asctime()`, the function call and the use of the return value could be protected by first calling `__ghsLock()` and afterwards calling `__ghsUnlock()`.

The function `__ghs_GetThreadLocalStorageItem()` in **src/libsys/ind_thrd.c** returns a pointer to the specified per-thread data item, or `NULL`. Returning `NULL` causes the library to use a single, global data item. The default implementation for stand-alone programs returns `NULL` for all but one such request.

If you implement a per-thread `errno` in `__ghs_GetThreadLocalStorageItem()`, you must define the preprocessor symbol `USE_THREAD_LOCAL_ERRNO` in **src/libsys/ind_errn.c**.

The data item pointers `__ghs_GetThreadLocalStorageItem()` can return are as follows:

Specifier (Value)	Example Data Item Declaration
`__ghs_TLS_asctime_buff (0)`	`char asctime_buff[30]`
`__ghs_TLS_tmpnam_space (1)`	`char tmpnam_space[L_tmpnam]`
`__ghs_TLS_strtok_saved_pos (2)`	`char *strtok_saved_pos`
`__ghs_TLS_Errno (3)`	`int errno`
`__ghs_TLS_gmtime_temp (4)`	`struct tm gmtime_temp`
`__ghs_TLS___eh_globals (5)`	`void *__eh_globals`

Specifier (Value)	Example Data Item Declaration
`__ghs_TLS_SignalHandlers (6)`	`SignalHandler SignalHandlers[_SIGMAX]`

Upgrading Earlier GetThreadLocalStorage() Implementations

In versions of MULTI prior to 4.2.3, the function `GetThreadLocalStorage()`, declared in **ind_thrd.h**, was used to retrieve thread-specific or global library data structures. If you previously customized `GetThreadLocalStorage()` for your environment, the simplest way to upgrade is to add the following implementation of `__ghs_GetThreadLocalStorageItem()`.

```
enum __ghs_ThreadLocalStorage_specifier
{
     __ghs_TLS_asctime_buff,
     __ghs_TLS_tmpnam_space,
     __ghs_TLS_strtok_saved_pos,
     __ghs_TLS_Errno,
     __ghs_TLS_gmtime_temp,
     __ghs_TLS___eh_globals,
     __ghs_TLS_SignalHandlers
};

void *__ghs_GetThreadLocalStorageItem(int specifier)
{
     ThreadLocalStorage *tls = GetThreadLocalStorage();
     if(!tls)
          return (void *)0;
     switch (specifier)
     {
          case (int)__ghs_TLS_Errno:
               return (void *)&tls->Errno;
          case (int)__ghs_TLS_SignalHandlers
               return (void *)&tls->SignalHandlers;
          case (int)__ghs_TLS_asctime_buff:
               return (void *)&tls->asctime_buff;
          case (int)__ghs_TLS_tmpnam_space:
               return (void *)&tls->tmpnam_space;
          case (int)__ghs_TLS_strtok_saved_pos:
               return (void *)&tls->strtok_saved_pos;
          case (int)__ghs_TLS_gmtime_temp:
```

```
                return (void *)&tls->gmtime_temp;
            case (int)__ghs_TLS___eh_globals:
                return (void *)&tls->__eh_globals;
        }
        return (void *)0;
}
```

Using Alternate Thread-Safe Library Functions

Green Hills provides alternatives for some library functions that are not inherently thread-safe. These alternate, thread-safe functions have names ending in `_r`, such as `asctime_r()` and `rand_r()`. For example, you can use `rand_r()` to replace calls to `srand()` and `rand()`.

Customizing Thread-Safe Library Functions

Green Hills provides source code for some low-level functions in the **src/libsys** and **src/libstartup** directories, so you can customize them for your particular environment. When modified, these functions must remain thread-safe so that functions in the Green Hills libraries that call them also remain thread-safe. For a list of these functions, see the following section.

Green Hills Standard C Library Functions

The functions documented in the ISO C99 Standard are contained in the libraries listed in “Libraries” on page 770. You can find lists of these functions, as well as thread-safety information, in the following sections:

- “Standard C Functions in libansi.a” on page 783
- “Wide Character Functions” on page 797
- “libmath.a Functions” on page 798
- “libind.a Functions” on page 811
- “libstartup.a Functions” on page 811

Some standard function names are converted into special names by the linker based on options. For more information, see “Standard Function Names Converted by the Linker” on page 811.

The following code letters identify the level of thread-safety of the library functions listed in the remainder of this chapter.

Code	Thread Safety
`K`	Safe to call from all INTEGRITY contexts, and completely thread-safe.
`K*`	Safe to call from all INTEGRITY contexts, except when using floating-point arguments. Completely thread-safe.
`Y`	Completely thread-safe.
`P`	Partially thread-safe. You must implement a lock for complete thread safety.
`I`	Performs I/O or modifies I/O data structures. This is a special case of a partially thread-safe function.
`T`	Achieves thread safety by per-thread storage, so long as per-thread storage is implemented. See the preceding discussion on “Allocating Space in Each Thread for Static Data” on page 779
`E`	Writes to the global variable `errno`. Achieves thread safety by per-thread storage of `errno`.
`N`	Not thread-safe.

Note

Under INTEGRITY, all functions marked with a `Y`, `P`, `I`, `T`, or `E` are completely thread-safe. These functions are safe to call from within a

Task, however, they may not be safe to call from an interrupt context unless they are also marked with a `K`.

Under u-velOSity, the thread-safety of functions marked with a `P`, `I`, `T`, or `E` depends on the kernel libraries you are using. Please refer to the *u-velOSity User's Guide* for more information.

Standard C Functions in libansi.a

This section lists functions available in **libwchar_s32.a**, **libwchar_u16.a**, **libansi.a**, and the low-level functions in **libsys.a** on which these standard functions depend. The information is laid out in two tables:

- "libansi.a Standard C Functions and Dependencies" on page 783
- "libsys.a Function Implementation" on page 796

Depending on the operating system you are running on your target, and whether or not you are connecting to your target through a MULTI debug server, you may need to implement environment-specific behavior for the low-level functions in **libsys.a**. The **libsys.a** function implementation table describes how the Green Hills tools implement these functions by default.

libansi.a Standard C Functions and Dependencies

This table lists functions in **libansi.a**. The low-level functions in **libsys.a** on which each standard function depends are listed in the **Dependency** column. For information about whether or not you need to implement environment-specific behavior for a **libsys.a** function, see "libsys.a Function Implementation" on page 796 and "Low-Level System Library libsys.a" on page 818.

Function		Dependency	Thread Safety
`void`	`abort(void);`	`raise()`	`Y`
`int`	`abs(int x);`		`K`
`char`	`*asctime(const struct tm *timeptr);`	`__gh_timezone()`	`T`
`char`	`asctime_r(const struct tm *timeptr, char *buffer);`	`__gh_timezone()`	`Y`

Function		Dependency	Thread Safety
void	_assert(const char *problem, const char *filename, int line);	raise() write()	IE
void	assert(int value);	raise() write()	IE
int	atexit(void (*func)(void));		P
int	atoi(const char *nptr);		E
long	atol(const char *nptr);		E
long long	atoll(const char *nptr);		E
int	bcmp(char *b1, char *b2, int length);		K
void	bcopy(char *from, char *to, int n);		K
void	*bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));		K
wint_t	btowc(int c);		Y
void	bzero(char *pt, int n);		K
void	*calloc(size_t nmemb, size_t size);	sbrk()	P
void	cfree(char *item);	sbrk()	P
void	clearerr(FILE *stream);		I
clock_t	clock(void);	times()	Y
char	*ctime(const time_t *timer);	__gh_timezone() localtime()	T
char	*ctime_r(const time_t *timer, char *buffer);	__gh_timezone() localtime()	Y
div_t	div(int numer, int denom);		K
int	eprintf(const char *format, ...);	write()	IE
void	exit(int status);	_Exit()	P
int	fclose(FILE *stream);	close() write()	IE
FILE	*fdopen(int fno, const char *mode);	write() close()	IE
int	feof(FILE *stream);		Y
int	ferror(FILE *stream);		Y
int	fflush(FILE *stream);	write()	IE

Function		Dependency	Thread Safety
int	ffs(int i);		K
int	fgetc(FILE *stream);	read()	IE
int	fgetpos(FILE *restrict stream, fpos_t *restrict pos);	lseek()	IE
char	*fgets(char *restrict str, int n, FILE *restrict stream);	read()	IE
wint_t	fgetwc(FILE *stream);	read()	IE
wint_t	fgetwc_unlocked(FILE *stream);	read()	NE
wchar_t	*fgetws(wchar_t *restrict s, int n, FILE *restrict stream);	read()	IE
void	_filbuf(FILE *file);	read()	IE
int	fileno(FILE *stream)		Y
void	_flsbuf(int ch, FILE *file);	write()	IE
FILE	*fopen(const char *restrict filename, const char *restrict mode);	close() open() write()	IE
int	fprintf(FILE *restrict stream, const char *restrict format, ...);	write()	IE
int	fputc(int c, FILE *stream);	write()	IE
int	fputs(const char *restrict s, FILE *restrict stream);	write()	IE
wint_t	fputwc(wchar_t c, FILE *stream);	write()	IE
wint_t	fputwc_unlocked(wchar_t c, FILE *stream);	write()	NE
int	fputws(const wchar_t restrict *s, FILE *restrict stream);	write()	IE
size_t	fread(void restrict *ptr, size_t size, size_t nmemb, FILE *restrict stream);	read()	IE
void	free(void *ptr);	sbrk()	P
FILE	*freopen(const char *restrict filename, const char *restrict mode, FILE *restrict stream);	close() open() write()	IE

Function		Dependency	Thread Safety
int	fscanf(FILE *restrict stream, const char *restrict format, ...);	close() lseek() read() sbrk() write()	IE
int	fseek(FILE *stream, long offset, int whence);	lseek() write()	IE
int	fseeko(FILE *stream, off_t offset, int whence);	lseek() write()	IE
int	fsetpos(FILE *stream, const fpos_t *pos);	lseek() write()	IE
long	ftell(FILE *stream);	lseek()	IE
off_t	ftello(FILE *stream);	lseek()	IE
int	fwide(FILE *stream, int mode);		N
int	fwprintf(FILE *restrict stream, const wchar_t *restrict format, ...);	write()	IE
size_t	fwrite(const void *restrict ptr, size_t size, size_t nmemb, FILE *stream);	write()	IE
int	fwscanf(FILE *restrict stream, const wchar_t *restrict format, ...);		IE
int	getc(FILE *stream);	read()	IE
int	getc_unlocked(FILE *stream);	read()	NE
int	getchar(void);	read()	IE
int	getchar_unlocked(void);	read()	NE
char	*getenv(const char *name);	environ	P
char	*gets(char *s);	read()	IE
int	getw(FILE *stream);	read()	IE
wint_t	getwchar(void);	read()	IE
intmax_t	imaxabs(intmax_t, intmax_t)		Y
imaxdiv_t	imaxdiv(intmax_t, intmax_t)		Y
char	*index(const char *str, const char ch);		K
int	isalnum(int c);		K
int	isalpha(int c);		K

Function	Dependency	Thread Safety
int isblank(int c);		K
int iscntrl(int c);		K
int isdigit(int c);		K
int isgraph(int c);		K
int islower(int c);		K
int isprint(int c);		K
int ispunct(int c);		K
int isspace(int c);		K
int isupper(int c);		K
int iswalnum(wint_t wc);		K
int iswalpha(wint_t wc);		K
int iswblank(wint_t wc);		K
int iswcntrl(wint_t wc);		K
int iswctype(wint_t wc, wctype_t desc);		K
int iswdigit(wint_t wc);		K
int iswgraph(wint_t wc);		K
int iswlower(wint_t wc);		K
int iswprint(wint_t wc);		K
int iswpunct(wint_t wc);		K
int iswspace(wint_t wc);		K
int iswupper(wint_t wc);		K
int iswxdigit(wint_t wc);		K
int isxdigit(int c);		K
long labs(long j);		K
ldiv_t ldiv(long numer, long denom);		K
long long llabs(long long j);		K
lldiv_t lldiv(long long numer, long long denom);		K

Function		Dependency	Thread Safety
struct lconv	*localeconv(void);		Y
void	*malloc(size_t size);	sbrk()	P
int	mblen(const char *s, size_t n);		Y
size_t	mbrlen(const char *restrict s, size_t n, mbstate_t *restrict ps);		E
size_t	mbrtowc(wchar_t *restrict pwc, const char *restrict s, size_t n, mbstate_t *restrict ps);		E
int	mbsinit(const mbstate_t *ps);		Y
size_t	mbsrtowcs(wchar_t *restrict dst, const char **restrict src, size_t len, mbstate_t *restrict ps);		Y
size_t	mbstowcs(wchar_t *restrict pwcs, const char *restrict s, size_t n);		Y
void	*memchr(const void *s, int c, size_t n);		K
int	memcmp(const void *s1, const void *s2, size_t n);		K
void	*memrchr(const void *s, int c, size_t n);		K
char	*mktemp(char *str);	getpid()	P
time_t	mktime(struct tm *timeptr);	__gh_timezone() localtime() localtime_r()	Y
void	perror(const char *s);	write()	IE
int	printf(const char restrict *format, ...);	lseek() write()	IE
int	putc(int c, FILE *stream);	write()	IE
int	putc_unlocked(int c, FILE *stream);	write()	NE
int	putchar(int c);	write()	IE
int	putchar_unlocked(int c);	write()	NE
int	putenv(const char *c, int n);		PE

Function		Dependency	Thread Safety
int	puts(const char *s);	write()	IE
int	putw(int w, FILE *stream);	write()	IE
wint_t	putwc(wchar_t wc, FILE *stream);	write()	IE
wint_t	putwchar(wchar_t wc);	write()	IE
void	qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));		K
int	rand(void);		P
int	rand_r(long *seedSent);		K
int	rand_r2(long seed[2])		K
void	*realloc(void *ptr, size_t size);	sbrk()	P
int	remove(const char *filename);	unlink()	Y
void	rewind(FILE *stream);	lseek() write()	IE
char	*rindex(const char *str, const char ch);		K
int	scanf(const char *restrict format, ...);	close() read() sbrk() write()	IE
void	setbuf(FILE *restrict stream, char *restrict buf);	close() write()	IE
int	setenv(const char *envname, const char *envval, int overwrite);		PE
int	setlinebuf(FILE *stream);		N
char	*setlocale(int category, const char *locale);		Y
int	setvbuf(FILE *restrict stream, char *restrict buf, int mode, size_t size);	close() write()	IE
int	snprintf(char *restrict s, size_t n, cosnt char *restrict format, ...);		K*
int	sprintf(char * restrict s, const char *restrict format, ...);		K*
void	srand(unsigned int seed);		N

Function		Dependency	Thread Safety
int	sscanf(const char *restrict s, const char *restrict format, ...);		K*
int	strcasecmp(const char *s1, const char *s2); Ignore the difference between uppercase and lowercase		K
char	*strcat(char *restrict s1, const char *restrict s2);		K
char	*strchr(const char *s, int c);		K
int	strcmp(const char *s1, const char *s2);		K
int	strcoll(const char *s1, const char *s2);		K
char	*strcpy(char *restrict s1, const char *restrict s2);		K
size_t	strcspn(const char *s1, const char *s2);		K
char	*strdup(const char *s)		P
char	*strerror(int errnum);		K
size_t	strftime(char *restrict s, size_t maxsize, const char *restrict format, const struct tm *restrict timeptr);	__gh_timezone()	Y
size_t	strlen(const char *s);		K
int	strncasecmp(const char *s1, const char *s2, size_t n); Ignore the difference between uppercase and lowercase		K
char	*strncat(char *restrict s1, const char *restrict s2, size_t n);		K
int	strncmp(const char *s1, const char *s2, size_t n);		K
char	*strncpy(char *restrict s1, const char *restrict s2, size_t n);		K
char	*strndup(const char *s, size_t n)	sbrk()	P

Function		Dependency	Thread Safety
size_t	strnlen(const char *s, size_t n)		K
char	*strpbrk(const char *s1, const char *s2);		K
char	*strrchr(const char *s, int c);		K
char	*strsave(char *str);	sbrk()	P
size_t	strspn(const char *s1, const char *s2);		K
char	*strstr(const char *s1, const char *s2);		K
intmax_t	strtoimax(const char * restrict, char **restrict, int)		E
char	*strtok(char *restrict s1, const char *restrict s2);		T
char	*strtok_r(char *s1, const char *s2, char **ppLast);		K
long long	strtoll(const char * restrict, const char **restrict, int)		E
long	strtol(const char *restrict nptr, char **restrict endptr, int base);		E
unsigned long	strtoul(const char *restrict nptr, char **restrict endptr, int base);		E
unsigned long long	strtoull(const char * restrict, const char **restrict, int)		E
uintmax_t	strtoumax(const char * restrict, char **restrict, int)		E
size_t	strxfrm(char *restrict s1, const char *restrict s2, size_t n);		K
void	swab(char *from, char *to, int nbytes);		K
int	swprintf(wchar_t *restrict s, size_t n, const wchar_t *restrict format, ...);		Y
int	swscanf(const wchar_t *restrict s, const wchar_t *restrict format, ...);		Y

Function	Dependency	Thread Safety
FILE *tmpfile(void);	close() open() unlink() write()	IE
char *tmpnam(char *s);	access()	TPE
char *tmpnam_r(char *s);	access()	PE
int tolower(int c);		K
int toupper(int c);		K
wint_t towctrans(wint_t wc, wctrans_t desc);		Y
wint_t towlower(wint_t wc);		Y
wint_t towupper(wint_t wc);		Y
int ungetc(int c, FILE *stream);	close() sbrk() write()	IE
wint_t ungetwc(wint_t wc, FILE *stream);	close() sbrk() write()	IE
wint_t ungetwc_unlocked(wint_t wc, FILE *stream);	close() sbrk() write()	NE
int unsetenv(const char* envname)		P
int vfprintf(FILE *restrict stream, const char *restrict format, va_list arg);	write()	IE
int vfscanf(FILE *restrict stream, const char *restrict format, va_list arg);	close() read() sbrk() write()	IE
int vfwprintf(FILE *restrict stream, const wchar_t *restrict format, va_list arg);	write()	IE
int vfwscanf(FILE *restrict stream, const wchar_t *restrict format, va_list arg);	close() read() sbrk() write()	IE
int vprintf(const char *restrict format, va_list arg);	write()	IE
int vscanf(const char *restrict format, va_list arg);	close() read() sbrk() write()	IE
int vsnprintf(char *restrict s, size_t n, const char *restrict format, va_list arg);		K*
int vsprintf(char *restrict s, const char *restrict format, va_list arg);		K*

Function		Dependency	Thread Safety
int	vsscanf(const char *restrict s, const char *restrict format, va_list arg);		K*
int	vswprintf(wchar_t *restrict s, const wchar_t *restrict format, va_list arg);		Y
int	vswscanf(const wchar_t *restrict s, const wchar_t *restrict format, va_list arg);		P
int	vwprintf(const wchar_t *restrict format, va_list arg);	write()	IE
int	vwscanf(const wchar_t *restrict format, va_list arg);	close() read() sbrk() write()	IE
size_t	wcrtomb(char *restrict s, wchar_t wc, mbstate_t *restrict ps);		Y
wchar_t	*wcscat(wchar_t *restrict s1, const wchar_t *restrict s2);		Y
wchar_t	*wcschr(const wchar_t *s, wchar_t wc);		Y
int	wcscmp(const wchar_t *s1, const wchar_t *s2);		Y
int	wcscoll(const wchar_t *s1, const wchar_t *s2);		Y
wchar_t	*wcscpy(wchar_t *restrict s1, const wchar_t *restrict s2);		Y
size_t	wcscspn(const wchar_t *s1, const wchar_t *s2);		Y
size_t	wcsftime(wchar_t *restrict s, size_t maxsize, const wchar_t *restrict format, const struct tm *restrict timeptr);	__gh_timezone()	Y
size_t	wcslen(const wchar_t *s);		Y
wchar_t	*wcsncat(wchar_t *restrict s1, const wchar_t *restrict s2, size_t n);		Y
int	wcsncmp(const wchar_t *restrict s1, const wchar_t *restrict s2, size_t n);		Y

Function	Dependency	Thread Safety
`wchar_t *wcsncpy(wchar_t *restrict s1, const wchar_t *restrict s2, size_t n);`		Y
`wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);`		Y
`wchar_t *wcsrchr(const wchar_t *s, wchar_t wc);`		Y
`size_t wcsrtombs(char *restrict dst, const wchar_t **restrict src, size_t len, mbstate_t *restrict ps);`		Y
`size_t wcsspn(const wchar_t *s1, const wchar_t *s2);`		Y
`wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);`		Y
`intmax_t wcstoimax(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);`		Y
`wchar_t wcstok(wchar_t *restrict s1, const wchar_t *restrict s2, wchar_t **restrict ptr);`		Y
`long wcstol(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);`		Y
`long long wcstoll(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base)`		Y
`size_t wcstombs(char * restrict s, const wchar_t *restrict pwcs, size_t n);`		Y
`unsigned long wcstoul(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);`		Y
`unsigned long long wcstoull(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);`		Y
`uintmax_t wcstoumax(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);`		Y
`size_t wcsxfrm(wchar_t *restrict s1, const wchar_t *restrict s2, size_t n);`		Y
`int wctob(wint_t wc);`		Y

Function	Dependency	Thread Safety
`wctrans_t wctrans(const char *property);`		Y
`wctype_t wctype(const char *property);`		Y
`wchar_t *wmemchr(const wchar_t *s, wchar_t wc, size_t n);`		Y
`int wmemcmp(wchar_t *restrict s1, const wchar_t *restrict s2, size_t n);`		Y
`wchar_t *wmemcpy(wchar_t *restrict s1, const wchar_t *restrict s2, size_t n);`		Y
`wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);`		Y
`wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);`		Y
`int wprintf(const wchar_t *restrict format, ...);`	`write()`	IE
`int wscanf(const wchar_t *restrict format, ...);`	`close() read() sbrk() write()`	IE

Note

Memory allocated in `putenv()` is never freed. `unsetenv()` does not free any memory when it removes an entry from the environment.

libsys.a Function Implementation

This table lists the **libsys.a** low-level functions on which some **libansi.a** functions depend, and how the Green Hills tools implement each low-level function by default. If the default behavior is not adequate for your environment, you might need to re-implement the function.

Functions that Green Hills libraries implement as MULTI system calls communicate with your host machine. It is unlikely that you will need to change these functions when you connect to your target through MULTI. If you intend to continue using these functions without connecting to your target through MULTI, such as in a final product, you must re-implement these functions to suit your needs.

Function	Module	Implementation
`access()`	**ind_stat.c**	MULTI system call
`close()`	**ind_io.c**	MULTI system call
`creat()`	**ind_io.c**	MULTI system call
`_Exit()`	**ind_exit.c**	MULTI system call
`getpid()`	**ind_gpid.c**	Returns `17`
`__gh_timezone()`	**ind_tmzn.c**	8 hours West of UTC/GMT
`localtime()`	**ind_tmzn.c**	US daylight savings rules
`localtime_r()`	**ind_tmzn.c**	US daylight savings rules
`lseek()`	**ind_io.c**	MULTI system call
`open()`	**ind_io.c**	MULTI system call
`raise()`	**ind_sgnl.c**	ISO C99 compliant
`read()`	**ind_io.c**	MULTI system call
`sbrk()`	**ind_heap.c**	Allocates from `.heap`
`time()`	**ind_time.c**	MULTI system call
`unlink()`	**ind_io.c**	MULTI system call
`write()`	**ind_io.c**	MULTI system call

You may also need to change `char **environ`, an object in **ind_crt1.c** that is implemented using MULTI argument passing by default.

Wide Character Functions

The following pair of functions depend on the character encoding selected at link time (see “Host and Target Character Encoding” on page 203 and “Kanji Character Support” on page 658). Separate implementations are found in the libraries **libutf8_s32.a**, **libutf8_u16.a**, **lib8bit_s32.a**, **lib8bit_u16.a**, **libwc_s32.a**, and **libwc_u16.a**.

Function	Thread Safety
`int mbtowc(wchar_t * restrict pwc, const char *restrict s, size_t n);`	Y
`int wctomb(char *s, wchar_t wchar);`	Y

libmath.a Functions

The following tables list functions in **libmath.a**, and are organized by the header file in which they are declared.

Functions from math.h in libmath.a

Function		Thread safety
int	__fpclassifyd(double x);	Y
int	__fpclassifyf(double x);	Y
int	__fpclassifyl(double x);	Y
int	__isfinited(double x);	Y
int	__isfinitef(double x);	Y
int	__isfinitel(double x);	Y
int	__isinfd(double x);	Y
int	__isinff(double x);	Y
int	__isinfl(double x);	Y
int	__isnand(double x);	Y
int	__isnanf(double x);	Y
int	__isnanl(double x);	Y
int	__isnormald(double x);	Y
int	__isnormalf(double x);	Y
int	__isnormall(double x);	Y
int	__signbitd(double x);	Y
int	__signbitf(double x);	Y
int	__signbitl(double x);	Y
double	acos(double x);	E
float	acosf(float x);	E
long double	acosl(long double x);	E
double	acosh(double x);	E
float	acoshf(float x);	E

Function		Thread safety
long double	acoshl(long double x);	E
double	asin(double x);	E
float	asinf(float x);	E
long double	asinl(long double x);	E
double	asinh(double x);	Y
float	asinhf(float x);	Y
long double	asinhl(long double x);	Y
double	atan(double x);	Y
float	atanf(float x);	Y
long double	atanl(long double x);	Y
double	atan2(double y, double x);	Y
float	atan2f(float y, float x);	Y
long double	atan2l(long double y, long double x);	Y
double	atanh(double x);	E
float	atanhf(float x);	E
long double	atanhl(long double x);	E
double	cbrt(double x);	Y
float	cbrtf(float x);	Y
long double	cbrtl(long double x);	Y
double	ceil(double x);	Y
float	ceilf(float x);	Y
long double	ceill(long double x);	Y
double	copysign(double x, double y);	Y
float	copysignf(float x, float y);	Y
long double	copysignl(long double x, long double y);	Y
double	cos(double x);	Y
float	cosf(float x);	Y
long double	cosl(long double x);	Y
double	cosh(double x);	E

Function		Thread safety
float	coshf(float x);	E
long double	coshl(long double x);	E
double	erf(double x);	E
float	erff(float x);	E
long double	erfl(long double x);	E
double	erfc(double x);	E
float	erfcf(float x);	E
long double	erfcl(long double x);	E
double	exp(double x);	E
float	expf(float x);	E
long double	expl(long double x);	E
double	exp2(double x);	E
float	exp2f(float x);	E
long double	exp2l(long double x);	E
double	expm1(double x);	E
float	expm1f(float x);	E
long double	expm1l(long double x);	E
double	fabs(double x);	Y
float	fabsf(float x);	Y
long double	fabsl(long double x);	Y
double	fdim(double x, double y);	E
float	fdimf(float x, float y);	E
long double	fdiml(long double x, long double y);	E
double	floor(double x);	Y
float	floorf(float x);	Y
long double	floorl(long double x);	Y
double	fma(double x, double y, double z);	Y
float	fmaf(float x, float y, float z);	Y
long double	fmal(long double x, long double y, long double z);	Y

Function		Thread safety
double	fmax(double x, double y);	Y
float	fmaxf(float x, float y);	Y
long double	fmaxl(long double x, long double y);	Y
double	fmin(double x, double y);	Y
float	fminf(float x, float y);	Y
long double	fminl(long double x, long double y);	Y
double	fmod(double x, double y);	E
float	fmodf(float x, float y);	E
long double	fmodl(long double x, long double y);	E
double	frexp(double value, int *exp);	Y
float	frexpf(float value, int *exp);	Y
long double	frexpl(long double value, int *exp);	Y
double	gamma(double x);	E
double	hypot(double x, double y);	E
float	hypotf(float x, float y);	E
long double	hypotl(long double x, long double y);	E
int	ilogb(double x);	Y
int	ilogbf(float x);	Y
int	ilogbl(long double x);	Y
int	isfinite(double x);	Y
int	isinf(double x);	Y
int	isnan(double x);	Y
int	isnormal(double x);	Y
double	j0(double x);	Y
double	j1(double x);	Y
double	jn(int n, double x);	Y
double	ldexp(double x, int exp);	E
float	ldexpf(float x, int exp);	E
long double	ldexpl(long double x, int exp);	E

Function		Thread safety
double	lgamma(double x);	E
float	lgammaf(float x);	E
long double	lgammal(long double x);	E
double	log(double x);	E
float	logf(float x);	E
long double	logl(long double x);	E
double	log10(double x);	E
float	log10f(float x);	E
long double	log10l(long double x);	E
double	log1p(double x);	E
float	log1pf(float x);	E
long double	log1pl(long double x);	E
double	log2(double x);	E
float	log2f(float x);	E
long double	log2l(long double x);	E
double	logb(double x);	E
float	logbf(float x);	E
long double	logbl(long double x);	E
double	lrint(double x);	E
float	lrintf(float x);	E
long double	lrintl(long double x);	E
long long	llrint(double x);	E
long long	llrintf(float x);	E
long long	llrintl(long double x);	E
long	lround(double x);	E
long	lroundf(float x);	E
long	lroundl(long double x);	E
long long	llround(double x);	E
long long	llroundf(float x);	E

Function		Thread safety
long long	llroundl(long double x);	E
double	modf(double value, double *iptr);	Y
float	modff(float value, float *iptr);	Y
long double	modfl(long double value, long double *iptr);	Y
double	nan(const char *tagp);	Y
float	nanf(const char *tagp);	Y
long double	nanl(const char *tagp);	Y
double	nearbyint(double x);	Y
float	nearbyintf(float x);	Y
long double	nearbyintl(long double x);	Y
double	nextafter(double x, double y);	E
float	nextafterf(float x, float y);	E
long double	nextafterl(long double x, long double y);	E
double	nexttoward(double x, long double y);	E
float	nexttowardf(float x, long double y);	E
long double	nexttowardl(long double x, long double y);	E
double	pow(double x, double y);	E
float	powf(float x, float y);	E
long double	powl(long double x, long double y);	E
double	remainder(double x, double y);	E
float	remainderf(float x, float y);	E
long double	remainderl(long double x, long double y);	E
double	remquo(double x, double y, int *quo);	E
float	remquof(float x, float y, int *quo);	E
long double	remquol(long double x, long double y,int *quo);	E
double	rint(double x);	Y
float	rintf(float x);	Y
long double	rintl(long double x);	Y
double	round(double x);	Y

Function		Thread safety
float	roundf(float x);	Y
long double	roundl(long double x);	Y
double	scalbn(double x, int n);	E
float	scalbnf(float x, int n);	E
long double	scalbnl(long double x, int n);	E
double	scalbln(double x, long int n);	E
float	scalblnf(float x, long int n);	E
long double	scalblnl(long double x, long int n);	E
int	signbit(double);	Y
double	sin(double x);	Y
float	sinf(float x);	Y
long double	sinl(long double x);	Y
double	sinh(double x);	E
float	sinhf(float x);	E
long double	sinhl(long double x);	E
double	sqrt(double x);	E
float	sqrtf(float x);	E
long double	sqrtl(long double x);	E
double	tan(double x);	E
float	tanf(float x);	E
long double	tanl(long double x);	E
double	tanh(double x);	E
float	tanhf(float x);	E
long double	tanhl(long double x);	E
double	tgamma(double x);	E
float	tgammaf(float x);	E
long double	tgammal(long double x);	E
double	trunc(double x);	Y
float	truncf(float x);	Y

Function		Thread safety
long double	truncl(long double x);	Y
double	y0(double x);	E
double	y1(double x);	E
double	yn(int n, double x);	E

Functions from fenv.h in libmath.a

Function		Thread safety
void	feclearexcept(int excepts);	Y
void	fegetenv(fenv_t *envp);	Y
void	fegetexceptflag(fexcept_t *flagp, int excepts);	Y
int	fegetround(void);	Y
int	feholdexcept(fenv_t *envp);	Y
void	feraiseexcept(int excepts);	Y
void	fesetenv(const fenv_t *envp);	Y
void	fesetexceptflag(const fexcept_t *flagp, int excepts);	Y
int	fesetround(int round);	Y
int	fetestexcept(int excepts);	Y
void	feupdateenv(const fenv_t *envp);	Y

Functions from complex.h in libmath.a

Function		Thread safety
double complex	cabs(double complex z);	E
float complex	cabsf(float complex z);	E
long double complex	cabsl(long double complex z);	E

Function		Thread safety
double complex	cacos(double complex z);	E
float complex	cacosf(float complex z);	E
long double complex	cacosl(long double complex z);	E
double complex	cacosh(double complex z);	E
float complex	cacoshf(float complex z);	E
long double complex	cacoshl(long double complex z);	E
double complex	casin(double complex z);	E
float complex	casinf(float complex z);	E
long double complex	casinl(long double complex z);	E
double complex	casinh(double complex z);	E
float complex	casinhf(float complex z);	E
long double complex	casinhl(long double complex z);	E
double complex	carg(double complex z);	E
float complex	cargf(float complex z);	E
long double complex	cargl(long double complex z);	E
double complex	catan(double complex z);	E
float complex	catanf(float complex z);	E
long double complex	catanl(long double complex z);	E
double complex	catanh(double complex z);	E
float complex	catanhf(float complex z);	E

Function		Thread safety
long double complex	catanhl(long double complex z);	E
double complex	ccos(double complex z);	E
float complex	ccosf(float complex z);	E
long double complex	ccosl(long double complex z);	E
double complex	ccosh(double complex z);	E
float complex	ccoshf(float complex z);	E
long double complex	ccoshl(long double complex z);	E
double complex	cexp(double complex z);	E
float complex	cexpf(float complex z);	E
long double complex	cexpl(long double complex z);	E
double complex	cimag(double complex z);	E
float complex	cimagf(float complex z);	E
long double complex	cimagl(long double complex z);	E
double complex	clog(double complex z);	E
float complex	clogf(float complex z);	E
long double complex	clogl(long double complex z);	E
double complex	conj(double complex z);	E
float complex	conjf(float complex z);	E
long double complex	conjl(long double complex z);	E
double complex	cpow(double complex x, double complex y);	E

Function		Thread safety
float complex	cpowf(float complex x, float complex y);	E
long double complex	cpowl(long double complex x, long double complex y)	E
double complex	cproj(double complex z);	E
float complex	cprojf(float complex z);	E
long double complex	cprojl(long double complex z);	E
double complex	creal(double complex z);	E
float complex	crealf(float complex z);	E
long double complex	creall(long double complex z);	E
double complex	csin(double complex z);	E
float complex	csinf(float complex z);	E
long double complex	csinl(long double complex z);	E
double complex	csinh(double complex z);	E
float complex	csinhf(float complex z);	E
long double complex	csinhl(long double complex z);	E
double complex	csqrt(double complex z);	E
float complex	csqrtf(float complex z);	E
long double complex	csqrtl(long double complex z);	E
double complex	ctan(double complex z);	E
float complex	ctanf(float complex z);	E
long double complex	ctanl(long double complex z);	E

Function		Thread safety
double complex	ctanh(double complex z);	E
float complex	ctanhf(float complex z);	E
long double complex	ctanhl(long double complex z);	E

Functions from stdlib.h in libmath.a

Function		Thread safety
double	atof(const char *nptr);	E
double	strtod(const char *restrict nptr, char **restrict endptr);	E
float	strtof(const char *restrict nptr, char **restrict endptr);	E
long double	strtold(const char *restrict nptr, char **restrict endptr);	E

Functions from wchar.h in libmath.a

Function		Thread safety
double	wcstod(const wchar_t *restrict nptr, wchar_t **restrict endptr);	E
long double	wcstold(const wchar_t *restrict nptr, wchar_t **restrict endptr);	E
float	wcstof(const wchar_t *restrict nptr, wchar_t **restrict endptr);	

Functions from setjmp.h in libmath.a

Function		Thread safety
void	longjmp(jmp_buf env, int val);	Y

Function	Thread safety
int setjmp(jmp_buf env);	Y

Functions from time.h in libmath.a

Function	Thread safety
double difftime(time_t time1, time_t time0);	Y

libind.a Functions

Function		Thread safety
int	_rnerr(int num, int linenum, char *str, void *ptr, void **trace, int depth, size_t len)	Y
int	_rnerr2(int num, int linenum, char *str)	Y
void	*memmove(void *s1, const void *s2, size_t n);	K

Additional information about wide character is provided in “Wide Character Functions” on page 797.

Note

The functions `_rnerr()` and `_rnerr2()` are used in run-time error checking. For more information about run-time error checking, see “Run-Time Error Checks” on page 159.

libstartup.a Functions

The following table lists the standard C library functions in **libstartup.a**.

Function		Thread safety
void	*memcpy(void * restrict d, const void * restrict s, size_t n)	YK
void	*memset(void *s, int c, size_t n);	YK

Standard Function Names Converted by the Linker

Some Standard C library functions are stored in two formats. The linker looks for calls to these functions using the standard names and converts those into calls to either of the special names, based on the settings of certain options. Always refer to the standard name; the linker is responsible for associating it with a special one. These functions are:

- `wchar_t *` conversion functions (**-wchar_u16**):
  - `__ghs_wchar_u16_wcstod`

    - `__ghs_wchar_u16_wcstof`
    - `__ghs_wchar_u16_wcstold`
- `wchar_t *` conversion functions (**-wchar_s32**):
    - `__ghs_wchar_s32_wcstod`
    - `__ghs_wchar_s32_wcstof`
    - `__ghs_wchar_s32_wcstold`
- `time_t` functions (**-time64**):
    - `__ghs_time64_difftime`
    - `__ghs_time64_gmtime`
    - `__ghs_time64_gmtime_r`
    - `__ghs_time64_localtime`
    - `__ghs_time64_localtime_r`
    - `__ghs_time64_mktime`
    - `__ghs_time64_time`
    - `__ghs_time64_times`

Customizing the Run-Time Environment Libraries and Object Modules

The following sections contain information about the libraries and modules that make up the Green Hills run-time environment, and explain how to create a project that allows you to customize them.

Creating a Project with a Customizable Run-Time Environment

While the standard Green Hills run-time environment is sufficient for most users, you may need to customize some of its underlying routines to suit your needs. To create a project that allows you to customize the run-time environment:

1. If you do not have an existing project, click the  button in the Project Manager to open the **Project Wizard**, and select the appropriate settings until you reach the **Link Options** screen.

   If you have an existing project, right-click a program in that project and select **Configure**.
2. On the screen that allows you to specify your **Program Layout**, select one or more of the **Libraries** check boxes (see also the documentation about settings for Stand-Alone programs in the *MULTI: Managing Projects and Configuring the IDE* book). If you set your **Program Layout** to `Link to ROM and Execute out of RAM`, some startup code runs out of ROM (see "Assigning Program Sections to ROM and RAM" on page 81).

The **Project Wizard** adds the following files your target resources project (**tgt/resources.gpj**), depending on the check boxes you selected:

- **libstartup/startup.gpj** — A project that contains the following startup code:
  - **crt0.800** — see "Startup Module crt0.o" on page 815.
  - **libstartup.gpj** — see "Low-Level Startup Library libstartup.a" on page 817.
- **libsys/libsys.gpj** — see "Low-Level System Library libsys.a" on page 818.
- **libboardinit/libboardinit.gpj** — see "Board Initialization Library libboardinit.a" on page 823. This library is not available for all targets.

You can customize the source to the startup modules as necessary and rebuild them. Be careful when making any changes to the build options for these projects, because some of them are required for proper operation. For example, **libstartup/startup.gpj** and **libsys/libsys.gpj** define the preprocessor symbol `EMBEDDED` when compiling. This symbol is required for some modules in the libraries. For more information about building and setting build options, see the documentation about the MULTI Project Manager in the *MULTI: Managing Projects and Configuring the IDE* book.

Note

The interface to these modules might change between releases of the product. If you customize your run-time environment and later upgrade to a new release, you may need to propagate your customizations to the run-time environment shipped with that new releases. You may not be able to combine the run-time environment from a given release with other libraries in a later release.

Working With Makefiles

If you are using makefiles to invoke the compiler driver and want to customize your run-time environment:

1. Create your project with the **Project Wizard** using the steps provided in "Creating a Project with a Customizable Run-Time Environment" on page 813.
2. Modify each run-time environment file as needed.
3. Use your makefile to link with the modified files.

Optimizing Startup Code

If you want to build faster, smaller startup code with reduced functionality, define the `MINIMAL_STARTUP` preprocessor symbol in the **libstartup/libstartup.gpj** and **libsys/libsys.gpj** projects. This startup code may be useful for very small and simple test applications that run out of RAM. The following list contains some of the features that are not supported in this library mode:

- ROM to RAM copy
- PIC
- PID

- multithreading
- manual profiling

This library mode is provided as-is. It is intended for experienced users who understand its limited feature set.

Startup Module crt0.o

crt0.o is the default startup module linked with your program. The source code for **crt0.o** is contained in the project **libstartup/startup.gpj**, in the file **crt0.800**. This file contains the `_start()` function, which is the default entry point for the program. If you set your **Program Layout** to `Link to ROM and Execute out of RAM`, the initialization code in this module is placed in and executes out of the uncompressed `.ROM.boottext` section.

crt0.800

crt0.800 is a small target-specific assembly language file containing code to set up a C program environment before calling into a C function. It executes as follows:

1. On startup, the code uses a system call to determine whether a debug server is controlling execution of the program (as opposed to the program running stand-alone, without being connected to a debug server).
2. If the system call is successful, **crt0.800** assumes that the debug server has already performed some register initialization (such as the stack pointer).

   If the system call is not successful, **crt0.800** initializes additional registers, including the stack pointer. Consult the code to determine which other registers are manipulated, because this varies across architectures.
3. After register initialization, `_start()` calls the function `__ghs_ind_crt0()`, the target-independent startup routine located in **libstartup.a**.

Customizing crt0.800

If you create your own startup code, you must initialize fixed registers (defined by your processor's ABI) to point to the correct memory regions. The register most

commonly requiring initialization is the stack pointer, because a valid stack is generally required before a high-level language function can be called.

When initializing the stack pointer, the following information may be helpful:

- The `.stack` section defined in your linker directives (**.ld**) file specifies the location and size of the run-time stack.

  For example, the following excerpt from a linker directives file specifies that the stack starts on the first 16-byte aligned address following the `.heap` section in memory. The stack is configured to be `0x80000` bytes:

  ```
  .heap align(16) pad(0x100000) :
  .stack align(16) pad(0x80000) :
  ```

- `__ghsbegin_stack` and `__ghsend_stack` specify the beginning and the end of the `.stack` section. For more information about special symbols of this format, see “Beginning, End, and Size of Section Symbols” on page 468.
- The stack pointer should be initialized to the end of this section, because it grows downward. To check if the stack is exhausted at run time, compare the stack pointer to `__ghsbegin_stack`.

Low-Level Startup Library libstartup.a

The **libstartup.a** library contains code that copies sections of a program's text and initialized data from ROM to RAM. On PIC and/or PID systems, it also adjusts initializers based on the location of these sections at run time. The code is contained in **libstartup.gpj**.

If you set your **Program Layout** to `Link to ROM and Execute out of RAM`, the initialization code in this library is placed in and executes out of the uncompressed `.ROM.boottext` section.

ind_crt0.c

This module contains startup code in the function `__ghs_ind_crt0()`. This function:

1. clears uninitialized sections such as `.bss`.
2. copies initialized sections from their locations in ROM to their final locations in RAM, as specified in your linker directives file.
3. decompresses compressed initialized sections from their locations in ROM and copies them to their final location in RAM. For more information, see “Copying a Section from ROM to RAM at Startup” on page 461.
4. relocates initialized pointers to any position independent object, if you are using PIC and/or PID (see “Relocating Initialized Pointers” on page 817).
5. calls the library initialization routine `__ghs_ind_crt1()` in the `.text` section through a function pointer. The application then executes code in the `.text` section. For more information, see “ind_crt1.c” on page 819.

Relocating Initialized Pointers

If your program uses PIC and/or PID, the **ind_crt0.c** code relocates initialized pointers to any position independent object. For example, consider the following C code:

```
extern int foo();
int (*ptr)() = &foo;
```

This declares a global function pointer `ptr` that is initialized to the address of the function `foo()`. Because the location of `foo()` is not known at link time, the compiler emits data that describes each such relocated initialization in the program. For example, when compiling for PIC, the compiler generates data to inform **ind_crt0.c** that the initializer of the variable `ptr` requires a run-time modification specifying the location of the program's code, as desired. After **ind_crt0.c** finishes, all initialized pointers contain valid run-time addresses.

ind_mcpy.800, ind_mset.800

These modules contain the C run-time library functions `memcpy()` and `memset()`, which are used during initialization to copy and clear data sections.

ind_reset.800

This module contains low-level initialization routines. You may need to modify it for your target.

ind_uzip.c

This module contains code that decompresses program sections when copying them from ROM to RAM. It is pulled in by the linker if any CROM sections are contained in the program.

ind_initmem.c

This module is not used on V850 and RH850.

Low-Level System Library libsys.a

The **libsys.a** library contains code that performs run-time library initialization, profiling, and system calls. The source code is contained in ***install_dir*/src/libsys**.

Several **libsys.a** I/O functions perform system calls on the host when you are connected to your target with the MULTI Debugger. For more information, see “libsys.a Function Implementation” on page 796.

ind_alloc.c

This module contains functions that allocate memory of the given type and alignment from the heap. These functions call `sbrk()` (in **ind_heap.c**) to allocate the memory, but also provide alignment guarantees.

ind_call.800, ind_dots.800

These modules implement low-level system call capability. The routine `__ghs_syscall()` is called by various system call routines, such as `open()`, `close()`, `read()`, and `write()`. The `__ghs_syscall()` routine transfers control to the start address of the `.syscall` section.

Debug servers can set a breakpoint at the start of the `.syscall` section to emulate system calls on the host. When the breakpoint is hit, the debug server knows that a system call occurred. The arguments are retrieved and the system call is emulated on the host.

ind_crt1.c

This module contains the first C function called by the standard libraries after memory and static and global data have been initialized. It initializes the ANSI C and other run-time libraries before jumping to the application entry point (`main`).

ind_errn.c

This module handles reads and writes to `errno` and can be customized to maintain a per-thread `errno` value. See also **ind_thrd.c**.

ind_gpid.c

This module returns the current process number. This is used in `mktemp()` to add uniqueness to the generated filenames.

ind_lock.c

This module implements the global locking mechanism needed for thread-safe libraries.

ind_manprf.c

This module contains code to perform stochastic target-based timing profiling when enabled by the **-timer_profile** option (see "Enabling Sampled Profiling" on page 63).

ind_stackcheck.c

This module contains code to perform stack checking when enabled by the **-stack_check** option.

ind_thrd.c

This module implements the file-specific locking mechanism and optional per-thread run-time library data structures. In addition, this module provides the following functions to allow saving and restoring arbitrary state across `setjmp()` and `longjmp()` calls:

```
int __ghs_SaveSignalContext(jmp_buf jmpbuf)
void __ghs_RestoreSignalContext(jmp_buf jmpbuf)
```

ind_mcnt.800, ind_gcnt.800, ind_bcnt.c, ind_mprf.c, ind_gprf.c

These modules provide profiling support. The routine `__ghs_mcount()` is used for call count profiling and `__ghs_gcount()` is used for call graph profiling. The module **ind_bcnt.c** contains the profiling routine `__ghs_bcount()`, which handles calls that are emitted by the compiler to implement code coverage analysis.

ind_heap.c

This module contains the dynamic memory allocation routine `sbrk()`. The `.heap` section in the linker directives file specifies the location, size, alignment, and initialization, if any, of the heap. For example:

```
.heap align(16) pad(0x100000) :
.stack align(16) pad(0x80000) :
```

This specifies that the heap starts on the first 16-byte aligned address following the previous section in memory and is `0x100000` bytes in size. The `.stack` section then follows the heap area. The `pad` directive instructs the linker that the heap is uninitialized on startup. This enables you to place the run-time heap anywhere in memory; the run-time library automatically allocates memory where you place this section.

Specifying a size for the heap helps ensure that your program does not use more heap memory than expected. Any attempt to allocate more memory than specified in `.heap` causes `sbrk()` to fail and return `(void*)-1`.

ind_io.c

This module contains default implementations of system routines most likely needed for a Linux/Solaris-like implementation, including:

```
int open(const char *filename, int mode, ...);
int creat(const char *filename, int prot);
int close(int fno);
int read(int fno, void *buf, int size);
int write(int fno, const void *buf, int size);
int unlink(char *name);
```

These system calls enable the program to open, read, write, and close files.

The calls in **ind_io.c** perform these operations on the host using the system call interface routine `__ghs_syscall()`, described in “ind_call.800, ind_dots.800” on page 819.

ind_iob.c

This module contains the default file buffer, `_iob`, used by the C library's **stdio.h** interface. The user may want to modify this to reduce or increase the number of files that may be opened through this interface. The user may also modify the default buffering by modifying `__gh_iob_init()`.

ind_exit.c

This module contains the following routines:

Routine	Description
`__ghs_at_exit(struct __GHS_AT_EXIT*)`	Registers functions that will be called by `_Exit()` (similar to the C library `atexit()`).
`void _Exit(int)`	Calls clean-up routines registered by the program via `__ghs_at_exit()` and allows the program to terminate cleanly via the `__ghs_syscall(SYSCALL_EXIT, code)` system call. Clean-up routines registered by `atexit()` and C++ global destructors are handled separately in `exit()`, which finishes by calling `_Exit()`.

Other Low-Level Functions

The following files contain a basic set of Linux/Solaris-like operating system routines. Each routine is commented to show the function it performs and the values it returns. These routines allow the linker to resolve an operating system's symbols, referenced by the Green Hills run-time libraries.

Source File	Routine(s)
ind_gmtm.c	`struct tm *gmtime(const time_t *timer);`
ind_renm.c	`int rename(const char *old, const char *new);`
ind_sgnl.c	`int raise(int sig);` `void (*signal(int sig, void (*func)(int)))(int);`
ind_stat.c	`int access(char *name, int mode);`
ind_syst.c	`int system(const char *string);`

Source File	Routine(s)
ind_time.c	`time_t time(time_t *tptr);` `int times (struct tms *buffer);`

ind_tmzn.c	`struct tm *localtime(const time_t *timer);`
	`void tzset(void);`
	`int __gh_timezone(void);`

ind_trnc.c	`int truncate(const char* path, long length);` This function is deprecated and may be removed in a future release.

Board Initialization Library libboardinit.a

If you choose to customize the **Board Initialization** library in the **Project Wizard** (see "Creating a Project with a Customizable Run-Time Environment" on page 813), it adds **libboardinit/libboardinit.gpj** to your Top Project. This project builds **libboardinit.a**, which varies by target and might contain some or all of the following features.

- ROM-based program support. This includes linker directives files and code to initialize the memory, chip selects, TLBs, and caches to allow the program to execute starting from reset.
- Cache synchronization support. This consists of the routine `void __ghs_board_cache_sync(void* s, size_t n)`, which is called after each ROM-to-RAM copy of a section containing program text. This routine flushes the data cache and invalidates the instruction cache contents starting at address *s* and continuing for *n* bytes.
- Serial port support. This might include routines to transmit and receive characters, as well as optionally including an example of an interrupt service routine to handle serial port interrupts.
- I/O library overrides. This includes code to connect the C/C++ library's `stdin` input stream and `stdout` output stream to the serial port. This code defines alternate versions of the `read()` and `write()` system routines that are otherwise defined by **ind_io.c** in the low-level system library **libsys.a**.
- Timer support. This consists of code that is called when target-based timer profiling is enabled. For more information about this feature, see "Enabling Sampled Profiling" on page 63.

The memory initialization code can be placed either at the reset vector location or in a routine `__ghs_board_memory_init`, which is called from the Startup Module **crt0.o** if it exists. After this routine returns, the memory defined by the linker directives file must be accessible to the program. This routine should be written in assembly because C code might inadvertently rely on stack memory being accessible.

The remaining device initialization code for the serial port, the interrupt controller, the I/O library overrides, and the timer setup code is placed in the routine `__ghs_board_devices_init`. `__ghs_board_devices_init` can be written in C because memory will be initialized, although not all of the C library is usable.

If the file **board_init.txt** was automatically included in your new project, it contains further information about debugging stand-alone and other programs on your board.

Features that Depend on the Default Green Hills Startup Code

This section provides a partial list of features and options that depend on the Green Hills startup code. If you customize this code, the following features might not work as expected:

Feature	Startup Code Requirement
`atexit()`	Processing in the Green Hills `_exit()` function.
C++ destructors	The Green Hills `exit()` function.
Multithreading	Might require `__gh_lock_init()` to be called.
PIC and PID	Might require base registers to be set up.
`stdin`, `stdout`, and `stderr`	`__gh_iob_init()` must be called.

The following options also depend on Green Hills startup code:

Option	Startup Code Requirement
-a **-p** **-pg**	The Green Hills `exit()` or `_exit()` functions dump profiling data. To learn how to dump profiling data during a program's execution, see the documentation about manually dumping profiling data in the *MULTI: Debugging* book.
-additional_sda_reg	Requires Green Hills startup code.
-globalreg	Certain variables (such as those declared with `register int glob`) are initialized to the value `0` by startup code.

Note

This is not a complete list of features that depend on Green Hills startup code. Because customizing this code may cause certain features to exhibit unexpected behavior, do so only when necessary.

Chapter 17

Mixing Languages and Writing Portable Code

Contents

Mixing Languages

Green Hills compilers can combine C and C++ routines within the same executable files, subject to certain constraints. This section provides some basic information about using Green Hills compilers to create a mixed language executable and describes how the compiler driver builds mixed language executables. Additional language-specific details are provided in the subsequent sections.

All Green Hills compiler drivers are compatible with each other. This permits a C driver to compile a C++ module, and a C++ driver to compile a C module. The driver uses the input filename extension to determine the correct language, rather than assuming that the name of the driver determines the source code language.

Though compatible during compilation, the various drivers differ during the link phase. To link an application, the driver must determine all of the languages in use in order to know which libraries to include. The driver assumes that every application has modules written in C and assembly language, and that there is at least one module written in the driver's default language. If the command line includes source files written in other languages (as indicated by the file extension), then the driver recognizes that those languages exist in the application as well.

Consequently, mixing any one language with C is easy because the driver always assumes C is in use. To ensure the correct linkage when mixing a language with C, use the driver for the language other than C for linking the application.

Initialization of Libraries

A multiple language application may need to perform input and output in more than one language. With a little care to avoid conflicts between languages, this is fully supported. If input and output will always be performed on different files by each language, then the main program in a single language application automatically handles the initialization and deinitialization of each language's run-time routines. Therefore, if the application will only perform I/O in one language other than C, then it is easy to write the main program for the application in that language. For more complex requirements, write a main program in C to perform the initialization and deinitialization of the library run-time routines. Examples of main programs in C are given in the next section.

Examples of main() Programs

All language environments automatically perform initializations at the start of the execution and cleanup at the termination. The initializations and cleanup vary depending on the language. Examples of initializations and cleanup include command line argument initialization, standard input/output configuration, and static object creation/destruction. Since the C language initializations and cleanup differ from that of the other languages, you must often insert code inside the `main()` function to ensure proper configuration.

C main() Program for C++

If a program uses both C and C++, and the `main()` function is written in C, insert a call to `_main()` as the first executing statement and a call to `exit(0)` as the last executing statement (see example below). You must do this to ensure that static constructors and destructors are properly configured. For more details on using C++ in C programs refer to “Using C++ in C Programs” on page 831.

```
void main(void)
{
  _main();/* must be first executable line */
  /* rest of main goes here */
  exit(0);/* must be last executable line */
}
```

Performing I/O on a Single File in Multiple Languages

Some applications benefit from performing input and output on a single file or device from more than one language; one example of this is pre-opened files. In C, these are `stdin`, `stdout`, and `stderr`. In C++, they are the same as C or, alternatively, `iostreams` `cin` , `cout` , or `cerr` may be used.

All languages have full access to these pre-opened files, and input and output can easily be mixed between the languages on these files. However, for the best results, a complete input or output operation is done in a single language. In C, any call to a library function which performs input or output is a complete operation. If this rule is followed, all data will be output correctly and in the intended sequence. The C library routine `fflush()` flushes the buffer of the pre-opened files in all languages. In addition, to flush one of C++ `iostreams`, use the notation *`file`* `<< flush`. For example:

```
std::cout << std::flush;
```

Performing input and output on a single file which is not preopened is more difficult. It is possible to open the file once in each language and perform input and output independently in each language. In many cases this would be unacceptable, particularly when working with a device rather than a simple file.

C Routines and Header Files In C++

The C++ language allows much use of existing C code. Therefore, it is fairly straightforward to call functions written in C from C++. The syntax of the two languages is very similar and the use of header files has been continued in C++.

By default in C++, the names of functions are encoded or mangled, whereas in C, the names of functions are unchanged. C++ provides the `extern` specifier to identify non-C++ functions so their names will not be mangled. Therefore, this specifier allows declarations for C functions to be included and then linked with the compiled C object code.

To specify a C declaration:

- Specify or declare functions individually. The following example specifies that two functions with `extern "C"` linkage are to be declared.

```
extern "C" {
  int fclose(FILE *);
  FILE *fopen(const char *, const char *);
}
```

We recommend that you do not use a function declaration with an empty parameter list. This means different things in C and C++ (see "Function Prototyping in C vs. C++" on page 832). For compatibility in both languages, specify the parameters, or use a parameter list consisting solely of the keyword `void` if there are no parameters. For example:

```
int func1(int a, short b, const int *c); /* GOOD */
void func2(void);                        /* GOOD */
int func3();                             /* BAD  */
```

- If code contains both C and C++, then the `extern` constructs can be placed within `#ifdef __cplusplus` statements. This practice is common within header files. For example,

```
#ifdef __cplusplus
extern "C" {
#endif
void assert(int);
void _assert(const char *,const int,const char *);
#ifdef __cplusplus
}
#endif
```

Using C++ in C Programs

To call C++ from C, declare an `extern "C"` interface for the C++ code. Otherwise, the C code must manually perform some of the tasks that the C++ compiler performs automatically. You must know the following internal mechanics and details of the C++ implementation:

- The C++ compiler encodes or mangles function and class member names. Any C++ function or class members called from a C program must be referenced by the encoded or mangled names.

- The manner in which a C++ compiler handles member functions must be known. All member functions (except static member functions) have the special object member pointer `this` inserted automatically as the first argument in the parameter list. A C programmer must manually add the argument `this` when calling any member functions from C.
- Handling constructors and destructors for static objects requires special processing. On most systems, the `main()` function has a special function call to `_main()` inserted to ensure that all static constructor/destructor calls are made properly. If `main()` is not in a C++ module, then the C programmer must manually include calls to `_main()` in the C main module. The `_main()` code is contained in the C++ library and therefore must be linked into the final executable.

Note

When using INTEGRITY and creating a C task, `_main` is called before `main()` so that if `main()` is in C code but there are C++ static constructors, the call is still constructed.

- Virtual functions are also handled automatically by a C++ compiler, but involve additional coding to access or use them from a C environment.

We recommend that your C code always call into C++ code with `extern "C"`; using any other method results in code that is not portable among compilers,` or even among releases of the same compiler.

Function Prototyping in C vs. C++

In ANSI C and C++, header files provide prototypes for library functions which enforce a standard interface between the calling program and the called function.

Function prototyping requires that the function declaration include the function return type and the number and type of the arguments. When a prototype is available for a function, the compiler is able to perform argument checking and coercion on calls to that function. If a prototype is not available for a function when it is called, ANSI C will behave like K&R C. The return type of the function is assumed to be `int`, and actual arguments will be promoted to `int`, `long`, or `double` types, or pointers as appropriate. In C++, however, it is an error to call a function which has not been declared with a prototype.

Another important difference between ANSI C and C++ is that a non-prototype declaration of a function, such as:

```
char *function_name();
```

has no effect on the number and type of the arguments in ANSI C. In C++ it is understood as:

```
char *function_name(void)
```

which means that the function has no arguments at all. If the function declaration occurs within the scope of an `extern "C"` declaration, the function has non-C++ linkage, and therefore cannot be overloaded. This means that if a traditional K&R style declaration of a function appears in a header file, and the `#include` directive which accesses that header file is enclosed in `extern "C" { }`, then it will be impossible to redeclare that function with arguments.

Writing Portable Code

Many compiler vendors have implemented the C and C++ languages for a wide variety of system architectures. One important reason for this is that these languages greatly simplify the task of building and maintaining software for multiple platforms. However, not all features of C and C++ accommodate this goal. In fact, both languages intentionally provide features that perform differently on different systems. For a program to be truly portable, the programmer must avoid these non-portable features of the C and C++ languages. This chapter discusses some of the non-portable assumptions that can cause programs to fail and how to avoid portability problems with Green Hills C/C++ compilers.

The C/C++ language specifications define programs in such a way that portable programs will always work with all appropriate language compilers, including any Green Hills C/C++ compiler. However, difficulties arise when programmers make non-portable assumptions about the machine or compiler that they are using. As a result, a program may appear to compile and operate correctly when compiled with one vendor's compiler, but not with a Green Hills compiler.

Certain differences between compilers are specific to the individual compiler vendor. In addition, many more differences are particular to the processor and the operating system being used. To avoid these differences when porting between different system architectures, portable code should be consistently written.

Compatibility Between Green Hills Compilers

All Green Hills C/C++ compilers follow the same interpretation of the C/C++ languages as described in this manual; however, some Green Hills C or C++ compilers do not include certain features or options.

For information about compatibility with Green Hills compilers for other languages, see “Mixing Languages” on page 828.

Word Size Differences

Green Hills compilers support machines with 32-bit and 64-bit word size. Porting C/C++ programs between machines of different word sizes requires particular care because word size may affect most primary data types.

Some machines are *byte addressable* , meaning that their addresses refer to 8-bit bytes. Typically, byte addressable machines operate on 8, 16, 32, 64, and 128-bit quantities. Other machines are *word addressable*, meaning that their addresses refer to words of a standard size varying from 16 to 64 bits. Word addressable machines typically operate on multiples of the word size.

A program that operates on a machine of one word size may not operate on a machine of a different word size. Thus, word size incompatibility problems may arise if two different machines have different word sizes, or if one machine is word addressable and the other is byte addressable. The word size affects the range of numbers implemented by integer data types, as well as the precision and range of single and double precision floating-point data types.

The most common word size problems are integer and floating-point underflows, overflows, and loss of precision. The layout of bit aligned data structures will vary with the word size, so overlaying structures in memory makes programs difficult to port to another compiler. Address arithmetic done in integer variables is often not portable.

Range of Representable Values

The size of each basic numeric type controls the range of values which may be represented by that type. The header files **<limits.h>** and **<float.h>** provide defined symbols which represent the minimum and maximum values for all numeric data

types in C/C++. A portable program should use these symbols and never depend on the use of values outside the allowed range.

If arithmetic operations cause overflow, underflow, or loss of precision, the program may not detect the error or may behave differently on different systems.

Relative Sizes of Data Types

Both C and C++ place very weak requirements on the relative size of the basic types, though programs in either language commonly assume otherwise. For example, both languages only require that `short` be no larger than `int` and that `int` be no larger than `long`. It would be legal for `short`, `int`, and `long` to all be the same size or for them all to be different. For instance, with some 32-bit Green Hills C/C++ compilers, `short` is 16 bits and both `int` and `long` are 32 bits. The assumption that `int` is twice as large as `short`, but the same size as `long` is non-portable, because, with 64-bit Green Hills C/C++ compilers, `short` is 16 bits, and `int` and `long` are either 32 or 64 bits.

Another common non-portable assumption is that pointers are the same size as `int` or `long`. Neither is guaranteed. With most 32-bit Green Hills C/C++ compilers, pointers are 32 bits. With 64-bit Green Hills C/C++ compilers, pointers may be either 32 or 64 bits, independent of the size of `int` or `long`. Use the `uintptr_t` and `intptr_t` types provided by **stdint.h** as portable integer types large enough to represent any data pointer, but not function pointers.

In both languages, all integer constants have type `int` unless marked with a type suffix. In certain cases, the use of a plain integer constant instead of a long integer constant may be non-portable.

Endianness Problems

Programs that overlay characters and integers in memory or that use character pointers to integer variables and vice versa are often not portable between machines with different byte ordering.

Programs that declare a single variable with different integer types in different modules may fail when ported to a machine with a different byte order.

Alignment Requirements

Some systems will not load or store a two byte object unless that object is on an even address. Other systems have a similar requirement for four or eight byte objects. Others may allow certain accesses, but require more time to perform them. Therefore, alignment of data is both a matter of correctness and of time efficiency. Although increased alignment may improve performance, it also consumes space, due to padding inserted to achieve alignment.

The alignment requirements on each system are chosen both to satisfy the restrictions of the hardware and to achieve a reasonable balance between performance and space. The alignment rules are often not configurable and they differ for each system. Thus, programs that make assumptions about the relative position of data objects in memory or elements within classes, structures, or arrays are not portable—even among the Green Hills C++ compilers.

The C/C++ languages impose the following restrictions on size and alignment:

- The alignment of a structure, union, class, or array is equal to or greater than the maximum alignment requirement of any of its members.
- The size of a structure, union, class, or array is always a multiple of the maximum alignment requirement of any of its members.
- The offset of any member of a structure, union, class, or array is always a multiple of its alignment requirement.
- All dynamic memory allocation routines provided with the compiler will return a pointer aligned to the maximum alignment for any object on that machine.

All Green Hills C/C++ compilers also satisfy these principles:

- The stack is maintained on an alignment suitable for any object.
- Parameters and local variables are allocated on the stack according to their alignment requirement.
- Local variables are arranged on the stack to avoid unnecessary padding due to alignment.

Structures, Unions, Classes, and Bitfields

The preceding issues of size, byte order, and alignment all affect the allocation of data in memory. In particular, compound data structures such as unions, structures, classes, bitfields, and arrays are very much affected by these issues.

Unions

A union allows the same memory location to be accessed as more than one type. This is inherently non-portable. Suppose a union consists of an integer and an array of four characters. Whether the first element of the array is the most or least significant part of the integer depends on byte order. It is not even certain that the integer and the array of characters have the same size.

These problems increase when a program combines integer, floating-point, and pointer fields, and are even more severe when unions consist of structures or bitfields.

Structures and Classes

Green Hills C/C++ compilers always allocate fields in a structure or class in the order in which the program declares them.

The exact offset of each field from the base of the structure/class depends on the size and alignment of the field itself and of those which precede it. The offset of the first field is always `0` in C, but not necessarily in C++ (for instance with multiple inheritance). Also, padding is inserted as necessary to satisfy the alignment requirement of each subsequent field, and may also be added at the end of the structure/class to make its overall size a multiple of its alignment.

Any program is non-portable if it assumes the offset of a field within a class/structure or if it assumes that certain fields in two different classes/structures always have the same offset.

Bitfields

The allocation of bitfields in a structure is dependent upon alignment rules. Additionally, the exact layout of bits within a bitfield varies between systems and cannot be assumed by a portable program.

Assumptions About Function Calling Conventions

Early implementations of C used a very straightforward approach to function calls. All parameters were pushed on the stack from right to left. All integral types smaller than `int` were promoted to `int` and `float` was promoted to `double`. Given this implementation, it was possible to write functions in C which handled variable parameters, even before the **<stdarg.h>** facilities. But such functions are non-portable and depend on an intimate knowledge of the calling conventions.

Many modern C/C++ compilers pass some parameters in registers and may not evaluate parameters from right to left. Integer and floating-point variables, not to mention structures, may have different rules. One non-portable assumption is that a `double` may be passed to a function which expects two integers. Not only does this assume a relationship between the size of the two types and a certain ordering of bytes and words, but it assumes that doubles and integers follow all of the same rules. Both of these assumptions are erroneous.

A much more common assumption is that pointer and integers may be interchanged when passing parameters. Neither C nor C++ guarantee that a pointer will be assigned to an integer and back without loss of information—even if the pointer and the integer are the same size. The only safe way to write a function that can correctly accept either pointer or integer parameters is to use the **<stdarg.h>** facilities.

Even among integral types, a program may assume that `int` and `long` are interchangeable. A C program can invoke **printf** using the **%d** operator to refer to a `long` parameter; however, this program will fail when ported to a system where `long` is larger than `int`. The correct way is to use **%ld** for `long` parameters.

The same portability problems exist with respect to function return values. A function which returns an `int` should never be used to return a pointer or `long` or floating-point value, even though it may work reliably on a particular system. A common mistake is to omit a declaration of a function that returns a pointer, and then place a cast around the invocation of that function. The cast cannot fix the error; it only prevents the compiler from reporting it.

Pointer Issues

Nearly all machines supported by Green Hills compilers are byte addressable, but neither C nor C++ require this. On some machines, a pointer to an `int` and a pointer to a `char` are not interchangeable. ANSI C requires that void pointers handle all pointer types, but the void pointer must be cast or assigned to its original type before being used. Similarly, C does not require that function pointers and data pointers be interchangeable, but some C programs incorrectly make this assumption.

C supports portable pointer arithmetic, provided it is used correctly. In ANSI C, the difference of two pointers is only defined for cases where both pointers refer to two elements within the same array object. Even so, in some unusual cases the difference may be outside the range of `ptrdiff_t` (which is either `int` or `long`). Subtraction or comparison of pointers to two separate objects may give non-portable results due to differences in memory layout or because pointers are signed on one system and unsigned on another.

Pointer arithmetic should always be done directly on the pointers when possible. If you need an integer type, use `uintptr_t` or `intptr_t`.

NULL Pointer

In all Green Hills C/C++ compilers, and most C compilers in general, the `NULL` pointer has the value `0`. But there are still two portability issues. One issue is that some older programs depend upon the contents of memory location `0` being `0`. This is now a well recognized programming error and some modern machines purposely give a memory fault for any attempt to read or write to location `0`.

A more subtle problem is the size of `NULL`. On a machine where pointers are larger than `int`, it is incorrect to use the constant 0 as a `NULL` pointer, because `0` is of type `int`, which is smaller than a pointer. This matters when passing `NULL` to a function that takes variable parameters or is not declared with a prototype.

Character Set Dependencies

Not all computer systems use the same characters. Although all computer systems recognize letters, digits, and the standard punctuation characters, considerable variation exists among the less commonly used characters. Therefore, programs which use the less common characters may not be portable.

Green Hills compilers use the ASCII character set and the ASCII collating sequence. Some language implementations use a different *collating sequence* , such as EBCDIC. Programs which manipulate character data, especially string sorting algorithms, may be dependent on a particular character collating sequence (the order in which characters are defined by the implementation). If one character appears before a second character in the collating sequence, then the first character will be considered smaller than the second character when they are compared. In the ASCII collating sequence, the lowercase letters `'a'` to `'z'` appear as the contiguous integer values 97 to 122 (decimal). In other collating sequences, such as EBCDIC, the lowercase letters are not contiguous.

To make character and string sorting programs portable, dependencies on the character collating sequence should be avoided. If a program is designed to operate with a collating sequence other than ASCII, it may be necessary to modify string and character comparison code in order to operate with ASCII.

Floating Point Range and Accuracy

The range, precision, accuracy, and base of floating-point arithmetic can vary widely between machines. This can lead to many portability problems that can only be addressed numerically. Green Hills compilers use the standard IEEE floating-point representation.

Operating System Dependencies

The file and I/O device naming conventions vary greatly among computer systems. Thus, programs that access operating system resources (such as files) by their system names are often not portable. In order to write portable programs, the use of explicit filenames in the program should be minimized. Preferably, these names can be input to the program when the program is run.

If a program contains explicit filenames it may be necessary to change them to names acceptable to the target system. Refer to your target operating system documentation for a description of legal filenames for that environment.

Assembly Language Interfaces

Programs which use embedded assembly code or interface to external assembly will require all of the assembly code to be rewritten when the program is transported to a new machine.

Evaluation Order

None of the language specifications fully describe the order in which the various components of an expression or statement must be evaluated. Additionally, the language specifications disallow computations whose results depend upon which permitted evaluation order is used. Many illegal programs have gone undetected because they have only been compiled with one compiler. Because the evaluation order may differ between compilers, some illegal programs may not operate as expected when compiled with a Green Hills compiler.

Some language implementations may evaluate the arguments to a function from right to left, others from left to right.

Green Hills compilers may execute expressions with side effects (such as subroutine, procedure, or function calls) in a different order than other compilers. When a variable is modified as a side effect of an expression, and its value is also used at another point in the expression, it is not specified whether the value used at either point in the expression is the value before or after modification. Different values for the same variable could potentially be used at different places in the expression, depending on the evaluation order observed by the compiler.

Green Hills compilers may also execute operators with side effects (such as ++, --, or +=) in a different order from that of other compilers.

Machine-Specific Arithmetic

Certain arithmetic operators in C/C++ are intended to generate the most efficient corresponding operation on the target machine. If all input values are within the expected range, the results are portable. Out of range values may give different results on different systems.

Shift

The shift operators in C/C++ possess machine-specific arithmetic characteristics. If the right-hand operand is negative or is greater than or equal to the number of bits in the promoted left-hand operand, the behavior is undefined and the results vary among compilers and hardware, and depending on whether the right hand operand is constant.

If the left-hand operand of a right shift is signed, C/C++ does not require the compiler to propagate the sign bit. While all Green Hills compilers propagate the sign bit, a conforming compiler is allowed to propagate zeros into the high bits.

Division

Division always satisfies the following rule:

```
(a / b) * b + a % b == a
```

When either `a` or `b` is negative, a C89 or C++ compiler is allowed to round `a / b` either toward zero or toward negative infinity. In C99, a compiler must always round toward zero. Green Hills always rounds toward zero, but a non-GHS compiler may round toward negative infinity.

Illegal Assumptions About Compiler Optimizations

Some programs illegally depend on the exact code generated by a particular compiler. Such programs are particularly difficult to port to an advanced optimizing compiler because the optimizer makes major changes in the code in order to reduce the program's size and increase the program's speed. This section describes some of the most common illegal assumptions made about code generation. The optimizations referenced in this chapter are described in greater detail in Chapter 4, “Optimizing Your Programs” on page 293.

Implied Register Usage

Some programs rely on the exact register allocation scheme used by the compiler. Such programs are illegal and will require modification before they can be transported.

For example, C/C++ programs that rely on register variables being allocated sequentially to pass hidden parameters will not work. Hidden returns (i.e., using `return` and expecting to return the value of the last evaluated expression) will not work either.

Memory Allocation Assumptions

Green Hills compilers allocate memory in a different manner than that of typical compilers. To ensure portability, programs compiled with a Green Hills compiler must not make assumptions regarding the order or allocation of variables in memory except where the language standard specifies it. If a program depends on the illegal memory allocation peculiarities of a particular compiler, it may experience portability problems. Note the following:

- Green Hills compilers do not necessarily allocate variables in the order of declaration.
- Green Hills compilers may not allocate unused variables.

Memory Optimization Restrictions

This section is very important when the system code or application uses shared memory or signals.

Using the **Advanced→Optimization Options→Memory Optimization** option (**-OM**) enables the compiler to assume that memory locations do not change asynchronously with respect to the running program (see "Memory Optimization" on page 327). In particular, when the compiler reads or writes to some memory location, it assumes that the same value remains there several instructions later. To avoid the (potentially high) speed penalties involved in re-reading memory, the compiler searches a register for a copy of the value to use instead.

If the `volatile` keyword is not used correctly, this option may cause problems for many parts of operating systems, device drivers, memory mapped I/O locations, shared memory environments, multiple process environments, and interrupt-driven routines, and when Linux/Solaris style signals are enabled.

An example of the potential problems involved with memory optimizations is that many Linux/Solaris device drivers need to use memory locations which are really

I/O registers that can change at any time. A typical example of a loop waiting for a device register to change is:

```
while (!(*TSRADDR & (1 << TXSBIT)));
```

If memory optimizations are enabled while compiling this loop, the compiler may generate code that reads the value pointed to by `TSRADDR` only once. If the **Advanced→Optimization Options→Optimize All Appropriate Loops** option (**-OL**) is also enabled, it is almost certain that this will be the case. When this happens, the loop executes either once or forever (depending on the value of the bit when it is first tested) and will be rendered either ineffective or fatal. Depending on the situation, the compiler may detect such loops and generate code that operates correctly even with **Advanced→Optimization Options→Memory Optimization** option (**-OM**) enabled. However, if the loop body were to test more than one bit at the same address, the compiler may distort the loop in an attempt to read memory as few times as possible.

The compiler assumes that the `volatile` type qualifier is used when it is available. This means that any **Optimization→Optimization Strategy** implies **Advanced→Optimization Options→Memory Optimization** in C++ or in the ANSI modes of C. If circumstances prevent the use of `volatile`, you should disable the **Advanced→Optimization Options→Memory Optimization** option (**-Onomemory**). Note that this will leave the other general optimizations enabled (see “Optimization Options” on page 145).

Problems with Source-Level Debuggers

Variable Allocation

When a variable is allocated to a register it will always reside in that register. However, because other variables may share the register, it may not always contain the current value of the variable. This may cause a source-level debugger to give incorrect results. If a variable is looked at outside the range of its use, the compiler may have temporarily allocated that register for some other purpose. The best strategy is to check results just after they are assigned, or when the current value is going to be used later. Near the end of a function, most of the local variables will no longer be in use, so it is more likely that the register has been re-allocated.

Optimizations May Interfere with Debugging Ability

Some optimizations may interfere with the Debugger's ability to look into a running program and provide meaningful information. Green Hills provides several optimization strategies so that you can choose the best balance between debugging ability and optimization for each stage of your project. For more information, see "Optimization Strategy" on page 145.

Problems with Compiler Memory Size

The compiler primarily uses memory for the program, static data structures, global declarations, parse trees, and generated machine code. Global declarations consist of the global constant, type, variable, and function declarations. Memory usage increases when a compilation includes large numbers of declarations. Even unused global declarations must be stored throughout the compilation. If memory size problems exist, reduce the size of the **include** files by including only the necessary declarations.

Basic blocks also require memory. Every possible branch creates a new block. Memory usage may increase on machine-generated programs with very large `switch` statements or with a very large number of small `if` statements.

The C and C++ compilers store the entire program into a parse tree before proceeding with optimization and code generation. The optimizer modifies the parse tree and then passes it on to the code generator. The code generator produces an internal representation of the machine code to be output for the function. Another optimization phase is then called to modify this machine code. Finally, the optimized machine code for the function is output. After the machine code is output, the memory being used for the parse tree, optimization, and machine code generation is released for use in processing the next function.

Usually, the size and complexity of the largest function in the program determine the maximum memory usage for parse trees and machine code. If memory size problems exist, turn off the optimizer and reduce the size or complexity of the relevant functions.

Chapter 18

Enhanced asm Macro Facility for C and C++

Contents

Although having the ability to write portable code is an advantage of using the C language, sometimes it is necessary to introduce machine-specific assembly language instructions into C code. This need arises most often within operating system code that must deal with hardware registers that would otherwise be inaccessible from C. You can use the `asm` facility to include this assembly code.

In earlier versions of C, the `asm` facility included a line that looked like a call to the function `asm` that took one argument, a string:

```
asm("assembly instruction here");
```

Unfortunately, this technique has shortcomings when the assembly instruction needs to reference C language operands. The user has to guess the register or stack location into which the compiler would put the operand and encode that location into the instruction. If the compiler's allocation scheme changes or, more likely, if the C code surrounding `asm()` changes, the correct location for the operand in `asm()` also changes. The user has to be aware that the C code would affect `asm()` and change it accordingly.

In addition, `asm()` statements that contain labels or other unique identifiers might cause errors if you enable certain optimizations. You may need to use temporary labels to prevent these errors (See “Labels” on page 376 for more information).

The new facility presented here is compatible with old code, since it retains the old capability. In addition, you can define `asm` macros that describe how machine instructions should be generated when their operands take particular forms that the compiler recognizes, such as register or stack variables.

Although this enhanced `asm` facility is easier to use than before, the user is still strongly discouraged from using it for routine applications, because those applications will not be portable to different machines. The `asm` facility helps implement operating systems in a clean way.

Optimizations may work incorrectly on C programs that use the `asm` macro facility, leading to compile-time or run-time errors. This is more likely when the `asm` macros contain instructions or labels that are unlike those that the C compiler generates. No compiler warning or error messages will be issued. Furthermore, the user may need to rewrite `asm` code in the future, in order to maximize its benefits as new optimization technology is introduced into the compilation system.

Wherever possible, consider using intrinsic functions instead of `asm` macros. `Asm` macros act as an optimization barrier to the compiler, since the compiler is unable to interpret the actions or understand the instructions in the `asm` macro. Intrinsic functions, on the other hand, are understood by the compiler, so the compiler need not make worst-case assumptions about the behavior of the instructions. Intrinsic functions are also easier to use since the compiler takes care of loading and storing the operands automatically, as necessary.

Definition of Terms

The following terms are used when describing the `asm` macro facility.

Term	Definition
`asm` macro	The mechanism by which programs use the enhanced `asm` facility. An `asm` macro has a definition and uses. The definition includes a set of pattern/body pairs. Each pattern describes the storage modes that the actual arguments must match for the `asm` macro body to be expanded. The uses resemble C function calls.
storage mode	The compiler's idea of where the argument can be found at run time. Examples are “in a register” or “in memory.”
pattern	Specifies the modes for each of the arguments of an `asm` macro. When the modes in the pattern all match those of the use, the corresponding body is expanded.
`asm` macro body	The portion of code that will be expanded by the compiler when the corresponding pattern matches. The body may contain references to the formal parameters, in which case the compiler substitutes the assembly language location for the corresponding parameter.

Using and Defining the asm Macro Facility

The following is a general example intended to demonstrate how to define and use an `asm` macro. For an example specific to V850 and RH850, see “V850 and RH850 asm Macros” on page 855.

Suppose your machine has an instruction that takes one operand, which must be in a register. It may be convenient to have a function that produces inline code to use this instruction, and works with register variables or constants.

This example consists of two parts: the definition of the `asm` macro and its use.

Pseudo-Code Definition

Define an `asm` macro, called `ASM`:

```
asm void ASM(newpri)
{
%reg newpri
      asm_instruction newpri
%con newpri
      asm_move newpri, temp_register
      asm_instruction temp_register
}
```

The lines that begin with `%` are patterns. If the arguments at the time the macro is called match the storage modes in a pattern, the code that follows the pattern line will expand.

Use

The table below shows the (assembly) code that the compiler will generate with two different uses of the assembly instruction. It uses the following introductory code (along with the above definition):

```
void f()
{
     register int i = 3;
     ASM(i);
     ASM(3);
}
```

Code	Matches	Generates
`ASM(i);`	`% reg`	*asm_instruction register_i_is_in*
`ASM(3);`	`% con`	*asm_move* newpri -> *temp_register* *asm_instruction temp_register*

The first use of `ASM` has a register variable as its argument (assuming that `i` actually gets allocated to a register). This argument has a storage mode that matches `reg`, the storage mode in the first pattern. Therefore, the compiler expands the first code body. `newpri`, the formal parameter in the definition, has been replaced in the expanded code by the compiler's idea of the assembly-time name for the variable

`i` (*register_i_is_in*). Similarly, the second use of `ASM` has a constant as its argument, which leads to the compiler's choosing the second pattern. Here again `newpri` has been replaced by the assembly-time form for the constant `3`.

Using asm Macros

You can use the enhanced `asm` facility to define `asm` macros. `asm` macros are constructs that behave syntactically like static C functions. Each `asm` macro has one definition and zero or more uses per source file. You must put the macro's definition in the file that uses the macro (or include the definition with `#include`). You can define the same `asm` macro multiple times (and differently) in several files.

The `asm` macro definition declares a return type for the macro code, specifies patterns for the formal parameters, and provides bodies of code to expand when the patterns match. When the compiler encounters an `asm` macro call, it replaces each use of a formal parameter with an assembly language location as it expands the code body. This constitutes an important difference between C functions and `asm` macros. In some cases, an `asm` macro can change the values of its arguments, whereas a C function can only change a copy of its arguments.

If you pass any of the following as an argument to an `asm` macro, the macro can read and modify that argument directly:

- Structures contained in registers.
- Scalar variables that the compiler would not promote if passed to a function without a prototype.

For all other types of arguments, the compiler creates a temporary variable and passes that variable to the `asm` macro instead. Whenever the compiler uses a temporary variable, modifying the value directly within the `asm` macro body does not affect the value of the original argument. These temporary variables are eligible for register allocation and are allocated to registers if the register allocator has sufficient physical registers available to it:

- Function names
- Structures contained in memory
- Structure and array elements
- Compound expressions

- Addresses
- Scalar variables that the compiler would promote if passed to a function without a prototype

Note

The compiler promotes the type of the scalar variable as if you were passing it to a function without a prototype in C. The compiler does this whether or not you specify the `asm` macro's parameter list in a prototype form. This means that the compiler promotes variables of type `unsigned char`, `signed char`, `signed short`, `unsigned short`, and `float`, and then passes them to the macro through temporary variables.

Calls to `asm` macros look exactly like normal C function calls. You can use them in expressions and they can return values. The arguments to an `asm` macro can be arbitrary expressions, except that they must not contain uses of the same or other `asm` macros. Return values are passed according to the target-specific calling convention.

The following pseudo-code shows how passing addresses works:

```
asm void make10(addr)
{
%reg addr
    asm_move 10 -> temp_register
    asm_store temp_register -> *addr
%nearmem addr
    asm_load addr -> temp_register_1
    asm_move 10 -> temp_register_2
    asm_store temp_register_2 -> *temp_register_1
%error
}
void func()
{
    int i, array[10];
    make10(&i);
    make10(array);
    make10(&array[3]);
}
```

Definition

The syntactic classes *type_specifier*, *identifier*, and *parameter_list* are presented in the style used in C-language compilers. A syntactic description enclosed in square brackets ([ ]) is optional, unless the right bracket is followed by *, which means "zero or more repetitions." For example:

```
asm [type_specifier]identifier ([parameter_list])
{
[storage_mode_specification_line
     asm_body] *
}
```

An asm macro consists of the keyword asm followed by what looks like a C function declaration. Inside the macro body there are zero or more pairs of *storage_mode_specification_lines* (patterns) and corresponding *asm_bodies*. If the *type_specifier* is other than void, the asm macro should return a value of the declared type.

storage_mode_specification_line

```
% [storage_mode [identifier [, identifier]* ] ]*
```

A *storage_mode_specification_line* consists of a single line (no continuation with the backslash character (\) is permitted) that begins with a percent sign (%) and contains the names (*identifiers*) and *storage modes* of the formal parameters. Modes for all formal parameters must be given in each *storage_mode_specification_line* (except for error). The percent sign (%) must be the first character on a line (including whitespace). If an asm macro has no *parameter_list*, the *storage_mode_specification_line* may be omitted.

Storage Modes

These are the storage modes that the compiler recognizes in asm macros:

Storage mode	Description
con	A compile-time constant.

Storage mode	Description
error	Generates a compiler error. You can use this mode to flag errors at compile time if no appropriate pattern exists for a set of actual arguments.
farsprel	A location on the stack that is too far away to be accessed in a single instruction.

lab	A compiler-generated unique label. The *identifier(s)* that are specified as being of mode lab do not appear as formal parameters in the asm macro definition, unlike the preceding modes. Such identifiers will be expanded to be unique.
	Example:
	`asm void check_for_bit_set(r)`
	`{`
	`%reg r %lab endlab`
	`  b_if_not_bit_set r, endlab`
	`  software_trap`
	`endlab:`
	`%error`
	`}`
	`void check_status(int i, int j)`
	`{`
	`  /* Using the macro twice would cause the "endlab" label to be`
	`   * defined twice if it was not specified as unique with %lab */`
	`  check_for_bit_set(i);`
	`  check_for_bit_set(j);`
	`}`

mem	Matches any allowed machine addressing mode, with the exception of reg and con.
farmem	A location in memory that cannot be accessed with a single load instruction. This may need to be broken up into multiple instructions, and the steps for doing this may depend on the compilation mode.
nearmem	A location in memory that can be accessed with a single load instruction (such as nearby stack variables or variables in a Small Data Area).
reg	A treg or ureg.
treg	A compiler-selected temporary register.
ureg	A C register variable that the compiler has allocated in a machine register.

asm Body

The asm body represents the (presumed) assembly code that the compiler will generate when the modes of all of the formal parameters match the associated pattern. Syntactically, the asm body consists of the text between two pattern lines that begin with a percent sign (%) or between the last pattern line and the right curly

brace (}) that ends the asm macro. C-language comment lines are not recognized as such in the asm body. Instead, they are simply considered part of the text to be expanded.

Formal parameter names may appear in any context in the asm body, delimited by non-alphanumeric characters. For each instance of a formal parameter in the asm body the compiler substitutes the appropriate assembly language operand syntax that will access the actual argument at run time. As an example, if one of the actual arguments to an asm macro is x, an automatic variable, a string like 4(%fp) would be substituted for occurrences of the corresponding formal parameter. An important consequence of this macro substitution behavior is that asm macros can change the value of their arguments. This differs from standard C semantics.

For lab parameters, a unique label is chosen for each new expansion.

If an asm macro is declared to return a value, it must be coded to return a value of the proper type in the machine register that is appropriate for the implementation.

An implementation restriction requires that no line in the asm body may start with a percent sign (%).

The V850 and RH850 compiler also supports the following primitive in the body of an asm statement:

%SPOFF(*m*) If *m* is of the farsprel storage class, this expands to an integer containing its offset from the stack pointer. For example:

```
%farsprel m
addi r3, sp, hi1(%SPOFF(m))
ld.w lo(%SPOFF(m)) [r3], r3
```

V850 and RH850 asm Macros

The following is an example of how to use asm macros for the V850 architecture. This example shows an operand multiplied by the value 10.

For this example, it is assumed that constants can be loaded in a single mov instruction; if this is not the case, the hi()/lo() pairs must be used for the entire construct. Alternatively, users can refrain from passing constants, and the %error

directive can be used to check for non-conformance. Notice that `%farsprel` must be handled before `%farmem`, since `%farmem` is a superset of `%farsprel`.

On the V850, any of the argument registers may be used as temporary (scratch) registers in an `asm` macro. `r10` is the return register, so the return value is placed there.

C preprocessor directives may not be used within an `asm` macro.

Also note that the number of possible cases for an `asm` macro goes up exponentially with the number of general arguments. `asm` macros are designed to be used in specific cases where the desired usage patterns are well understood.

```
asm int mult10(a) {
%reg a
 mov a, r10
 mulh 10, r10
%con a
 mov a, r10
 mulh 10, r10
%farsprel a
 movhi hi(%SPOFF(a)), sp, r6
 ld.w lo(%SPOFF(a))[r6], r10
 mulh 10, r10
%nearmem a
 ld.w a, r10
 mulh 10, r10
%farmem a
 movhi hi(a), r0, r6
 ld.w lo(a)[r6], r10
 mulh 10, r10
%error
}

int mem_var = 3;
void foo(int *proc) {
}
int main(void) {
 int stack_var = 4;
 int local_var = 5;
 foo(&stack_var); /* force stack_var on the stack */
 if ((mult10(mem_var) != 30) ||
  (mult10(stack_var) != 40) ||
```

```
  (mult10(local_var) != 50))
  printf("Asm macro error in %s\n", __FILE__);
 return 0;
}
```

Guidelines for Writing asm Macros

Here are some guidelines for writing `asm` macros.

- Use intrinsic functions preferentially over `asm` macros. Intrinsic functions do not inhibit optimizations the same way that `asm` macros may, and they are easier to use.
- Know the implementation. You must be familiar with the C compiler and assembly language with which you are working. You can consult the Application Binary Interface for your machine for the details of function calling and register usage conventions.
- Observe register conventions. An `asm` macro can alter temporary registers at will, but the permanent registers should not be written to directly, as the arguments to `asm` macros might be allocated to permanent registers. However, you may write to any formal parameter, regardless of whether the compiler automatically allocated it to a register. You must know in which register or registers the compiler returns function results.
- Handle return values. `asm` macros may “return” values. That means they behave as if they were actually functions that had been called via the usual function call mechanism. `asm` macros must therefore mimic C's behavior in that respect, returning values in the same way as normal C functions. Float and double results sometimes get returned in different registers from integer results. On some machine architectures, C functions return pointers in different registers from those used for scalars. Finally, structures may be returned in a variety of implementation-dependent ways.
- Cover all cases. The `asm` macro patterns should cover all combinations of storage modes of the parameters. The minimal set that covers all cases is `reg`, `con`, and `mem`. The compiler attempts to match patterns in the order of their appearance in the `asm` macro definition. If the compiler encounters a storage mode of `error` while attempting to find a matching pattern, it generates a compile-time error for that particular `asm` macro call.

- Beware of argument handling. `asm` macro arguments are used for macro substitution. Thus, unlike normal C functions, `asm` macros can alter the underlying values to which their arguments refer. Altering argument values makes it impossible to substitute an equivalent C function call for the `asm` macro call.
- `asm` macros are inherently non-portable and implementation-dependent. Although they make it easier to introduce assembly code reliably into C code, the process cannot be made foolproof. You will always need to verify correct behavior by inspection and testing.
- Debuggers will generally have difficulty with `asm` macros. You cannot set source-level breakpoints in the `asm` body.
- Make sure that storage mode specification lines begin with `%` in the first column. There must not be any whitespace before this character.

Chapter 19

GNU Extended Assembly

Contents

GNU extended `asm` is supported through the option **--gnu_asm**.

Please see the online manual GCC Inline Assembly HowTo [http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html] for generic information concerning basic and extended assembly syntax.

V850 and RH850 Extended GNU Support

The option **--gnu_asm** is enabled by default in V850 and RH850.

The list of target-independent constraints provided by the compiler is:

- `'X'`: Unconstrained.
- `'g'`: `'r'` or `'i'` or `'m'`.
- `'0'`: Matching 0th constraint.
- `'1'`: Matching 1st constraint.
- `'2'`: Matching 2nd constraint.
- `'3'`: Matching 3rd constraint.
- `'4'`: Matching 4th constraint.
- `'5'`: Matching 5th constraint.
- `'6'`: Matching 6th constraint.
- `'7'`: Matching 7th constraint.
- `'8'`: Matching 8th constraint.
- `'9'`: Matching 9th constraint.
- `'r'`: Any integer register.
- `'f'`: Any float register.
- `'m'`: Any memory location.
- `'p'`: Any memory location that is valid for a load/store.
- `'o'`: Any memory location that is offsettable.
- `'V'`: Any memory location that is non-offsettable.
- `'<'`: Any memory that is incremented before or after the operation.
- `'>'`: Any memory that is decremented before or after the operation.

- `'i'`: Any immediate integer (including symbolic references).
- `'n'`: Any immediate integer (excluding symbolic references).
- `'s'`: Any immediate symbolic references.
- `'F'`: Any floating point constant..

In addition to the target-independent, two new constraints have been defined for V850 and RH850:

- `'R'`: Even numbered general purpose register.
- `'v'`: Vector register.

Besides the constraints, the constraint modifiers provide finer control over the effect of the constraints. The list of available constraint modifiers is:

- `'='`: The operand is written.
- `'+'`: The operand is both read and written.
- `'&'`: The operand is an early clobber operand, which is modified before the instruction is finished using the input operands.

Extended GNU inline assembly is particularly useful when inlining in C, assembly instructions that modify more than one register.

For instance, the SIMD Vector data copying instruction (in its `M-type` addressing form), available for RH850:

```
VST.DW vreg3, [reg1]%, reg2
```

can be inlined in C using extended GNU assembly syntax in the following way:

```
void vstdw_mod(int *address, long long value, long long modop,
    long long *retmodop)
{
    asm ("vst.dw %2, [%3]%, %1" : "+m" (address), "+R"(modop):
     "v"(value), "r"(address));
    *retmodop = modop;
}
```

In the above example (that has to be built with the SIMD support option **-rh850_simd**) the compiler will use the constraint `'R'` to allocate the variable

`modop` in an even register, and the constraint `'v'` to allocate the variable `value` in a vector register, as required by the instruction.

Chapter 20

The ELF File Format

Contents

This chapter explains the formats of ELF (Executable and Linking Format) files. Sections of this chapter have been reproduced with permission from UNIX System Laboratories, Inc. For more information about ELF files, see *System V Application Binary Interface*, 1993, UNIX System Laboratories, Inc., published by Prentice-Hall, Inc.

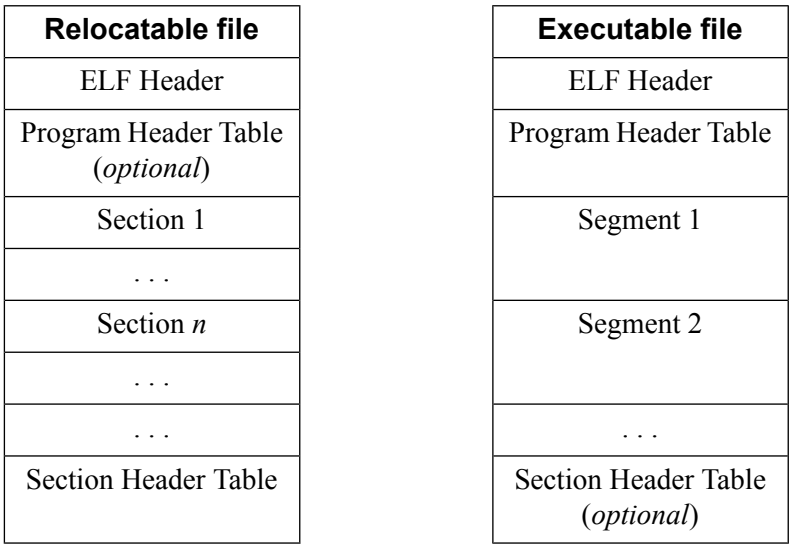
This chapter deals only with the 32-bit ELF format. Programs and object files built for 64-bit targets use the 64-bit ELF format, which is similar to the 32-bit format, but uses larger fields to store some values. For information about the 64-bit ELF format, see the Wikipedia reference page [http://en.wikipedia.org/wiki/Executable_and_Linkable_Format].

Formats of Relocatable and Executable Files

ELF object files are the standard output of the compiler, assembler, and linker. An ELF file can be either of the following:

- a *relocatable object file* — which contains program code and data and is suitable for linking with other object files.
- an *executable file* — which contains a program suitable for execution.

The following diagram shows the differences in format between these kinds of ELF files:



Both forms of ELF file begin with an *ELF Header*, which serves as the table of contents. All other data and tables in the file may appear in any order.

An executable file must have a *Program Header Table*, which tells the system how to load the program. This component is optional in a relocatable file.

A relocatable file requires a *Section Header Table*, containing information about its *sections* in order to be subsequently linked. Each section has an entry in the table, giving information such as the section's name and size. Sections form the majority of the relocatable object file, and comprise such information as instructions, data, symbol table, and relocation information.

32-Bit ELF Data Types

The ELF file format supports 32-bit architectures with 8-bit bytes. It must be modified for 64-bit architectures. ELF files represent some control data with a machine-independent format to identify object files and interpret their contents. Part of an ELF file uses the encoding of the target processor, regardless of the machine on which the file was created:

Type name	Size	Alignment	Purpose
`Elf32_Addr`	4	4	Unsigned program address
`Elf32_Half`	2	2	Unsigned medium integer
`Elf32_Off`	4	4	Unsigned file offset
`Elf32_Sword`	4	4	Signed large integer
`Elf32_Word`	4	4	Unsigned large integer
`unsigned char`	1	1	Unsigned small integer

All data structures that the ELF file format defines follow the usual size and alignment guidelines for the relevant class. If necessary, data structures contain explicit padding to ensure 4-byte alignment for 4-byte objects in order to force structure sizes to a multiple of 4. Data also have suitable alignment from the beginning of the file. For example, a structure containing `Elf32_Addr` will be aligned on a 4-byte boundary within the file.

For portability, ELF data structures use no bitfields.

ELF Headers

Some ELF file control structures can grow because the ELF header contains their actual sizes. If the ELF file format changes, a program may encounter control structures that are larger or smaller than expected. An ELF header is set by the following C structure declaration:

```
#define EI_NIDENT 16
typedef struct {
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half    e_type;
    Elf32_Half    e_machine;
    Elf32_Word    e_version;
    Elf32_Addr    e_entry;
    Elf32_Off     e_phoff;
    Elf32_Off     e_shoff;
    Elf32_Word    e_flags;
    Elf32_Half    e_ehsize;
    Elf32_Half    e_phentsize;
    Elf32_Half    e_phnum;
    Elf32_Half    e_shentsize;
    Elf32_Half    e_shnum;
    Elf32_Half    e_shstrndx;
} Elf32_Ehdr;
```

The fields of `struct Elf32_Ehdr` are as follows.

`e_ident`

Identifies the file as an ELF file and provides machine-independent data to decode the contents. For more information, see “ELF Identification” on page 869.

`e_type`

Identifies the ELF file type. Permitted values are as follows:

- `0` (`ET_NONE`): No file type
- `1` (`ET_REL`): Relocatable file
- `2` (`ET_EXEC`): Executable file
- `0xFF00` (`ET_LOPROC`): Begin processor-specific range
- `0xFFFF` (`ET_HIPROC`): End processor-specific range

`e_machine`

Specifies the target architecture for an individual file. Permitted values are as follows:

- `0` (`EM_NONE`): Invalid machine
- `2` (`EM_SPARC`): Sun SPARC
- `3` (`EM_386`): Intel 80386
- `4` (`EM_68K`): Freescale 68000
- `6` (`EM_486`): Intel 80486
- `8` (`EM_MIPS`): MIPS
- `19` (`EM_960`): Intel i960
- `20` (`EM_PPC`): Power Architecture
- `21` (`EM_PPC64`): Power Architecture 64-bit mode
- `36` (`EM_V800`): Renesas V800 series
- `37` (`EM_FR20`): Fujitsu FR
- `39` (`EM_MCORE`): Freescale MCore
- `40` (`EM_ARM`): ARM
- `42` (`EM_SH`): Hitachi SH
- `44` (`EM_TRICORE`): Infineon TriCore
- `52` (`EM_COLDFIRE`): Freescale ColdFire
- `58` (`EM_STARCORE`): Freescale StarCore
- `78` (`EM_FIREPATH`): Broadcom FirePath
- `88` (`EM_M32R`): Renesas M32R
- `106` (`EM_BLACKFIN`): ADI Blackfin

`e_version`

Identifies the ELF file version. Permitted values are as follows:

- `0` (`EV_NONE`): Invalid version
- `1` (`EV_CURRENT`): Current version

The value `1` (one) signifies the original file format; extensions will create new versions with higher numbers.

`e_entry`

Specifies the virtual address to which the system first transfers control upon starting the process. If the file has no associated entry point, this field contains the value zero.

`e_phoff` Specifies the program header table's file offset in bytes. If the file has no program header table, this field contains the value zero.
`e_shoff` Specifies the section header table's file offset in bytes. If the file has no section header table, this field contains the value zero.
`e_flags` Specifies processor-specific flags associated with the file. Flag names take the form `EF_`*`machine_flag`*. • `0x20000000` (`EF_V800_850`): V850 processor • `0x10000000` (`EF_V800_850E`): V850E processor • `0x04000000` (`EF_V800_850F`): V850E1F processor • `0x02000000` (`EF_V800_850E2`): V850E2 processor • `0x00000800` (`EF_V800_850E_16`): V850E2 processor
`e_ehsize` Specifies the ELF header size in bytes.
`e_phentsize` Specifies the size in bytes of one entry in the file's program header table; all entries are the same size.
`e_phnum` Specifies the number of entries in the program header table. The product of `e_phentsize` and `e_phnum` specifies the table's size in bytes. If a file has no program header table, `e_phnum` contains the value zero.
`e_shentsize` Specifies the section header's size in bytes. A section header is one entry in the section header table; all entries are the same size.
`e_shnum` Specifies the number of entries in the section header table. The product of `e_shentsize` and `e_shnum` specifies the section header table's size in bytes. If a file has no section header table, this field contains the value zero.
`e_shstrndx` Specifies the index of the section containing the string table used for section names. By convention, this is the section `.shstrtab`. If the file has no section name string table, this field contains the value `SHN_UNDEF`.

ELF Identification

ELF provides a file framework to support multiple processors, multiple data encodings, and multiple classes of machines. To support this file family, the initial bytes of the file specify how to interpret the file, independent of the processor on which the inquiry is made and independent of the file's remaining contents.

The initial bytes of an ELF header correspond to the `e_ident` member. The identification indexes are tabulated below:

Index name	Value	Meaning
`EI_MAG0` to `EI_MAG3`	`0` to `3`	File identification
`EI_CLASS`	`4`	File class
`EI_DATA`	`5`	Data encoding
`EI_VERSION`	`6`	File version
`EI_PAD`	`7`	Start of padding bytes
`EI_NIDENT`	`16`	Size of `e_ident[]`

These indexes access bytes that contain the following values:

`EI_MAG0` to `EI_MAG3`

A file's first 4 bytes identify the file as ELF as follows:

- `e_ident[EI_MAG0]`: `0x7F` (`ELFMAG0`)
- `e_ident[EI_MAG1]`: `'E'` (`ELFMAG1`)
- `e_ident[EI_MAG2]`: `'L'` (`ELFMAG2`)
- `e_ident[EI_MAG3]`: `'F'` (`ELFMAG3`)

EI_CLASS This byte identifies the file's class, or capacity. The file format is portable among machines of various sizes; it does not impose the size of the largest machine on the smallest. Permitted values are as follows: • 0 (ELFCLASSNONE): Invalid class • 1 (ELFCLASS32): 32-bit objects. Supports machines with files and virtual address spaces up to 4 Gigabytes. It uses the basic types defined above. • 2 (ELFCLASS64): 64-bit objects. Reserved for 64-bit architectures. Its appearance here shows how the file format may change, but the 64-bit format is otherwise unspecified. Other classes are defined as necessary, with different basic types and sizes for file data.
EI_DATA This byte specifies the data encoding of the processor-specific data in the file. Permitted values are as follows: • 0 (ELFDATANONE): Invalid data encoding • 1 (ELFDATA2LSB): Specifies 2's complement values, with the least significant byte occupying the lowest address. • 2 (ELFDATA2MSB): Specifies 2's complement values, with the most significant byte occupying the lowest address. Other values are reserved and are assigned to new encodings as necessary.
EI_VERSION This byte specifies the ELF header version number. Currently, this value must be EV_CURRENT, as explained above for e_version.
EI_PAD This value marks the beginning of the unused bytes in e_ident. These bytes are reserved and set to zero; programs that read ELF files should ignore them. The value of EI_PAD changes if currently unused bytes are given meanings.
EI_NIDENT This value specifies the size of e_ident[].

ELF Sections

Section Headers

You can use an ELF file's section header table to locate all the file's sections. The section header table is an array of Elf32_Shdr structures. A section header table index is a subscript into this array. The ELF header `e_shoff` specifies the byte offset from the beginning of the file to the section header table; `e_shnum` tells how many entries the section header table contains; and `e_shentsize` specifies the size in bytes of each entry.

Sections contain all information in a file except the ELF header, the program header table, and the section header table. Sections satisfy several conditions:

- Every section in an ELF file has exactly one section header describing it. Section headers may exist that do not have an associated section.
- Each section occupies one contiguous (possibly empty) sequence of bytes within a file.
- Sections in a file may not overlap. No byte in a file resides in more than one section.
- An ELF file may have inactive space. The various headers and the sections might not account for every byte in a file. The contents of the inactive space are unspecified.

A section header has the following structure:

```
typedef struct {
    Elf32_Word sh_name;
    Elf32_Word sh_type;
    Elf32_Word sh_flags;
    Elf32_Addr sh_addr;
    Elf32_Off  sh_offset;
    Elf32_Word sh_size;
    Elf32_Word sh_link;
    Elf32_Word sh_info;
    Elf32_Word sh_addralign;
    Elf32_Word sh_entsize;
} Elf32_Shdr;
```

The fields of struct Elf32_Shdr are as follows.

`sh_name` Specifies the name of the section. Its value is an index into the section header string table section and specifies the location of a null-terminated string.
`sh_type` Specifies the section's contents and semantics. For more information, see “Section Attribute Flags” on page 874.
`sh_flags` Specifies 1-bit flags that describe miscellaneous attributes of the section.
`sh_addr` Specifies the address at which the section's first byte should reside if the section appears in the memory image of a process. Otherwise, this field contains zero.
`sh_offset` Specifies the byte offset from the beginning of the file to the first byte in the section.
`sh_size` Specifies the section's size in bytes. Unless the section type is `SHT_NOBITS`, the section occupies `sh_size` bytes in the file. A section of type `SHT_NOBITS` may have a non-zero size, but it occupies no space in the file.
`sh_link` Contains a section header table index link, whose interpretation depends on the section type.
`sh_info` Contains extra information, whose interpretation depends on the section type.
`sh_addralign` Some sections have address alignment constraints. For example, if a section contains a doubleword, the system may need to ensure doubleword alignment for the entire section. That is, the value of `sh_addr` must be equal to `0` (zero), modulo the value of `sh_addralign`. Currently, only `0` (zero) and positive integral powers of two are allowed. The values `0` (zero) and `1` (one) mean the section has no alignment constraints.
`sh_entsize` Some sections contain a table of fixed-size entries, such as a symbol table. For such sections, this field specifies the size in bytes of each entry. This structure contains zero if the section does not contain a table of fixed-sized entries.

Special Section Indexes

Symbol table entries index the section table through the `st_shndx` field. See "Symbol Tables" on page 880.

Name	Value	Meaning
`SHN_UNDEF`	`0`	A meaningless section. A symbol "defined" relative to this section is an undefined symbol.
`SHN_COMMON`	`-14`	Common, or `.bss`, symbols are allocated space in this section.
`SHN_ABS`	`-15`	Contents of the section are absolute values; they are not affected by relocation.
`SHN_GHS_SMALLCOMMON`	`-256`	Not available for all processors. Similar to `SHN_COMMON`, but for a limited number of small variables allocated to SDA.
`SHN_GHS_ZERO_COMMON`	`-255`	Not available for all processors. Similar to `SHN_COMMON`, but for a limited number of small variables allocated to ZDA.
`SHN_GHS_TINYCOMMON`	`-254`	Not available for all processors. Similar to `SHN_COMMON`, but for a limited number of small variables allocated to TDA.

Note

Although index `0` is reserved as an undefined value, the section header table does contain an entry for it.

Section Types

A section header's `sh_type` specifies the section's semantics as follows:

Name	Value	Meaning
`SHT_NULL`	`0`	Marks the section header as inactive. It does not have an associated section. Other members of the section header have undefined values.
`SHT_PROGBITS`	`1`	Contains information defined by the program that created the ELF file, whose format and meaning are determined solely by the program.

Name	Value	Meaning
SHT_SYMTAB	2	Contains a symbol table. An object file may have only one section of this type, but this restriction may be relaxed in the future. The table provides symbols for link editing, though it may also be used for dynamic linking. As a complete symbol table, it may contain many symbols unnecessary for dynamic linking.
SHT_STRTAB	3	Contains a string table. A file may have multiple string table sections.
SHT_RELA	4	Contains relocation entries with explicit addends, such as type Elf32_Rela for the 32-bit class of ELF files. A file may have multiple relocation sections.
SHT_NOTE	7	Contains notes that flag this ELF file indicating certain properties.
SHT_NOBITS	8	Occupies no space in the file but otherwise resembles SHT_PROGBITS.
SHT_REL	9	Contains relocation entries without explicit addends, such as type Elf32_Rel for the 32-bit class of ELF files. A file may have multiple relocation sections.

Section Attribute Flags

A section header's sh_flags field contains 1-bit flags that describe the section's attributes.

Name	Value	Meaning
SHF_WRITE	0x1	Contains data that should be writable during process execution.
SHF_ALLOC	0x2	Occupies memory during process execution. Some control sections do not reside in the memory image of the final application; this attribute is off for those sections.
SHF_EXECINSTR	0x4	Contains execution machine instructions.
SHF_GHS_ABS	0x400	Has an absolute, non-relocatable address.

Two structures in the section header, sh_link and sh_info, contain special information, depending on section type as follows:

sh_type	sh_link	sh_info
SHT_REL SHT_RELA	The section header index of the associated symbol table.	The section header index of the section to which the relocation applies.
SHT_SYMTAB	The section header index of the associated string table.	One greater than the symbol table index of the last local symbol (binding STB_LOCAL).
other	SHN_UNDEF	0

Section Names

Section names beginning with a period (.) are not meant for general use by application programs, although application programs may use these sections if their existing meanings are satisfactory. Application programs may use names without the leading period to avoid conflicts with predefined section names.

Frequently Used Sections

Some sections contain program and control information. Sections in the following list have predefined meanings, with the indicated types and attributes.

.bss Contains uninitialized data that contributes to the program's memory image. The system initializes the data with zeros when the program begins to run. Type: SHT_NOBITS Attributes: SHF_ALLOC | SHF_WRITE
.data Contains initialized data that contributes to the program's memory image. Type: SHT_PROGBITS Attributes: SHF_ALLOC | SHF_WRITE

`.linfix` When using Code Factoring or fixing "out of range relocations" inline, the assembly becomes out of sync with the debugging information. The linker notes the changes to the program that require updates to the debugging information in the `.linfix` section. MULTI merges the information in the `.dla` file and the `.linfix` section when debugging the program. The `.linfix` section is not downloaded to the target. Type: `SHT_PROGBITS` Attributes: None
`.rel`*`name`* `.rela`*`name`* These sections contain relocation information. Conventionally, *name* is supplied by the section to which the relocations apply. A relocation section for `.text` has the name `.rel.text` or `.rela.text`. Types: `SHT_REL` and `SHT_RELA` Attributes: none
`.rodata` Read-only data area. Similar to the `.data` section, but composed of constant data. Type: `SHT_PROGBITS` Attributes: `SHF_ALLOC`
`.shstrtab` Contains section names. Type: `SHT_STRTAB` Attributes: None
`.strtab` Contains strings, most commonly the strings that represent the names associated with symbol table entries. Type: `SHT_STRTAB` Attributes: none
`.symtab` Contains a symbol table. Type: `SHT_SYMTAB` Attributes: none

`.text` Contains the “text”, or executable instructions, of a program. Type: `SHT_PROGBITS` Attributes: `SHF_ALLOC | SHF_EXECINSTR`

Some sections are specific to Green Hills ELF; these are listed in the following table. Target processors may support some or all of these sections.

`.syscall` A program code section to support the Green Hills system call mechanism. Type: `SHT_PROGBITS` Attributes: `SHF_EXECINSTR | SHF_ALLOC`
`.secinfo` A table, created by the linker, that describes actions to be taken on sections as they are loaded for program execution (sections to be cleared, copied from ROM to RAM, and so on). Type: `SHT_PROGBITS` Attributes: `SHF_ALLOC`

`.fixaddr`
`.fixtype`
Tables created by the compiler for position independent code (PIC) and position independent data (PID) static pointer adjustments to be made when the program is loaded for execution.
Type: `SHT_PROGBITS`
Attributes: `SHF_ALLOC`

`.sdabase` If not in PID mode, the run-time system initializes the Small Data Area (SDA) base register to be the address of this section (`.sdabase`). Type: `SHT_NULL` Attributes: `SHF_ALLOC`

.sdata Initialized Small Data Area .zdata Initialized Zero Data Area. Similar to the .data section, but of limited size and more efficiently addressed. Type: SHT_PROGBITS Attributes: SHF_ALLOC | SHF_WRITE
.sbss Uninitialized Small Data Area. .zbss Uninitialized Zero Data Area. Similar to the .bss section, but of limited size and more efficiently addressed. Type: SHT_PROGBITS Attributes: SHF_ALLOC | SHF_WRITE
.heap Section describing the area of memory for dynamic allocations through malloc and related functions. Type: SHT_NOBITS Attributes: SHF_ALLOC | SHF_WRITE
.stack Section describing the area of memory that the program stack will occupy. Type: SHT_NOBITS Attributes: SHF_ALLOC | SHF_WRITE

Relocation Types

Relocation entries contain this information. There are two forms of relocation entries, corresponding to SHT_REL and SHT_RELA sections. Relocations can have offsets called addends from the symbols being referenced by the ELF32_R_SYM() information in r_info. In SHT_REL, the addends are encoded in the instruction or data fields of the location referenced by the r_offset field. In SHT_RELA, each relocation entry contains a dedicated r_addend field. The SHT_RELA format is preferred because it simplifies the interpretation of relocations, as well as allows offsets that are greater than the fields can contain.

Example 20.1. A Relocation Entry Corresponding to SHT_REL

```
typedef struct {
    Elf32_Addr r_offset;
    Elf32_Word r_info;
} Elf32_Rel;
```

Example 20.2. A Relocation Entry Corresponding to SHT_RELA

```
typedef struct {
    Elf32_Addr r_offset;
    Elf32_Word r_info;
    Elf32_Sword r_addend;
} Elf32_Rela;
```

`r_offset` Specifies the location at which to apply the relocation action. For a relocatable file, the value is the byte offset from the beginning of the section to the storage unit affected by the relocation. For an executable file or a shared object, the value of which is the virtual address of the storage unit affected by the relocation.

`r_info`
Specifies both the symbol table index with respect to which the relocation must be made and the type of relocation to apply information for a process's program image. For example, a call instruction's relocation entry contains the symbol table index of the function being called. You may find the following macros helpful when reading from or writing to the `r_info` field:
`#define ELF32_R_SYM(i)   ((i) >> 8)`
`#define ELF32_R_TYPE(i)   ((unsigned char) (i))`
`#define ELF32_R_INFO(s,t) (((s)<<8) + (unsigned char) (t))`

`r_addend` Specifies a constant value used to compute the final value to be stored into the relocatable field.

A relocation section references two other sections: a symbol table and a section to modify. The section header's `sh_link` and `sh_info` specify these sections.

Symbol Tables

The symbol table of an ELF file contains information to locate and relocate a program's symbolic definitions and references. A symbol table index is a subscript into this array. Index `0` (zero) both designates the first entry in the table and serves as the undefined symbol index. The contents of the initial entry are specified below. A symbol table entry has the following format:

```
typedef struct {
    Elf32_Word    st_name;
    Elf32_Addr    st_value;
    Elf32_Word    st_size;
    unsigned char st_info;
    unsigned char st_other;
    Elf32_Half    st_shndx;
} Elf32_Sym;
```

where:

`st_name`

Contains an index into the file's symbol string table. The string table contains the character representations of the symbol names. If this field is non-zero, then it represents a string table index that specifies the symbol name. Otherwise, the symbol table entry has no name.

`st_value`

Specifies the associated symbol's value. Depending on the context, this may be an absolute value, an address, and so on.

`st_size`

Many symbols have associated sizes. For example, a data object's size is the number of bytes contained in the object. This field is zero if the symbol has no size or an unknown size.

`st_info`

Specifies the symbol's binding attributes and type, as explained in the symbol binding and symbol type sections below. The values and meanings are defined below:

```
#define ELF32_ST_BIND(i)   ((i)>>4)
#define ELF32_ST_TYPE(i)   ((i)&0xf)
#define ELF32_ST_INFO(b,t)   (((b)<<4)+((t)&0xf))
```

`st_other`

Contains the value zero and has no defined meaning.

st_shndx
Every symbol table entry is "defined" in relation to some section; this field contains the relevant section header table index. Some section indexes indicate special meanings. For more information, see "Special Section Indexes" on page 873.

Symbol Binding

A symbol's binding determines the linkage visibility and behavior.

Name	Value	Attributes
STB_LOCAL	0	Local symbols are not visible outside the object file containing their definition. Local symbols of the same name may exist in multiple files without interfering with each other.
STB_GLOBAL	1	Global symbols are visible to all object files being combined. One file's definition of a global symbol will satisfy another file's undefined reference to the same global symbol.
STB_WEAK	2	Weak symbols resemble global symbols, but their definitions have lower precedence.
STB_LOPROC	13	Values in this inclusive range (from STB_LOPROC to STB_HIPROC) are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.
STB_HIPROC	15	

Symbol Type

A symbol's type provides a general classification for the associated entity.

Name	Value	Attributes
STT_NOTYPE	0	The symbol's type is not specified.
STT_OBJECT	1	The symbol is associated with a data object, such as a variable, array, and so on.
STT_FUNC	2	The symbol is associated with a function or other executable.
STT_SECTION	3	The symbol is associated with a section. Symbol table entries of this type exist primarily for relocation and normally have STB_LOCAL binding.
STT_FILE	4	By convention, this symbol names the source file associated with the object file. For this symbol, the binding is STB_LOCAL and the section index is SHN_ABS. All STT_FILE symbols precede any other STB_LOCAL symbols.

Name	Value	Attributes
STT_LOPROC	13	Values in this inclusive range (from STT_LOPROC to STT_HIPROC) are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.
STT_HIPROC	15	

Symbol Values

Symbol table entries for different ELF file types have slightly different interpretations for the st_value member:

- In relocatable files, st_value contains alignment constraints for a symbol whose section index is SHN_COMMON.
- In relocatable files, st_value contains a section offset for a defined symbol. That is, st_value is an offset from the beginning of the section that st_shndx identifies.
- In executable files, st_value contains a virtual address. To make these symbols more useful for the dynamic linker, the section offset (file interpretation) gives way to a virtual address (memory interpretation) for which the section number is irrelevant.

Although the symbol table values have similar meanings for different file types, the information allows efficient access by the appropriate programs.

String Tables

String table sections contain null-terminated character sequences or strings. These strings are used to represent symbol and section names. An empty string table section is permitted; its section header's `sh_size` contains zero. Non-zero indexes are invalid for an empty string table. If the string table is not empty, you can reference a string as an index into the string table section. The first byte, which is index `0` (zero), contains a null character. Likewise, a string table's last byte contains a null character to ensure null termination for all strings. A string whose index is `0` (zero) specifies either no name or a null name, depending on the context.

A section header's `sh_name` contains an index into the section header string table (See the `e_shstrndx` field in "ELF Headers" on page 866). The following figures show a string table with 25 bytes and the strings associated with various indexes:

Index	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	.	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

The string table indexes are:

Index	String
0	*none*
1	`name.`
7	`Variable`
11	`able`
16	`able`
24	*null string*

A string table index may refer to any byte in the section. A string may appear more than once; references to substrings may exist, and a single string may be referenced multiple times. Strings that are not referenced are also allowed.

Program Headers

An ELF executable file's program header table is an array of structures, each describing a segment or other information the system needs to prepare the program for execution. A file segment contains one or more sections. Program headers are defined only for executable files. A file specifies its own program header size with the ELF header's `e_phentsize` and `e_phnum`.

```
typedef struct {
    Elf32_Word p_type;
    Elf32_Off  p_offset;
    Elf32_Addr p_vaddr;
    Elf32_Addr p_paddr;
    Elf32_Word p_filesz;
    Elf32_Word p_memsz;
    Elf32_Word p_flags;
    Elf32_Word p_align;
} Elf32_Phdr;
```

`p_type` The kind of segment this array element describes and how to interpret the array element's information. For more information, see “Program Types” on page 885.
`p_offset` Offset from the beginning of the file at which the first byte of the segment resides.
`p_vaddr` Address at which the first byte of the segment resides in memory.
`p_paddr` For targets where physical addressing is relevant, this field gives the segment's physical address. For other targets this field is unused and is set to zero by the linker.
`p_filesz` Number of bytes in the file image of the segment; it may be zero.
`p_memsz` Number of bytes in the memory image of the segment; it may be zero.
`p_flags` Flags relevant to the segment. For more information, see “Program Attribute Flags” on page 886.

p_align

Loadable process segments must have congruent values for p_vaddr and p_offset, modulo the page size. This field specifies the value to which the segments are aligned in memory and in the file. Values 0 and 1 mean no alignment is required. Otherwise, p_align should be a positive, integral power of 2, and p_vaddr should equal p_offset, modulo p_align.

Program Types

A program header's p_type specifies the program's semantics as follows:

Name	Value	Meaning
PT_NULL	0	The array element is unused; other member values are undefined. This type lets the program header table have ignored entries.
PT_LOAD	1	The array element specifies a loadable segment, described by p_filesz and p_memsz. The bytes from the file are mapped to the beginning of the memory segment. If the segment's memory size (p_memsz) is larger than the file size (p_filesz), and p_filesz is non-zero, the “extra” bytes are defined to hold the value 0 and to follow the segment's initialized area. The file size may not be larger than the memory size. Loadable segment entries in the program header table appear in ascending order, sorted on the p_vaddr member.
PT_DYNAMIC	2	The array element specifies dynamic linking information.
PT_INTERP	3	The array element specifies the location and size of a null-terminated pathname to invoke as an interpreter. This segment type is meaningful only for executable files (though it may occur for shared objects); it may not occur more than once in a file. If it is present, it must precede any loadable segment entry.
PT_NOTE	4	The array element specifies the location and size of auxiliary information.
PT_SHLIB	5	This segment type is reserved but has unspecified semantics.

Name	Value	Meaning
PT_PHDR	6	The array element, if present, specifies the location and size of the program header table itself, both in the file and in the memory image of the program. This segment type may not occur more than once in a file. Moreover, it may occur only if the program header table is part of the memory image of the program. If it is present, it must precede any loadable segment entry.
PT_LOPROC to PT_HIPROC	0x70000000 to 0x7fffffff	Reserved for processor-specific semantics.

Program Attribute Flags

A program to be loaded by the system must have at least one loadable segment (although this is not required by the file format). When the system creates loadable segment memory images, it gives access permissions as specified in the p_flags member. All bits included in the PF_MASKPROC mask are reserved for processor-specific semantics.

Name	Value	Meaning
PF_X	0x1	Execute
PF_W	0x2	Write
PF_R	0x4	Read
PF_MASKPROC	0xf0000000	Unspecified

If a permission bit is 0, that type of access is denied. Actual memory permissions depend on the memory management unit, which may vary from one system to another. Although all flag combinations are valid, the system may grant more access than requested. However, a segment will never have write permission unless it is specified explicitly. The following table shows both the exact flag interpretation and the allowable flag interpretation.

Flag	Value	Exact	Allowable
(none)	0	All access denied	All access denied
PF_X	1	Execute only	Read, execute

Flag	Value	Exact	Allowable
PF_W	2	Write only	Read, write, execute
PF_W+PF_X	3	Write, execute	Read, write, execute
PF_R	4	Read only	Read, execute
PF_R+PF_X	5	Read, execute	Read, execute
PF_R+PF_W	6	Read, write	Read, write, execute
PF_R+PF_W+PF_X	7	Read, write, execute	Read, write, execute

For example, typical text segments have read and execute - but not write - permissions. Data segments normally have read, write, and execute permissions.

Part IV

Appendices

Appendix A

Coding Standard Profile Reference

Contents

This appendix describes the preset coding standard profiles that ship with the Green Hills Tools. The tables in this appendix display:

- each implemented rule
- the diagnostics associated with that rule
- tags associated with that rule

If the standard provides different levels of implementation (such as “required” and “advisory”), additional columns provide a tag name for and associate rules with each level.

For more information about using coding standard profiles, see Chapter 14, “Coding Standards” on page 723.

GHS Standard Mode

The entire set of diagnostics is controlled by the tag `ghstd_all`.

Declarations, Definitions, and Initialization

Rule	Diag.	Tags
Function definitions and declarations must have explicit return types.	77 260 837 938	
The type of each returned expression must match the return type of the enclosing function.	117 940	
Function prototypes are required.	223 1547 1791	
Function parameters and return types in declarations must match the function prototype.	1844	
Functions must be declared at file scope. Defining a function inside another function is not permitted.	1731	
Functions with no parameters must be declared with the void parameter type.	1728	
Multiple definitions of an external identifier are not permitted (no commons).	1797	

Rule	Diag.	Tags
Declaring an object or function in the same file as both static and extern is not permitted.	172	
Non-standard unnamed fields and bitfields are not permitted.	619 620	
One-bit signed bitfields are not permitted.	108	
Bitfields of unspecified signedness must be strictly narrower than their underlying types.	1927	
Automatic variables must be set before they are used.	549	
Initializers must have explicit braces.	991 1162 1736	
String initializers that discard the final '\0' are not permitted.	1908	
Shadowing an identifier with another identifier in a narrower scope is not permitted.	460 1721	
Enumeration values must be in the range of "signed int".	66	
Variables declared "register" should have appropriate types.	1912 1913	

Expressions and Control Flow

Rule	Diag.	Tags
Side effects in the operand of the sizeof operator are not permitted.	1725	
The sizeof operator's operand must be a type, a variable, or a dereferenced pointer expression.	1931	
Shifting a value by a negative number or by more than the size of its actual type is not permitted.	63	
Unary minus on unsigned expressions is not permitted.	1702	
Assignments in controlling expressions are not permitted.	187	
Testing floating-point values for exact equality or inequality is not permitted.	1704	
Floating-point loop counters are not permitted.	1750	
All non-null statements must have at least one side effect.	174	
Top-level expressions must not contain useless code.	1910	
Each label must be followed by a statement.	1963	

Rule	Diag.	Tags
Falling through a case clause in a switch statement is not permitted.	1711 1838	
Each switch statement must contain a final default case.	1712 1837	
Each switch statement must contain at least one case.	1713	
The controlling expression of a switch statement must not be constant.	237	
The controlling expression of a switch statement must not be effectively Boolean.	1733	
Relational operators must be correctly parenthesized.	1892	
The address of an automatic variable must not escape its scope.	1056 1780	
Casts to or from pointers to incomplete class types are not permitted.	1915	
Boolean expressions and non-booleans should not be compared for equality.	1937	
Implicit integer conversions should not result in truncation.	69	

Preprocessing Directives

Rule	Diag.	Tags
Nested comments are not permitted.	9	
Unrecognized preprocessing directives are not permitted.	11	
Use of #define or #undef within a block is not permitted.	1732	

Character Set Portability

Rule	Diag.	Tags
Nonstandard or multi-byte characters are not permitted in #include directives.	1747	
Non-standard escape sequences are not permitted.	192	
Trigraphs are not permitted in line splices, character constants, or string literals.	1967	

Standard Libraries

Rule	Diag.	Tags
printf and scanf calls must have an argument of the correct type for each format specifier.	181 224 225 226	
printf and scanf format strings must be string literals, not variables which might be tainted by user input.	1932	
Use of setjmp and longjmp is not permitted.	1906	
Use of atof, atoi, or atol is not permitted.	1907	
Use of gets or scanf("%s") is not permitted.	1899 1900	

MISRA C 1998

This coding standard is provided as an example implementation of MISRA C 1998. It is not a full implementation. In addition to the tags listed in the tables below, the entire set of diagnostics is controlled by the tag `m98_all`.

Rule		Diag.	m98_req	m98_adv	Others
1	All code shall conform to ISO 9899 standard C, with no extensions permitted.	1546 667	X		`m98_1`
3	Assembly language functions should contain only assembly language.	1749 1839		X	`m98_3`
5	Only those characters defined in the ISO C standard shall be used.	192 1823	X		`m98_5`
7	Trigraphs shall not be used.	1695	X		`m98_7`
8	Multibyte characters and wide string literals shall not be used.	1720 1823	X		`m98_8`
9	Comments shall not be nested.	9	X		`m98_9`
10	Sections of code should not be commented out.	1771		X	`m98_10`
12	No identifier in one namespace shall match an identifier in another.	1723		X	`m98_12`
13	The basic types should not be used.	1698		X	`m98_13`

Rule		Diag.	m98_req	m98_adv	Others
14	The type char shall always be declared as unsigned char or signed char.	1699	X		`m98_14`
16	The underlying bit representations of floats shall not be used in any way.	1820	X		`m98_16`
17	typedef names shall not be reused.	1722 1841	X		`m98_17`
18	Numeric constants should be suffixed to indicate type.	1775		X	`m98_18`
19	Octal constants (other than zero) shall not be used.	1718	X		`m98_19`
20	All object and function identifiers shall be declared before use.	223	X		`m98_20`
21	Identifiers in an inner scope shall not hide identifiers in outer scopes.	460 1721	X		`m98_21`
24	Identifiers shall not simultaneously have internal and external linkage.	172	X		`m98_24`
25	An identifier with external linkage shall have exactly one definition.	1797	X		`m98_25`
26	If objects or functions are declared more than once...	147	X		`m98_26`
28	The register storage class should not be used.	1696		X	`m98_28`
29	The use of a tag shall agree with its declaration.	188 469 1842	X		`m98_29`
30	All automatic variables shall have been assigned a value before use.	549	X		`m98_30`
31	Braces shall be used to indicate structure in array and struct initializers.	146 991 1162	X		`m98_31`
32	In an enumerator list, the '=' construct shall not be used other than ...	1726	X		`m98_32`
33	The right hand operand of && or || shall not contain side effects.	1724	X		`m98_33`
34	The operands of && or || shall be primary expressions.	1729	X		`m98_34`

Rule		Diag.	m98_req	m98_adv	Others
35	Assignment operators shall not be used in expressions which return bool.	1738	X		m98_35
36	Logical operators should not be confused with bitwise operators.	1853 1854		X	m98_36
37	Bitwise operations shall not be performed on signed integer types.	1730	X		m98_37
38	The right hand operand of a shift shall lie between zero and width-1.	62 63	X		m98_38
39	The unary minus operator shall not be applied to an unsigned expression.	1702	X		m98_39
40	The sizeof operator should not be used on expressions with side effects.	1725		X	m98_40
42	The comma operator shall not be used, except in for loop headers.	1719	X		m98_42
43	Implicit conversions which may result in a loss of information ...	1739 1863 1864 1865 1866 1867 1868 1869 1870	X		m98_43
44	Redundant explicit casts should not be used.	1740		X	m98_44
45	Type casting from any type to or from pointers shall not be used.	1703	X		m98_45
47	No dependence should be placed on C's operator precedence rules.	1737		X	m98_47
48	Mixed precision arithmetic should use explicit casting.	1751		X	m98_48
49	Tests of a value against zero should be made explicit.	1855 1881		X	m98_49
50	Floating point variables shall not be tested for exact equality.	1704	X		m98_50
51	Evaluation of constant unsigned integer expressions should not wrap around.	1735		X	m98_51
52	There shall be no unreachable code.	111	X		m98_52

Rule		Diag.	m98_req	m98_adv	Others
53	All non-null statements shall have a side-effect.	174	X		`m98_53`
54	A null statement shall only occur on a line by itself.	1746	X		`m98_54`
55	Labels should not be used, except in switch statements.	1708		X	`m98_55`
56	The goto statement shall not be used.	1705	X		`m98_56`
57	The continue statement shall not be used.	1706	X		`m98_57`
58	The break statement shall not be used (except in a switch statement).	1707	X		`m98_58`
59	The body of an if, while, do, or for shall always be enclosed in braces.	1709 1826	X		`m98_59`
60	All if, else if constructs should contain a final else clause.	1710		X	`m98_60`
61	Every non-empty case clause in a switch statement shall have a break.	1711 1884	X		`m98_61`, `m98_switch`
62	All switch statements should contain a final default clause.	1712 1837	X		`m98_62`, `m98_switch`
63	A switch expression should not represent a Boolean value.	1733 1944		X	`m98_63`, `m98_switch`
64	Every switch statement shall have at least one case.	1713	X		`m98_64`, `m98_switch`
65	Floating point variables shall not be used as loop counters.	1750	X		`m98_65`
68	Functions shall always be declared at file scope.	1731	X		`m98_68`
69	Functions with variable numbers of arguments shall not be used.	1697	X		`m98_69`
71	Functions shall always have visible prototype declarations.	1547 1791 1800 1828	X		`m98_71`
72	The type given in the declaration and definition shall be identical.	1844	X		`m98_72`
73	Identifiers shall either be given for all the parameters, or for none.	1727	X		`m98_73`

Rule		Diag.	m98_req	m98_adv	Others
74	If identifiers are given, then those in the definition shall be identical.	1745	X		m98_74
75	Every function shall have an explicit return type.	77 260 837 938	X		m98_75
76	Functions with no parameters shall be declared with parameter type void.	1728	X		m98_76
78	The number of parameters passed shall match the function prototype.	140 165	X		m98_78
82	A function should have a single point of exit.	1734		X	m98_82
83	Each return shall have an expression; it shall match the return type.	117 940	X		m98_83
85	Functions called with no parameters should have empty parentheses.	1774		X	m98_85
87	#include statements shall only be preceded by other directives or comments.	1748	X		m98_87
88	Non-standard characters shall not occur in header file names.	1747	X		m98_88
90	C macros shall only be used for ...	1776 1859 1860 1886	X		m98_90
91	Macros shall not be #define'd and #undef'd within a block.	1732	X		m98_91
92	#undef should not be used.	1715		X	m98_92
93	A function should be used in preference to a function-like macro.	1862		X	m98_93
94	A function-like macro shall not be 'called' without its arguments.	54 76	X		m98_94
95	Arguments to a macro shall not contain preprocessing directives.	10	X		m98_95
96	The whole definition, and each parameter, shall be enclosed in parentheses.	1743 1829	X		m98_96
97	Identifiers in preprocessor directives should be defined before use.	193		X	m98_97

Rule		Diag.	m98_req	m98_adv	Others
98	There shall be at most one # or ## operator in a single macro definition.	1716	X		`m98_98`
100	The defined preprocessor operator shall only be used in standard forms.	1831 1902	X		`m98_100`
101	Pointer arithmetic should not be used.	1752		X	`m98_101`
102	No more than 2 levels of pointer indirection should be used.	1741		X	`m98_102`
103	Relational operators shall not be applied to pointer types except ...	1961	X		`m98_103`
104	Non-constant pointers to functions shall not be used.	1777	X		`m98_104`
105	All the functions pointed to by a function pointer shall have the same type.	1778	X		`m98_105`
106	The address of an object shall not be assigned to an outer-scoped object.	1056 1780	X		`m98_106`
110	Unions shall not be used to access the sub-parts of larger data types.	1819 1827	X		`m98_110`
111	Bit fields shall only be defined of type unsigned int or signed int.	230 1717	X		`m98_111`
112	Bit fields of type signed int shall be at least 2 bits long.	108	X		`m98_112`
113	All the members of a structure (or union) shall be named.	1742	X		`m98_113`
114	Reserved words and function names shall not be redefined or undefined.	45 46 1861 1885 1888	X		`m98_114`
122	The setjmp macro and the longjmp function shall not be used.	1906	X		`m98_122`
125	The library functions atof, atoi and atol from <stdlib.h> shall not be used.	1907	X		`m98_125`

Appendix B

GNU Options

The builder currently accepts the following legacy GNU option names as aliases for official Green Hills options. These option aliases are deprecated. We recommend that you use the official Green Hills Software options.

Legacy GNU Option	Official Green Hills Option
-###	-#
-fcommon	--commons
-fexceptions	--exceptions
-fpeephole	-Opeephole
-fschedule-insns	-Opipeline
-fshort-enums	--short_enum
-fstrict-aliasing	-ansi_alias
-fsyntax-only	-syntax
-funroll-loops	-Ounroll
-fvolatile	-OM
-fno-common	--no_commons
-fno-exceptions	--no_exceptions
-fno-peephole	-Onopeephole
-fno-schedule-insns	-Onopipeline
-fno-short-enums	--no_short_enum
-fno-strict-aliasing	-no_ansi_alias
-fno-volatile	-Onomemory
-fno-unroll-loops	-Onounroll
-mapp-regs	-no_reserve_r2
-march=	-cpu=
-mbig -mbig-endian -EB	-bigendian
-mbigswitch	-bigswitch
-mcpu	-cpu=
-mdisable-callt	
-mlong-calls	-farcalls

-mno-app-regs	**-reserve_r2**
-mnobigswitch	**-nobigswitch**
-mno-disable-callt	
-mno-long-calls	**-nofarcalls**
-msda=	**-sda=**
-msdata=none	**-sda=0**
-mv850	**-cpu=v850**
-mv850e	**-cpu=v850e**
-mzda=	**-zda=**

Note

1. Of the options listed in the previous table, only **-EB**, **-EL**, **-mlong-calls**, and **-mno-long-calls** are recognized by the Builder. The remaining options are recognized only by the driver.

Appendix C

Green Hills Project File Format

Contents

Green Hills project (**.gpj**) files support several advanced methods for specifying the files and options to use when building your project with the MULTI Project Manager or command line build utility. This appendix describes the file format and covers advanced features that you can use only by manually editing project files.

Green Hills Project File Syntax

Green Hills project (**.gpj**) files have three sections:

- Header section — Specifies the project file's type. For Top Project files, also specifies project-wide directives.
- Options section — Specifies options that apply to the project and all of its children.
- Children section — Specifies files included by the project, their types, and options that apply to each respective file.

In any of these sections:

- You can add comments anywhere after the first line of a project file with the hash symbol ('`#`'). Comments continue to the end of the line.
- You can use the value of a macro or imported environment variable by preceding the name of the macro or variable with the dollar sign ('`$`'). For example, to use the value of the macro `my_library` as part of the **-l** option, type the following:

  ```
  -l$my_library
  ```

The Header Section

The header is the first section in a project file. The first line in a project file (and therefore the first line of the header) must be `#!gbuild`. The last line in the header is the project type, enclosed in brackets ('`[]`'). For a list of types, see the Project Files (**.gpj**) list in “File Types” on page 915.

For example, a header for a simple program project might look like this:

```
#!gbuild
# My Hello World Program
[Program]
```

The header section may also contain macro definitions in the following format: **macro** *macro_name=value*

There must not be any spaces between *macro_name*, =, and *value*. For example:

```
#!gbuild
# Macros for the project
macro DIR=my_project_directory
macro DEBUG=1
[Program]
```

If you override the value of a macro in a child project, the new value applies only to uses of the macro in the child and its descendants. For example, if you use a macro to specify an option in a project, that exact option setting is inherited by children even if the children override the macro.

Top Project headers must specify a `primaryTarget`, and may specify any number of additional Top Project directives. For more information about these directives, see "Top Project Directives" on page 909. For example:

```
#!gbuild
# My Top Project
# Additional Top Project directives go here
primaryTarget=v800_standalone.tgt
[Project]
```

The Project Options Section

The project options section lists options that apply to the project you are editing and all of its children. Each line must be indented by at least one whitespace character. You can put multiple options on the same line, separated by white spaces. If the option value contains spaces, enclose it in double quotes (`""`). You may need to escape backslashes inside quoted strings. The following example builds on the previous Top Project example:

```
#!gbuild
# Header section
primaryTarget=v800_standalone.tgt
```

```
[Project]

# Options section
    -bsp generic            # Use the generic BSP
    -G                      # Generate debug information
    -object_dir="my objs"   # Put object files in ./my objs
    -DDEBUG -DNOISY         # Two preprocessor defines
```

The Children Section

The children section lists all files that are children of the project you are editing. The filenames are not indented, and may use full or relative paths. If you use a relative path, the Builder tries to resolve the file relative to the current project file's directory, and then relative to each directory specified by applicable `:sourceDir` options (see “Source Directories Relative to This File” on page 143). If the filename contains spaces, enclose it in double quotes (`""`). You may need to escape backslashes inside quoted strings.

Tip

If you plan to open your project on hosts that run different operating systems, always use forward slashes (`/`) when specifying paths or filenames.

Each line that specifies a child file must specify the file's type in brackets (`[]`), unless it is the default type for that file's extension. For a list of defaults, see “File Types” on page 915. If you do not specify a type and there is no default, it is treated as a text (**.txt**) file and is not processed during a build. For example:

```
# Children section
src_hello/hello.gpj  [Program] # Override default [Subproject]
tgt/resources.gpj    [Project] # Override default [Subproject]
docs/readme_one.txt            # Extension sets type to [Text]
docs/readme_two.odd            # No default, treated as [Text]
```

The type you specify for any child project must match the type in that project's header. For example, if your Top Project includes a child program:

```
#!gbuild
# My Top Project default.gpj
primaryTarget=v800_standalone.tgt
```

```
[Project]
src_hello/hello.gpj         [Program] # The child program
```

The child must have `[Program]` as the file type in its header:

```
#!gbuild
# My Program src_hello/hello.gpj
[Program]                   # The type specified in the parent
```

Any indented, non-commented lines immediately following a child entry are treated as a child options section, which lists options that apply to that child and all of its children. These options do not apply to the project file you are editing or any of the child's siblings. Child option sections have the same syntax as the project options section. For example, if you used the following project file, **foo.o** would be placed in **objs\special**, while **hello.o** would be placed in **objs\hello**:

```
#!gbuild
[Program]

# Options Section
    -object_dir=objs/hello

# Children Section
foo.c
    # Child options for foo.c
    -object_dir=objs/special
hello.c
tgt/standalone_ram.ld
```

Top Project Directives

Top Project directives are directives that specify settings for your entire project. You can only use them in your Top Project. There are two groups of these directives:

- Group 1 directives
- Group 2 directives

All group 1 directives must appear before group 2 directives, and all Top Project directives must appear before the project type line (usually `[Project]`). To add a Top Project directive:

1. Edit your Top Project. Type the directive above the line that reads `[Project]`.
2. Save and close the file.
3. Reload the project in the Project Manager.

The following tables list available directives.

Group 1 Directives

All of these directives are optional.

defaultBuildOptions= *option ...*

Takes a space-separated list of **gbuild** options and applies them to this project and any subprojects, as if you had passed them on the command line. For example, if you add the line:

```
defaultBuildOptions=-all
```

to your Top Project, every file is rebuilt each time you build your project.

For a list of options you can use with this directive, see "The gbuild Utility Program" on page 509.

gbuildDir= *directory*

Specifies a relative or absolute path to the Compiler installation directory that the MULTI Project Manager uses to build this project. Relative pathnames are resolved relative to the location of the Top Project file.

This option is useful if you want to build a particular project using a different version of the Compiler than the one that the MULTI IDE is linked with. For example, if you recently upgraded and linked MULTI to a newer Compiler, but you still want to build a particular project with your old Compiler, you can use this option. Overriding the Compiler installation directory in this way only changes the tools used to build the project; it does not override any other MULTI IDE functionality that uses the Compiler installation. For example, the Help Viewer does not display manuals from the specified Compiler installation directory, it displays manuals from the linked Compiler installation directory.

Using the MULTI IDE to debug programs that are built with this option is only supported if:

1. The specified Compiler installation directory contains a version of the Compiler that is compatible with the version of the MULTI IDE used.
2. Both the specified Compiler installation directory and the Compiler installation directory that is linked with the MULTI IDE support the target specified for the project.

This option only affects projects built from the Project Manager; it is ignored by the **gbuild** utility program.

import *variable_name* Imports the environment variable named *variable_name* from the local environment.
macro *macro_name=value* Defines a new macro with the given name and value. There must not be any spaces between *macro_name*, =, and *value*.

Group 2 Directives

customization= *customization_file* This optional directive specifies a customization (**.bod**) file for your project. You can use this directive multiple times. For more information about customization files, see Appendix D, "Customization Files" on page 919.
primaryTarget= *target_file* This required directive specifies the target file for your project. This is one of the **.tgt** files included in the MULTI distribution that contains all of the target-specific settings necessary to compile code for your target.

Conditional Control Statements

Conditional control statements reside in the project file and are used to indicate to the Builder the portion of the project tree and options used in the build process. These control statements are especially useful when one portion of the project tree is built in multiple ways (for example, multiple targets). Each statement is evaluated during the build using information from the current file in the project tree and all of its parents. If the statement evaluates to true, the option or file is included in the build. If the statement evaluates to false, the option or file is silently skipped.

Some rules for using conditional control statements are:

- You can only specify these statements by manually editing the **.gpj** files.
- Conditional control statements can be used with options or filenames. The syntax is `{condition}` before the option or filename. For example:

```
{VERSION>5} ver5.c
     {isdefined(DEBUG)} -DDEBUG
```

- You can specify more than one conditional control statement. All statements must evaluate to true for the file to be included in the build. For example:

```
{optional}{isdefined(SUPPORTS_DSP)} dsp_subsystem.gpj
```

Note

Conditional control statements applied to options must have a whitespace before them in order to function properly.

Conditional Control Statement Syntax

The following are valid conditional control statements:

command=*command_name*   The option or file is included when building the project if the current file is processed using the specified command.   *command_name* must be the same as the `command` property specified in the `FileType` section of the **.bod** file (see “Components of Customization Files” on page 920).
comment   The file is never included when building the project, and is disabled in the Project Manager. **comment** applies to source or project files, but not options.
filetype=*filetype*   The option is included when building the project if the current file is of the specified type. *filetype* must match the `name` property specified in the `FileType` section of the **.bod** file (see “File Type Definitions” on page 921).   **filetype=*filetype*** is only useful for options that are inherited by multiple types.

expression

The option or file is included when building the project if the expression evaluates to `true`. The expression can include macros, unsigned integers, the following functions, standard Boolean operators, and standard comparison operators. If you pass a string to one of the following functions and that string begins with a numerical digit, you must put quotes (`"`) around the string.

The following are accepted functions:

- `isdefined(`*`macro_name`*`)` — Evaluates to `true` if *`macro_name`* is currently defined, and `false` otherwise.
- `isundefined(`*`macro_name`*`)` — Evaluates to `true` if *`macro_name`* is not currently defined, and `false` otherwise.
- `istarget(`*`target_name`*`)` — Evaluates to `true` if *`target_name`* is equal to the name of the project target, and `false` otherwise. The string *`target_name`* corresponds to the base name of the **.tgt** file defined as the `primaryTarget` in your Top Project. The comparison is case-insensitive, and *target_name* must not include the **.tgt** extension when used in this conditional expression. For example, if the target is **v800_standalone.tgt**, then `istarget("v800_standalone")` will evaluate to `true`.
- `streq(`*`macro_name`*`, `*`string`*`, ...)` — Evaluates to `true` if the string value of *`macro_name`* is equal to that of one or more of the provided *string* arguments, and `false` otherwise. The comparison is case-sensitive.
- `strprefix(`*`macro_name`*`, `*`string`*`, ...)` — Evaluates to `true` if one or more of the provided *string* arguments are a prefix of the string value of *macro_name*, and `false` otherwise. The comparison is case-sensitive.

Note: Macros used in a macro expression must be valid C identifiers.

The following are accepted operators:

- Standard Boolean operators — `!`, `&&`, and `||`
- Standard comparison operators — >=, <=, >, <, == and `!=`

For example:

```
{ isdefined(DEBUG_LEVEL) && DEBUG_LEVEL > 0 } -G
```

nobuild

The file is never included when building the project, but is still included in the Project Manager. **nobuild** applies to source or project files, but not options.

optgroup=*option_group*

The option or file is included if the command used to process the current file supports the specified option group. If you have two different tools that support the same option string, but the behavior of the option is different for each program, you should consider using this option in your Project file.

For example, if you have a Green Hills Compiler and a custom IDL compiler that both support the **-r** option, and you want to pass this option to every file compiled by the IDL compiler, you might add the following line to your Top Project:

```
{optgroup=MyIDLCompilerOptions} -r
```

For more information about option groups and configuration files, see "Components of Customization Files" on page 920.

optional

When applied to a file, the current file is only included when building the project if the file exists.

When applied to an option, the option is only included when building the project if the option is supported by the current target.

pass= [COMPILE] [LINK] [DEPEND] [FIRSTPASS] [ALL]

Causes the controlled option to only be used during the specified processing pass. **pass=** only applies to options.

rebuild

Causes the current file to be rebuilt every time the project is built. **rebuild** only applies to source files (not **.gpj** files or options).

Special Conditional Control Statements

The following are special conditional control statements that can only be applied to *Select One* **.gpj** files. These control statements do not selectively include or exclude the project that they are applied to. Instead, they control the printing of warning and errors when processing Select One project files.

selectone_optional

If an assembly file corresponding to the current target or a C file is not specified, the project is silently ignored. This behavior is similar to the effect of the **{optional}** condition when applied to files.

selectone_required

If an assembly file corresponding to the current target or a C file is not specified, an error is generated, which stops the build.

Macro Functions

Macro functions allow you to define a macro that does not have a static value. Instead, the value is determined by evaluating a function. You can place them wherever you could place a standard macro. To place a macro function, use the following syntax:

```
${macro_function}
```

Macro functions can contain references to standard macros. The available macro functions are:

`%expand_path(`*`relative_path`*`)`
Expands *relative path* to an absolute path. *relative_path* is resolved relative to the directory that contains the Top Project. For example:
`macro SCRIPTS=${%expand_path(../scripts)}`

`%option_value(`*`option_name`*`)`
Given a name of an option, this function returns the option's set value. This function only supports options that take string arguments.
option_name must match the `name` parameter of the option as defined in the **.bod** file, and must include the dash or other punctuation that normally accompanies the option name (for example, `-os_dir`). Do not quote the option name.
This function searches the entire set of options that apply to the build item in which it is used. If the specified option is not set, or if the option does not take a string argument, this function returns an empty string and issues a warning.
For example:
`-DTHE_OS_DIR=${%option_value(-os_dir)}`

For more information about macros, see the documentation about setting build macros in the *MULTI: Managing Projects and Configuring the IDE* book.

File Types

You may encounter the following types in your project files:

- Project Files (**.gpj**):

- `Auto Include` — Pulls into a project the contents, or subset of the contents, of a specific directory. This file type cannot specify any children, and must contain the **:autoInclude** option. While this file behaves similarly to a project file, it has the **.auto** extension. For more information about `Auto Include`, see the documentation about auto include in the *MULTI: Managing Projects and Configuring the IDE* book.
- `Library` — A project file to output a compiled library.
- `Merged Library` — A project file that allows you to combine multiple high-level source files, assembly files, libraries and objects into a single library file. **Relocatable Object** files are not supported. For more information, see "Creating a Library from the Compiler Driver" on page 478.
- `Program` — A project file to output an executable.
- `Project` — General project file. These general purpose containers can also be used to group `Programs` and `Libraries`.
- `Reference` — A file type that allows you to import a project subtree anywhere in the build hierarchy without forcing it to inherit the options of the project that it is included in. A `Reference` project file must begin with the `[Reference]` tag, include a single **:reference** option (see "Reference" on page 263), and must not have any children. When you build your project, the subtree is considered a child of the `Reference` project, but it does not inherit the `Reference` project's options or those of its parents.
- `Relocatable Object` — A project file to output a relocatable object containing the compiled code from one or more source files.
- `Select One` — A project file that contains multiple source files, only one that is used in the build. These projects are used for multiple-architecture hierarchies (see the documentation about building platform specific programs from the same source files in the *MULTI: Managing Projects and Configuring the IDE* book).
- `Shared Object` — A project file to output a shared library. These projects are only available for some target operating systems.
- `Singleton Library` — A project file to output a library containing a single object file. The library is forced into the final executable.

  - `Subproject` — A project file to group files within a project, but which does not output anything itself. `Subprojects` differ from `Projects` in that files contained within them are included in the link of their parent project. This is the default file type for files with a **.gpj** extension.
  - `INTEGRITY Application` — A project file that contains one or more virtual AddressSpaces. It outputs an image for dynamic download, or it outputs a stand-alone executable with a kernel, depending on options.
- Source and Object Files:
  - `Assembly` (**.s**, **.asm**, ***.target***) — Assembly language source files. For the value of *`target`*, see "Recognized Input File Types" on page 8.
  - `C` (**.c**) — C source files.
  - `C++` (**.cc**, **.cpp**, **.cxx**, **.c++**, **.C**, **.CXX**, **.CPP**) — C++ source files.
  - `Header` (**.h**, **.hh**, **.H**, **.h++**, **.hxx**, **.hpp**) — C and C++ header files.
  - `Linker Raw File` — A file containing raw data that the linker imports into the final program.
  - `Object` (**.o**) — Object files.
  - `Prebuilt Library` (**.a**) — Library file of object modules. These are usually used to specify binary libraries when there is no source available.
- Target Resource Files:
  - `Board Setup` (**.mbs**, **.dbs**) — Setup script file for initializing your target board.
  - `Linker Directives` (**.ld**, **.lnk**) — Linker directives file that contains a memory map for the target board and a section map for the program. For more information about the file format, see "Configuring the Linker with Linker Directives Files" on page 441.
  - `Target Connections` (**.con**) — Target configuration information for connecting through a debug server (see the documentation about connecting to your target in the *MULTI: Debugging* book).
  - `Script` (**.rc**, **.irc**, **.prc**) — MULTI command scripts.
- INTEGRITY files:
  - `BSP Description` (**.bsp**)

    — Target-specific description file used by the **Integrate** utility.

  - `Integrate file` (**.int**) — Configuration file for specifying the existence of and relationships between various kernel objects.
- Documentation:
  - `Portable Document Format` (**.pdf**) — Adobe PDF files.
  - `Text` (**.txt**) — Generic text files. This type is used by default for unclassified file extensions.

Appendix D

Customization Files

Contents

Components of Customization Files

Customization files, or **.bod** files, are used to customize your build environment by defining acceptable input file types and how to build them. Customization files contain three types of definitions:

- `FileTypes` — Definitions of types of files that can be built or included in project files.
- `Commands` — Definitions of the commands used to build or edit the various file types.
- `CommandOptions` — Definitions of the options supported by the commands defined in the `Commands` section. These definitions are broken into option groups, where each group represents the options supported by one or more commands.

The following sections describe the purpose of each type of definition and its syntax by using the custom file type demo project available from the Project Manager. For information about creating a project including this example, see the documentation about creating a project in the *MULTI: Managing Projects and Configuring the IDE* book.

Note

If you are writing a customization file for use with both Windows and Linux/Solaris hosts, we recommend that you use forward slashes (/) when specifying any pathnames.

Note

The Green Hills tools use **.bod** files to define the file types, commands, and options available by default. These **.bod** files are a useful resource as you create your own customization file. The **.bod** files can be found in the **defaults/bld_rules** subdirectory of your MULTI installation. The file **ghs_filetypes.bod** contains file type and command definitions. The files **ghs_*_options.bod** contain the command option definitions. Use these files for reference only; do not modify them.

Customization File Definitions

The following section will provide you with definitions used in **.bod** files, as well as examples for creating a customized file.

File Type Definitions

The `FileTypes` section of a customization file defines additional types of files that can be built as part of your project. The definitions in this section provide the Green Hills tools with information about how to identify and process these files.

```
FileTypes {
# FileTypes defines our custom types and how to
# fill in the command arguments

    BinInput {
    # Here we've created a new custom type called "Binary Input"
        name="Binary Input"
        extensions={"bin"} # The input extensions are .bin.
        # It outputs .c files (used for -clean).
        outputExtension="c"
        # It outputs source files instead of object files
        outputType="SourceFile"
        # Do not perform grep operations on this input.
        grepable=false
        # Use the matching command from the Commands section below.
        command="GHS Binary Converter"
        # Specifies the exact command line using
        # variables to substitute for context-sensitive elements.
        commandLine="$COMMAND $OPTIONS $INPUTFILE -o $OUTPUTFILE"
        # What to show in the progress/details windows.
        progress="Converting"
        # What to show in the right click menu.
        action="&Convert"
        color="#500050" # Color for files of this type.
        # Pass this file onto the link line (see below).
        findLinkLine=true
        # Run during the first pass on multi-pass builds.
        promoteToFirstPass=true
    }

    # The following two lines describe how to pass this file
    # on the link line.

    # Here we're passing this as an extra_file, which adds a
    # reference to the file into the debug info
    # (so that you can "e" to the .bin file).
    Program.linkableTypes += {"Binary Input"}
    Program.linkSpecifiers += {"Binary Input;-extra_file=$(INPUTFILE)"}
}
```

Each file type definition is contained within curly braces, can have an optional identifier (`BinInput` in the preceding example), and contains numerous properties.

The only required property is `name`, which is a string used to identify the type in the Project Manager. There are also a few other properties that are necessary for the build tools to automatically recognize and build the custom file type.

You can add the `extensions` property, which is a list of file extensions that automatically imply custom file types. Add the `command`, `commandLine` and `outputType` properties to specify how to build the custom file type. See "Customization File Syntax" on page 931 for additional details about these and other properties.

If your custom input file is used during a build, you can add that file to the debugging information generated for the final executable. By adding the file to the debugging information, you can access it through the MULTI Debugger by using the **e filename** command.

The preceding example demonstrates this through the use of two lines that append values to the `Program.linkSpecifiers` and `Program.linkableTypes` properties. These lines modify properties of the `Program` file type, which is a standard file type defined in **ghs_filetypes.bod**.

Note

`Program.linkSpecifiers` and `Program.linkableTypes` are not included in the curly braces surrounding the rest of the Binary Input file type definition.

- `Program.linkableTypes` — Lists the names of file types that are included on the linker command line when building the parent Program. When modifying this property, use `+=`, or you will overwrite the default types.
- `Program.linkSpecifiers` — Lists the file type names and the corresponding option to include on the linker command line when building the parent program. When modifying this property, use `+=`, or you will overwrite the default link specifiers.

  The **-extra_file** option indicates that the `INPUTFILE` in the previous example should be added to the debugging information and not to the linked executable. If you add a `FileType` value to the

linkableTypes property but not to the linkSpecifiers property, the output file is added to the linked executable.

Command Definitions

The Commands section of a customization file defines programs or commands required to build or edit custom file types. The definitions specify where the programs are found and which groups of options are supported.

```
Commands {
#Commands contains a list of custom commands and how to locate them
   GHS_Binary_Converter {
      # name of the command referred to in FileType
      name="GHS Binary Converter"
      # GHS_TOOLS_DIR is a macro specifying the location of "gbuild"
      exec="${GHS_TOOLS_DIR}/gbin2c"
      # option sets used by this command
      options={"SpecialOptions","GBCOptions"}
   }
}
```

Each command definition is contained within curly braces, can have an optional identifier (in this example, GHS_Binary_Converter), and contains several required properties. The required properties are:

- exec — The name of the program to run. If it contains spaces, the Builder treats the entire string as the program name (it does not parse the string into arguments). If the program name is specified without a directory, the builder searches for the program in the directories of the PATH environment variable.
- name — A string used to identify the command in FileType definitions.
- options — A list of option groups supported by this command. All commands must list SpecialOptions because this option group is supported by the Builder rather than the executed command.

For more information about other command definition properties, see “Customization File Syntax” on page 931.

CommandOption Definitions

The `CommandOptions` section of a customization file defines groups of command line options supported by the custom commands. Most commands should have their own option group.

```
CommandOptions {
   GBCOptions { # -- Option Set Name --
      # options for the GHS Binary Converter (referred to in the
      # Commands section)

      SizeOpt {
      # Size option for determining whether to output lengths

      {
         name="-size"   #exact name of the option passed
         value=0        #a unique identifier
         enumLabel="On" #text to show when selecting this option
      }

      {
      #Fake options can be added so that a real option can be disabled
      #lower in the hierarchy even though the tool has no such option.
      #When the command is invoked, no option will be passed.

         name="-no_size"      #option name (stored in .gpj files)
         value=1              #unique value for this option
         enumLabel="Off"      #text to show when selecting this option
         flags={"FAKEOPTION"} #don't pass anything on the command line
      }

      delimiter="NoArg"  #this is a stand-alone option (no arguments)
      merge="Replace"    #each occurance in the hierarchy
                         #replaces those above it
      optionType="Enum"  #this is an enumerated value option
      guiLabel="Output the length of each array"
                         #Text to display in the options dialog
      guiCategory="Binary Converter"
                         #Category to add this option into
      }
   }
}
```

Each option group is contained within curly braces, must have an identifier (`GBCOptions` in the example), and includes one or more option definitions. An option definition is also contained within curly braces, can have an identifier (`SizeOpt` in the example), and specifies properties used by the Builder when constructing command lines.

All option definitions must have at least one `name`, a `delimiter`, a `merge` rule, and a `optionType`. The `name` of the option is the exact string to pass to the

command. For options that take a parameter, the `name` is the prefix (for example, **–option=*param*** has the name "-option"). The `delimiter` specifies the connector between the option and the parameter ("Equal" for **-option=*param***). The `merge` rule specifies how multiple instances of the option, on different or the same levels of the hierarchy, are handled. The `optionType` determines which merge rules are available.

An option definition will contain more than one `name` if the option has more than one valid spelling, or if it has a fixed set of choices instead of a free-form parameter. In either of these scenarios, create a sub-definition by adding curly braces around each name and its corresponding value. Options with the same `value` setting are all considered synonyms. The first option listed is treated as the canonical spelling and is used when the option is set from the Project Manager. For more information, see "Customization File Syntax" on page 931.

If you have two different tools that support the same option string, but the behavior of the option is different for each program, you may need to use the **optgroup** conditional control statement in your Project file. For more information, see "Conditional Control Statements" on page 911.

Extended Examples

The following sections provide additional reference examples to use when creating customization files.

Example D.1. Using Scripts to Process Input Files

The subsequent example demonstrates how to use Perl scripts to convert input files into source code. This example defines a single command for the Perl interpreter, which is used to run scripts for three different custom file types.

```
%define scriptdir c:/scripts

Commands {
    {
        name="Perl Script"
        exec="perl"
        options={"SpecialOptions"}
    }
```

```
}


FileTypes {
    {
       name="Prtcl"
       extensions={"prtcl"}
       outputType="SourceFile"
       grepable=true
       extraFiles+={"$(OUTPUTDIR)/$(OUTPUTNAMEBASE)_gen.c"}
       extraFiles+={"$(OUTPUTDIR)/$(OUTPUTNAMEBASE)_gen.h"}
       extraFiles+={"$(OUTPUTDIR)/$(OUTPUTNAMEBASE)_log_gen.c"}
       extraFiles+={"$(OUTPUTDIR)/$(OUTPUTNAMEBASE)_log_gen.h"}
       command="Perl Script"
       commandLine="$COMMAND ${scriptdir}/pb_protocol.pl \
           $INPUTFILE $(OUTPUTDIR)/$(OUTPUTNAMEBASE)_gen.c \
           $(OUTPUTDIR)/$(OUTPUTNAMEBASE)_gen.h"

       progress="Processing"
       action="&Process"
       color="#500050"
       findLinkLine=false
       promoteToFirstPass=true
    }
    {
        name="PBCommands"
        outputType="SourceFile"
        grepable=true
        extraFiles={"docs_gen.txt"}
        command="Perl Script"
        commandLine="$COMMAND ../../src/probe/pb_ti_cmds.pl \
            --html='$(OUTPUTDIR)' $INPUTFILE"

        progress="Processing"
        action="&Process"
        color="#500050"
        findLinkLine=false
        promoteToFirstPass=true
    }

    {
        name="PerlScript"
        outputType="SourceFile"
```

```
        grepable=true
        command="Perl Script"
        commandLine="$COMMAND $INPUTFILE $OPTIONS"
        progress="Running"
        action="&Run"
        color="#500050"
        findLinkLine=false
    }
}
```

Example D.2. Converting Images for Linking

The following example demonstrates how to convert graphical images into a form that can be linked into a project, group the converted images together, and then link them in to the final executable.

There are three file types in this example:

- **Resource** — The images.
- **ResourceBundle** — A collection of images and other bundles.
- **NestedBundle** — A collection of images that must be included in a resource bundle.

These files are processed by two commands. Because different options are available for the various bundle types, there are three command definitions.

```
%if ! defined (SDK_BIN_DIR)
% define SDK_BIN_DIR __DIR__
%endif

CommandOptions {
   ResourceConverterOptions {
     ResourceID {
       name="id"
       merge="Replace"
       delimiter="Equal"
       optionType="String"
       guiLabel="Set Resource ID"
       guiCategory="Resource Converter"
       pass={"All"}
       commonLevel=101
     }
```

```
    ResourceName {
      name="name"
      merge="Replace"
      delimiter="Equal"
      optionType="String"
      guiLabel="Set the resource"
      guiCategory="Resource Converter"
      pass={"All"}
      commonLevel=101
    }


    OutputFormat {
    {
      name="type=default" # option name (stored in .gpj files)
      value=0
      delimiter="NoArg"
      enumLabel="Default type to that in bitmap file"
      flags={"FAKEOPTION"} # don't pass on command line
    }
    {
      name="type=BITMAP_TYPE_16BPP_565"
      value=1
      enumLabel="BITMAP_TYPE_16BPP_565"
    }
    {
      name="type=BITMAP_TYPE_1BPP_VERTICAL"
      value=2
      enumLabel="BITMAP_TYPE_1BPP_VERTICAL"
    }
    {
      name="type=BITMAP_TYPE_1BPP_IDEAL"
      value=3
      enumLabel="BITMAP_TYPE_1BPP_IDEAL"
    }
      delimiter="NoArg"
      merge="Replace"
      optionType="Enum"
      guiLabel="Set the bitmap output type"
      guiCategory="Resource Converter"
      commonLevel=101
    }
  }
```

```
   NestedBundlerOptions {
     NestedLocation {
       name="-i"
       merge="Replace"
       delimiter="space"
       optionType="String"
       guiLabel="Set Resource ID"
       guiCategory="Resource Bundler"
       pass={"All"}
       commonLevel=101
     }
   }
}

Commands {
    {
    name="Resource Converter"
    exec="${SDK_BIN_DIR}\\imgconvert.exe"
    options={"SpecialOptions","ResourceConverterOptions"}
    }
    {
    name="resource_bundler"
    exec="${SDK_BIN_DIR}\\resourcebundler.exe"
    options={"SpecialOptions"}
    }
    {
    name="nested_bundler"
    exec="${SDK_BIN_DIR}\\resourcebundler.exe"
    options={"SpecialOptions","NestedBundlerOptions"}
    }
}


FileTypes {
    Resource_Bundle {
      name="Resource Bundle"
      outputExtension="rsc"
      command="resource_bundler"
      outputType="Program"
      commandLine="$COMMAND -f $OBJECTS -o $OUTPUTFILE $OPTIONS"
      progress="Bundling"
      action="&Build"
      color="#c00000"
```

```
# Nested Bundles can be linked to make Resource Bundles.
      linkableTypes += {"Nested Bundle"}
# Indicates that there's also a header file which is output.
      extraFiles={"$(OUTPUTDIR)\\$(OUTPUTNAMEBASE).h"}
    }
    Nested_Bundle {
      name="Nested Bundle"
      extensions={"gpj"}
      outputExtension="robj"
      command="nested_bundler"
      outputType="Program"
      commandLine="$COMMAND -f $OBJECTS -o $OUTPUTFILE $OPTIONS"
      progress="Bundling"
      action="&Build"
      color="#c00000"
# Nested Bundles can be linked to make Nested Bundles.
      linkableTypes += {"Nested Bundle"}
    }

# These two 'Program.' lines below include the resource bundle
# into an executable program if you have a build structure like
# the following:
#    default.gpj
#       my_program.gpj  [Program]
#          my_resources.gpj [Resource Bundle]
#    my_code.gpj  [Subproject]
# Without these lines, the Resource Bundle will not
# be linked in to the final executable image.
#    Program.linkSpecifiers += {"Resource Bundle; \
#      -extra_file=$(OUTPUTFILE)"}
#    Program.linkableTypes += {"Resource Bundle"}


    {
      name="Resource"
      extensions={"bmp","jpg","jpeg","png","tiff"}
      command="Resource Converter"
      outputExtension="robj"
      outputType="ObjectFile"
      commandLine="$COMMAND $INPUTFILE output=$OUTPUTFILE \
              $OPTIONS"
      progress="Converting"
```

```
        action="&Convert"
        grepable=false
        color="#006000"
    }
}
```

Customization File Syntax

The following sections provide more details about the syntax of the customization files.

FileType Syntax

Commonly used `FileType` properties include:

- `action` — A text string displayed in the Project Manager's context menus that represents the process of building this file type. For example, the action for the compiler is **Compile**.
- `color` — The hexadecimal representation of the color to use when displaying files of this type in the Project Manager.
- `command` — The name of the command used to build files of this type. The name of the command must match the `name` property of a definition in the `Commands` section.
- `commandLine` — The exact command line used to build files of this type. Numerous variables are available for context-sensitive values. For more information see “Context Sensitive Variables” on page 933.
- `extensions` — A list of filename extensions to associate with this file type (see “File Types” on page 915).

  The `extensions` property is optional. If you do not have `extensions`, you must explicitly mention the `FileType` in the **.gpj** file for each file of that type so that the builder can recognize the `FileType`.
- `extraFiles` — A list of additional output files that should be removed when cleaning output files. Numerous variables are available for context-sensitive values. For more information see “Context Sensitive Variables” on page 933.

- `findLinkLine` — Specifies whether this file should be included in the command line when linking its parent executable. If this is `true`, directs the Builder to look for a parent executable that explicitly accepts this type as a `linkableType`. This property overrides the default behavior based on the `outputType`. The default value of this property is `false`.
- `graphicalEditor` — The `name` of the command used to edit files of this type. The name must match the name property of a definition in the `Commands` section. This property is useful if you want to edit the file with a tool other than the MULTI Editor.
- `graphicalEditorCommand` — The exact command line used to edit files of this type. Numerous variables are available for context-sensitive values. For more information see "Context Sensitive Variables" on page 933.
- `grepable` — Specifies whether or not this file can be searched. By default, all files can be searched.
- `linkableTypes` — A list of file type names that are included on the linker command line when linking this file. This property only applies if this file is linked.
- `linkSpecifiers` — A list of file type names and the option to include on the linker command line when linking this file. This property only applies if this file is linked.
- `name` — A string used to identify the file type in the Project Manager. This property is required.
- `outputExtension` — The filename extension on the primary output file generated by files of this type. The Builder uses this information to determine if the file must be built. It also uses this information when cleaning output files (**-clean**).
- `outputType` — The class of output file generated by building files of this type. The following list provides all values for `outputType`, and explains what each value indicates to the Builder:
  - `ObjectFile` — Building this file produces an object file that is linked into its build parent by default.
  - `SourceFile` — Building this file produces a source file that should not be linked into the parent by default.
  - `Program` — This file is a container that takes linkable children as input.

  - `Library` — This file is an archive that may be incrementally linked. It takes linkable children as input. A file type with this class should define a `LibraryLinkInfo` property subgroup with the following entries, formatted in the same way as the `commandLine` property:
    - `update` — The command line used to update a file in the library.
    - `addReplace` — The command line used to add a new file to the library or replace one that is already there.
    - `delete` — The command line used to remove a file from the library.
  - `PassToLinker` — This file is not built, but by default is linked as-is into its build parent.
  - `SelectOne` — Only one of this file's children is built, based on the value of **:select** options that are set.
  - `Subproject` — This file type is a container, but the file itself is not built. Children may still be linked into this file's build parent.
  - `None` — This file is not built and is not linked into its parent.
- `preExec, postExec` — Specifies a command line to run before or after processing all files of this type. If you are running a parallel build, this command triggers a choke point.
- `preExecSafe, postExecSafe` — Specifies a command line to run before or after processing all files of this type. If you are running a parallel build, this command does not trigger a choke point.
- `progress` — The string prefix shown in progress messages when building files of this type.
- `promoteToFirstPass` — Specifies whether processing this file should occur during the first pass of a multi-pass build. Set this property to **true** for file types that generate source code that are built into your project.

Context Sensitive Variables

When defining some of the properties in the `FileTypes` section, you might need to specify information that depends on the specific file being processed. Because the file type definition is generic, you can use the following variables as placeholders for the context-specific information.

The syntax for using these variables is:

- *$VARIABLE*
- *$(VARIABLE)* — This format is required if the variable name contains an underscore.

To specify a dollar sign in a property value, use `$$`. The supported variables are:

`COMMAND` The executable from the `Commands` definition that matches the `command` property of the `FileType`. Resolves to an absolute path if the `runInDirectory` flag is set.
`DISKFILE` Full path to the file being processed.
`EDITOR` Executable of the graphical editor defined by the `FileType` of the file being processed.
`FILENAME` The name of the file being processed. Does not include path information.
`FILENAMEBASE` Base name with no suffix of the file being processed.
`FILENAMELIBBASE` The base name of a library. Same as `FILENAMEBASE` except that it strips away the `lib` prefix if present.
`FILETYPEOPTIONS` List of file types included in the current module. This is used by the driver to determine when a module requires C or C++ libraries.
`INPUTFILE` The name of the input file being processed, including path information relative to the Top Project directory. For commands with multiple passes, this variable also includes a list of output files from the first pass when used with the second pass.
`INPUTDIR` Relative path to the directory containing the input file.
`ITEMID` Unique ID number for the file being processed. This number is not guaranteed to be the same across builds.
`KEEPOBJECTS` List of all objects in an archive that should remain in the archive after processing.

MAKE_DEPEND_OPTIONS Arguments specified in the `makeDepends` section of the `COMMAND` definition used for generating dependency information for the file being processed.
OBJECTS List of all the objects to be linked together for this file.
OBJECTBASE Base name with no suffix of the last object file in the list. This is a special case for single file libraries.
OBJECTSUFFIX Suffix for the object files.
OPTIONS A space-separated string containing all of the options set on this file, including inherited options.
OUTPUTDIR Relative path to the output directory.
OUTPUTFILE Relative path to the primary file output by processing the current file (that is, the object file for a source file). If the file requires two-pass processing, the extension on the `OUTPUTFILE` is different for the two passes.
OUTPUTNAMEBASE Base name with no suffix of the output name for the file being processed.
PARENTID Unique ID number for the parent of the file being processed. If there is no parent, the value is zero. This number is not guaranteed to be the same across builds.
RUNDIR The directory the command is run in. This is the same as the directory containing the Top Project except for commands with the `runInDirectory` flag set.
SECOND_PASS_OPTION Same as the `secondPassOption` from the `FileType` definition. Used for tools that operate using multiple passes to indicate the current pass.
TOPPROJECT Full path of the Top Project.
SIBLINGS Full paths of all files of the `siblingType`, which are descendents of the parent of the file being processed.

SIBLINGS*[type]*
Like SIBLINGS, but uses the specified type instead of the siblingType parameter.

Command Syntax

Commonly used Command syntax includes:

- exec — The name of the program to run. If it contains spaces, the Builder treats the entire string as the program name (it does not parse the string into arguments). If the program name is specified without a directory, the builder searches for the program in the directories of the PATH environment variable. Required.
- name — A string used to identify the command in FileType definitions. Required.
- options — A list of option groups supported by this command. All commands should support SpecialOptions. Required.
- specialOptions —Flags supported by the program that can be used for integration with advanced features in the Builder. Two commonly used special properties are:
  - runInDirectory — Specifies whether the command should be run with the current working directory set to the directory containing the input file. By default, the command is run with the working directory set to the directory containing the Top Project.
  - showCommands — Specifies the flag used to generate verbose output.

CommandOption Syntax

The following values are available for CommandOption property definitions:

- commonLevel — Specifies how visible the option is in the Project Manager:
  - 1 - 100 — A basic option.
  - 101 - 1000 — An intermediate option.
  - >1000 — [default] An advanced option.
- delimiter — Specifies the connector between the option and any parameter it may take:

  - `Equal` — Is in the form ***-option=value***.
  - `NoArg` — [default] Is in the form ***-option***.
  - `Space` — Is in the form ***-option value***.
  - `Touching` — Is in the form ***-optionvalue***.
- `merge` — Specifies how multiple settings of the option are defined in the same or different files:
  - `Concat` — Appends a value to the end of a list.
  - `None` — [default] The value is not inherited.
  - `Preconcat` — Prepends a value to the beginning of a list.
  - `Replace` — Replaces the previous value with the new one.
- `name` — The exact string or prefix to pass on the command line. This string must not contain spaces. Required.
- `optionType` — Specifies the type of option:
  - `Enum` — An on/off switch, and permits a `merge` type of `Replace`.
  - `EnumList` — A group of on/off switches, and permits a `merge` type of `Concat`.
  - `List` — [default] Accepts a comma-separated list of values and permits a `merge` type of `Concat` or `Preconcat`.
  - `String` — Accepts a single value and permits a `merge` type of `Replace`.
- `pass` — Specifies which parts of the toolchain the option is passed to:
  - `All` — [default] All tools (most common).
  - `Compile` — Compiler only.
  - `Depends` — Preprocessor dependencies resolution only.
  - `Link` — Linker only.
  - `None` — None of the tools.
- `value` — A numerical identifier for this option setting. If you have two sub-definitions with different spellings but the same meaning, specify the same numerical identifier for their `value` fields. If they have different meanings, use different numerical identifiers for each `value` field.

Preprocessor Directives

Like C and C++ source files, the customization files are processed by a preprocessor before they are loaded by the Builder. The preprocessor supports numerous standard directives you can use to conditionally include or disable portions of your customization file:

`%define` Defines a macro.

`%echo`
`%info`
Outputs a message.

`%elif`
`%elsif`
`%elseif`
`%else`
`%endif`
`%if`
Conditional inclusion directives. `%elif` and `%elsif` are equivalent to `%elseif`.

`%error` Outputs an error message.
`%ifdef` Shorthand for `%if defined`.
`%include` Includes another file.
`%warning` Outputs a warning message.

Preprocessor Functions

In addition to the preprocessor directives listed in the preceding section, you can use several preprocessor functions in your customization file. These functions are typically used in combination with the `%if` directive.

- `defined(`*`MACRO_NAME`*`)` — Evaluates to true if the macro is defined. For example:

  ```
  %if defined(FOO)
  ```

- `file_exists(`*`FULL_PATH`*`)` — Evaluates to true if the file or directory specified exists. For example:

  ```
  %if file_exists("${__DIR__}/myfile")
  ```

- `streq(`*`STRING1`*`,`*`STRING2`*`)` — Evaluates to true if the strings are equal. For example:

  ```
  %if streq("${__MULTI_HOST__}", "linux86")
  ```

Predefined Macros

You can use the following macros in any customization file:

- `__DIR__` — Directory of the file you are editing.
- `__INTEGRITY_DIST__` — Directory of the INTEGRITY installation (only defined for INTEGRITY targets).
- `__MULTI_HOST__` — The host machine's operating system.
- `__TOOLS_DIR__` — The MULTI Compiler installation directory.

The syntax for using one of these macros is `${MACRO}`. See Example D.3. Including Additional .bod files on page 939 for an example.

Example D.3. Including Additional .bod files

The following example demonstrates how to use preprocessor directives, functions, and macros to include additional, optional **.bod** files.

```
# Include kernel build types configuration file
%if file_exists("${__DIR__}/../kernel/kernel.bod")
%include "${__DIR__}/../kernel/kernel.bod"
%endif

# Include build type configuration file for examples
```

```
%include "${__DIR__}/../examples/copyfile.bod"

# Include build type configuration file for validations
%if file_exists("${__DIR__}/../rtos_val/validations.bod")
%include "${__DIR__}/../rtos_val/validations.bod"
%endif
```

Appendix E

The MULTI Eclipse Plug-in

Contents

The MULTI for Eclipse plug-in integrates the MULTI Debugger and Green Hills compilers with the Eclipse environment. You can use this plug-in to create, build, and run a Green Hills project; you can then debug the project using the MULTI Debugger. MULTI for Eclipse is ideal for using the same environment to develop both Java and C/C++ applications.

MULTI for Eclipse uses GNU Make to build projects. By default, Eclipse builds your project by automatically generating a makefile. However, you can easily disable this feature if you prefer the flexibility of writing your own makefile. When you build the project using MULTI for Eclipse, the Eclipse C/C++ Development Tools (CDT) use Green Hills compilers.

When you debug a project using MULTI for Eclipse, the Eclipse platform launches the MULTI Debugger, which interfaces with Green Hills simulators and embedded targets. With simple configuration management, the Debugger can support unlimited concurrent target connections, allowing you to debug complex heterogeneous systems. The MULTI Debugger also enables superior source code browsing and contains built-in support for advanced instruction and data trace. If you use TimeMachine, this trace data gives you the ability to step through, run through, or hit breakpoints you encountered earlier in program execution. For more information about TimeMachine, see the documentation about analyzing trace data with the TimeMachine tool suite in the *MULTI: Debugging* book.

While MULTI for Eclipse provides the convenience of automatically generated makefiles or the flexibility to set up your own makefiles, as well as the ability to develop Java and C/C++ applications in the same environment, it does not provide features that are available from the MULTI Project Manager. The following list outlines aspects of the MULTI Project Manager that you might find helpful. The Project Manager:

- Provides a **Project Wizard** that allows you to populate your project with example source code.
- Can build multiple executables per build using nested projects. This can be very helpful if you are building a Monolith INTEGRITY project or a Dynamic Download INTEGRITY project with multiple AddressSpaces.
- Provides a powerful command line interface, enabling advanced automated builds.
- Contains online help for every build option.

For more information about the MULTI Project Manager, see the documentation about the MULTI Project Manager in the *MULTI: Managing Projects and Configuring the IDE* book.

The following sections explain how to install MULTI for Eclipse, how to perform basic tasks within the Eclipse environment, and how to access MULTI components from within Eclipse. For information about Eclipse itself, see the documentation available on the Eclipse Web site [http://www.eclipse.org].

Prerequisites to Installing MULTI for Eclipse

Before you install MULTI for Eclipse, ensure that you have installed the following software:

- MULTI IDE and Green Hills Compiler installations — The MULTI IDE and Compiler installation directories must also be in your PATH environment variable. To ensure MULTI is in your PATH, open a new shell and type `multi`. If the MULTI Launcher starts up, the IDE is in your PATH. To ensure that the compiler is in your PATH, type `gdump` in the shell. If **gdump** runs, the compiler is also in your PATH.
- Eclipse and CDT installations. The supported versions are:
  - Eclipse 3.6.x with CDT 7.0.x
  - Eclipse 3.7.x with CDT 8.0.x
  - Eclipse 4.2.x with CDT 8.1.x

  You can download Eclipse and CDT from the Eclipse Web site [http://www.eclipse.org/cdt].
- GNU Make — The regular Make (MKS Make, for example) is insufficient.

  To install GNU Make on Windows, it is easiest to install either Cygwin, which you can download from the Cygwin Web site [http://www.cygwin.com], or MinGW and MSYS, which you can download from the MinGW Web site [http://www.mingw.org]. Your version of GNU Make must support DOS/Windows-style paths (such as **c:\path\to\file.txt**). GNU Make 3.80 or earlier should work.

  The bin directory for whichever you install must also be in your PATH environment variable. To ensure that the directory containing GNU Make is

in your PATH, open a new shell and type `make --version`. If the shell prints a GNU Make version message with a Free Software Foundation copyright notice, the appropriate directory is in your PATH.

- Java SE 5 or 6 — You can download Java from Oracle's Java web site [http://www.java.com]. Java SE 5 works with Eclipse 3.6. We recommend Java SE 6 for Eclipse 3.7.

Note

The instructions in the following sections were written for Eclipse 3.6 and CDT 7.0. They should also apply to newer versions.

Installing MULTI for Eclipse

To install MULTI for Eclipse, follow these steps:

1. Open Eclipse, and select **Help → Install New Software**.
2. In the **Install** dialog that appears, click **Add**.
3. In the **Add Repository** dialog that appears, click **Local**.
4. Navigate to the **eclipse** directory located in your MULTI compiler installation.
5. In the **Name** box, enter `MULTI for Eclipse`. Click **OK**.
6. In the **Name/Version** list, expand the **Green Hills MULTI for Eclipse** item. Select **Green Hills MULTI for Eclipse (*target_architecture*) *version_number***, where *target_architecture* is your target architecture and *version_number* is the latest version number in the list. If you have MULTI licenses for more than one architecture, you can select all the targets you are licensed for. Click **Next**.
7. If you accept the terms of the feature license, select **I accept the terms in the license agreement**.
8. Click **Finish**.
9. In the **Security Warning** window that appears, click **OK**.
10. In the **Software Updates** dialog box that appears, click **Restart Now** to restart Eclipse.

Creating a Green Hills Project from Within Eclipse

You can create a Green Hills project from within Eclipse. To create a project:

1. In Eclipse, select **File → New → Project**.
2. In the **Wizards** pane of the **New Project** wizard, select a C or C++ project. Click **Next**.
3. In the **Project name** text field, enter a name for your project. To save your project in a directory other than the default, clear the **Use default location** check box and specify a directory.
4. In the **Project types** pane, select the appropriate Green Hills project type. Click **Next**.
5. In the **Configurations** pane, select **Debug**, **Release**, or both.
   - **Debug** — Disables optimization.
   - **Release** — Enables optimization.

   If you select both configurations, **Debug** is set by default. You can change the configuration later. For information, see "Changing Compiler Options" on page 949.
6. If you want to use an automatically generated makefile (the default), click **Finish**.

If you want to use a custom makefile:

a. Click the **Advanced settings** button.

b. Select **C/C++ Build** on the left.

c. Clear the check box **Generate Makefiles automatically**.

d. Set other options as appropriate to control how Eclipse interacts with your makefile. For information about settings, see the documentation for CDT available from the Eclipse Web site [http://www.eclipse.org].

e. Click **OK** and then **Finish**.

Your project should appear in the **Project Explorer** view.

Creating an INTEGRITY Project

To create an INTEGRITY project, follow the preceding instructions that outline how to create a Green Hills project.

In addition to the preceding steps, you must specify a BSP and operating system installation directory and create an Integrate configuration file. To specify the BSP and OS directory:

1. In the **Project Explorer** view, right-click the top-level project and select **Properties**.
2. On the left side of the **Properties** window that appears, expand **C/C++ Build** and select **Settings**.
3. In the hierarchy present on the **Tool Settings** tab, select **Target**. On the right side of the window, in the **Board Support Package** drop-down box, select the appropriate board.

   You must set the BSP a total of three times, once for each top-level heading (**Compiler**, **Linker**, and **Integrate**) that appears in the hierarchy on the left.
4. In the hierarchy present on the **Tool Settings** tab, expand **Target**, and then select **Operating System**. On the right side of the window, in the **OS Directory** text field, specify your operating system installation directory.

   You must specify the OS directory a total of three times, once for each top-level heading (**Compiler**, **Linker**, and **Integrate**) that appears in the hierarchy on the left.

5. If your INTEGRITY installation includes the **intex** executable in the directory **rtos/multi/bin/*hosttype***, you must add that directory to your PATH. In the hierarchy present on the **Tool Settings** tab, under **GHS Integrate**, select **Integrate Options**. On the right side of the window, in the **Tools Directory** box, specify your MULTI Compiler installation directory.

Note

On Windows, if you use a **Browse** button in the **C/C++ Build** section of the **Properties** window, the resulting path contains backslashes. Because these backslashes cause Eclipse to generate an invalid makefile, you must change them to forward slashes.

If you created an INTEGRITY project, you must also create an Integrate configuration file. To create an Integrate configuration file:

1. in the **Project Explorer** view, right-click your INTEGRITY project and select **New → File**.
2. In the **File name** text field, enter a name for your configuration file. Append **.int** to the end of the filename. Click **Finish**.

Your Integrate configuration file should appear under your INTEGRITY project in the **Project Explorer** view. If you need to change Integrate options, right-click the project name (not the Integrate file) and select **Properties**.

To add content to your new source file, double-click the filename in the **Project Explorer** view. Enter content in the main view.

If you created a KernelSpace project, the `Filename` keyword in the Kernel section is always `default`, regardless of what you named your INTEGRITY project. The following example displays the most minimal Integrate configuration file possible for a KernelSpace project.

```
Kernel
        Filename                            default
EndKernel
```

For an INTEGRITY KernelSpace project, you must also include the **global_table.c** source file in your project. To do so, either:

1. Copy the standard example **global_table.c** located in the **kernel** directory of your INTEGRITY installation.
2. Paste this copy into your project.

Or:

1. Right-click your INTEGRITY project and select **New → File**.
2. Click the **Advanced** button.
3. Select the check box **Link to file in the file system**.
4. Click the **Browse** button to navigate to the **global_table.c** file located in the **kernel** directory of your INTEGRITY installation. Select this file and click **Open**.
5. Click **Finish**.

If you created a Dynamic Download project, the `Filename` keyword in the Kernel section is always `DynamicDownload`, regardless of what you named your INTEGRITY project.

An Integrate configuration file may contain any number of AddressSpace sections. The AddressSpace section requires the `Filename` keyword. Follow `Filename` with the name of the file that contains the AddressSpace executable. The name of this file is always `default.as`, regardless of what you named your INTEGRITY project. You can see how two `Filename` keywords are specified in the following example, which displays the most minimal Integrate configuration file possible for a Dynamic Download project.

```
Kernel
        Filename        DynamicDownload
EndKernel
AddressSpace
        Filename        default.as
        Language        C
        Library         libINTEGRITY.so
        Library         libc.so
EndAddressSpace
```

Note

Eclipse's automatically generated makefiles allow only one AddressSpace per Dynamic Download project. (The MULTI IDE's Project Manager

allows more advanced configuration options for INTEGRITY users.) If you need more than one AddressSpace in a Dynamic Download project built from within Eclipse, you must use a custom makefile. For information about using custom makefiles, see "Creating a Green Hills Project from Within Eclipse" on page 945.

For more information about how to format an Integrate configuration file, see the *Integrate User's Guide*.

Adding a Source File to Your Project

MULTI for Eclipse supports C, C++, and assembly source files. You can add C, C++, and assembly source files to a C++ project. You can add C and assembly source files to a C project. To add a source file to your project:

1. Ensure that your project is selected in the **Project Explorer** view and select **File → New → Source File**.
2. In the **Source File** text field, enter a name for your source file. Be sure to include the **.c** file extension for C source files, the **.cpp** file extension for C++ source files, and the **.s** file extension for assembly source files. Click **Finish**.

Your source file should appear under your project in the **Project Explorer** view.

To add content to your new source file, double-click the filename in the **Project Explorer** view. Enter content in the main view.

Changing Compiler Options

If you are using an automatically generated makefile, you can change the compiler and linker options that appear in the **C/C++ Build → Settings** section of the **Properties** window. You can either change these options on an option-by-option basis or by configuration. To change your options on an option-by-option basis:

1. To change options for the entire project — In the **Project Explorer** view, right-click your project name and select **Properties**.

   To change options for a single file — In the **Project Explorer** view, right-click your source file's filename and select **Properties**.

2. On the left side of the **Properties** window that appears, expand **C/C++ Build** and select **Settings**.
3. Click through all the levels of the hierarchy on the **Tool Settings** tab. Options you can set appear to the right. For information about these options, see Chapter 3, “Builder and Driver Options” on page 117.

Note

On Windows, if you use a **Browse** button in the **C/C++ Build** section of the **Properties** window, the resulting path contains backslashes. Because backslashes here cause Eclipse to generate an invalid makefile, you must change the backslashes to forward slashes.

Tip

To revert a file's options to the project's settings, right-click the file and select **Resource Configurations → Reset to Default**. Select the files you want to revert and click **OK**.

You can also change your project's configuration, which is its combination of option settings. The preset configurations are **Debug** and **Release** (see “Creating a Green Hills Project from Within Eclipse” on page 945), but you can create and save your own configuration as well. To do so, click the **Manage configurations** button in the **C/C++ Build** section of the **Properties** window. For more information, see the documentation available on the Eclipse Web site [http://www.eclipse.org].

Building and Running Your Project

You must build your project before you can run it. To build your project, select **Project → Build Project**. You can see the output of your build in the **Console** view. The output includes relevant compile and make errors.

To run your project, follow these steps:

1. Ensure that your project is selected in the **Project Explorer** view, and select **Run → Run Configurations**.

Note

Use of the Eclipse **Run Configurations** dialog is not supported for Dynamic Download INTEGRITY applications. Dynamic Download INTEGRITY applications should be downloaded from within MULTI. For instructions, see the documentation about loading a run-mode program in the *MULTI: Debugging* book.

2. The **Run Configurations** dialog appears. In the pane located on the left side of the dialog, double-click **GHS Local C/C++ Launch** for a new configuration of your project. Your project's name is selected automatically. Run configurations that you create in this dialog are saved for future use.

   Under the **Main** tab, the **Project** text field should now contain the name of your project.

3. Click **Search Project**. In the **Program Selection** dialog box that appears, ensure that the binary that is selected is the one you want to run. (If you have never successfully built your project, no binary is present.) Click **OK**.

4. In the **Connect to a target** text field, specify:

   - A setup script filename if applicable (Note that if you enter the path to a setup script, you must use forward slashes, not backslashes. For example, on Windows, you might enter the path `C:/GHS/`*`install_dir`*`/target/`*`architecture`*`/`*`bsp`*`/`*`script`*`.mbs`.)
   - The name of your debug server
   - The arguments required to start your debug server

   Enter this information as you would if you had entered the **connect** command in the MULTI command pane. You do not have to type **connect** in this field; only enter the relevant arguments. See the description of the **connect** command in the documentation about target connection commands in the *MULTI: Debugging Command Reference* book.

   For an example showing how to connect with a setup script, see Example E.1. Connecting with a Setup Script on page 952.

5. Click **Apply** and then **Run**.

You can see the output of your project in the **Console** view.

Example E.1. Connecting with a Setup Script

Suppose you want to run your program on a target that requires a setup script. You are connected to your target with a Green Hills Probe, and your target's IP address is 192.168.100.55. To create, specify, and run a board setup script, you might do the following:

1. In the **Project Explorer** view, right-click your project and select **New → File**.
2. In the **File name** text field, enter `script.mbs` and click **Finish**. Write a setup script within this file or copy one of the **.mbs** setup scripts provided in the **target** directory of your MULTI installation.
3. See "Building and Running Your Project" on page 950 and follow steps 1-4 for running your project.

   For step 4, enter arguments to the **connect** command. For example:

   ```
   setup=script.mbs mpserv 192.168.100.55
   ```

   where:

   - `setup=script.mbs` specifies the name of the file that contains your setup script.
   - `mpserv` is the name of the debug server in use.
   - `192.168.100.55` is the IP address of your target. This is a required *`argument`* for the **mpserv** debug server.
4. Click **Apply** and then **Run**.

Debugging Your Project with the MULTI Debugger

You must build your project before you can debug it. For information, see "Building and Running Your Project" on page 950. To debug your project with the MULTI Debugger:

1. Ensure that your project is selected in the **Project Explorer** view and select **Run → Debug Configurations**.
2. The **Debug Configurations** dialog appears. In the pane located on the left side of the dialog, double-click **GHS Local C/C++ Launch** for a new

configuration of your project, or select a saved configuration. Run configurations that you create in this dialog are saved for future use.

Under the **Main** tab, the **Project** text field should now contain the name of your project.

3. Click **Search Project**. In the **Program Selection** dialog box that appears, ensure that the binary that is selected is the one you want to run. (If you have never successfully built your project, no binary is present.) Click **OK**.
4. In the **Connect to a target** text field, specify:
   - A setup script filename if applicable (Note that if you enter the path to a setup script, you must use forward slashes, not backslashes. For example, on Windows, you might enter the path `C:/GHS/`*`install_dir`*`/target/`*`architecture`*`/`*`bsp`*`/`*`script`*`.mbs`.)
   - The name of your debug server
   - The arguments required to start your debug server

   Enter this information as you would if you had entered the **connect** command in the MULTI command pane. You do not have to type **connect** in this field; only enter the relevant arguments. See the description of the **connect** command in the documentation about target connection commands in the *MULTI: Debugging Command Reference* book.

   For an example showing how to connect with a setup script, see Example E.1. Connecting with a Setup Script on page 952.
5. Click **Apply** and then **Debug**.

The MULTI Debugger opens and automatically connects to your target. Debug your project as usual.

Appendix F

Third-Party License and Copyright Information

ldpreload.c Copyright C 2001 Steven Engelhardt All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Index

Symbols

A

B

C

D

--misra_2004=12.12, 179
--misra_2004=12.13, 180
--misra_2004=12.2, 178
--misra_2004=12.3, 178
--misra_2004=12.4, 178
--misra_2004=12.5, 179
--misra_2004=12.6, 179
--misra_2004=12.7, 179
--misra_2004=12.8, 179
--misra_2004=12.9, 179
--misra_2004=13.1, 180
--misra_2004=13.2, 180
--misra_2004=13.3, 180
--misra_2004=13.4, 180
--misra_2004=13.5, 180
--misra_2004=13.6, 180
--misra_2004=13.7, 180
--misra_2004=14.1, 180
--misra_2004=14.10, 181
--misra_2004=14.2, 181
--misra_2004=14.3, 181
--misra_2004=14.4, 181
--misra_2004=14.5, 181
--misra_2004=14.6, 181
--misra_2004=14.7, 181
--misra_2004=14.8, 181
--misra_2004=14.9, 181
--misra_2004=15.1, 181
--misra_2004=15.2, 182
--misra_2004=15.3, 182
--misra_2004=15.4, 182
--misra_2004=15.5, 182
--misra_2004=16.1, 182
--misra_2004=16.10, 183
--misra_2004=16.2, 182
--misra_2004=16.3, 182
--misra_2004=16.4, 182
--misra_2004=16.5, 183
--misra_2004=16.6, 183
--misra_2004=16.7, 183
--misra_2004=16.8, 183
--misra_2004=16.9, 183
--misra_2004=17.1, 183
--misra_2004=17.2, 183
--misra_2004=17.3, 183
--misra_2004=17.4, 183
--misra_2004=17.5, 183
--misra_2004=17.6, 184
--misra_2004=18.1, 184
--misra_2004=18.2, 184
--misra_2004=18.3, 184
--misra_2004=18.4, 184
--misra_2004=19.1, 184
--misra_2004=19.10, 185
--misra_2004=19.11, 185
--misra_2004=19.12, 185
--misra_2004=19.13, 185
--misra_2004=19.14, 185
--misra_2004=19.15, 186
--misra_2004=19.16, 186
--misra_2004=19.17, 186
--misra_2004=19.2, 184
--misra_2004=19.3, 184
--misra_2004=19.4, 184
--misra_2004=19.5, 184
--misra_2004=19.6, 185
--misra_2004=19.7, 185
--misra_2004=19.8, 185
--misra_2004=19.9, 185
--misra_2004=2.1, 172
--misra_2004=2.2, 172
--misra_2004=2.3, 172
--misra_2004=2.4, 173
--misra_2004=20.1, 186
--misra_2004=20.10, 187
--misra_2004=20.11, 187
--misra_2004=20.12, 187
--misra_2004=20.2, 186
--misra_2004=20.3, 186
--misra_2004=20.4, 186
--misra_2004=20.5, 186
--misra_2004=20.6, 186
--misra_2004=20.7, 187
--misra_2004=20.8, 187
--misra_2004=20.9, 187
--misra_2004=21.1, 187
--misra_2004=3.1, 173
--misra_2004=3.2, 173
--misra_2004=3.3, 173
--misra_2004=3.4, 173
--misra_2004=3.5, 173
--misra_2004=3.6, 173
--misra_2004=4.1, 173
--misra_2004=4.2, 173
--misra_2004=5.1, 174
--misra_2004=5.2, 174
--misra_2004=5.3, 174
--misra_2004=5.4, 174
--misra_2004=5.5, 174
--misra_2004=5.6, 174
--misra_2004=5.7, 174
--misra_2004=6.1, 174

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z