

MULTI: Building Applications for Embedded ARM64 v2021.1



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

LEGAL NOTICES AND DISCLAIMERS

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 © 1983-2021 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, and DoubleCheck 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 listing of Green Hills Software's periodically updated patent marking information, please visit http://www.ghs.com/copyright_patent.html.

PubID: build_arm64-696774

Branch: <http://toolsvc/branches/release-branch-2021-1-comp>

Date: April 22, 2021

Contents

Preface	xxv
About This Book	xxvi
The MULTI Document Set	xxvi
Conventions Used in the MULTI Document Set	xxvii
The Green Hills Toolchain	xxx
The C and C++ Compiler Driver	xxx
The Optimizing Compilers	xxx
The asarm64 Assembler	xxxi
The ax Librarian	xxxi
The elxr Linker	xxxi
Optimized Libraries and Header Files	xxxii
Utility Programs	xxxii
The Builder	xxxii
 Part I. Using the MULTI Compiler	 1
 1. The Compiler Driver	 3
Compiler Driver Syntax	5
Building an Executable from C or C++ Source Files	6
Working with Input Files	7
Recognized Input File Types	8
Passing Multiple Input File Types to the Driver	9
Specifying the Language When Linking a Program	9
Passing Linker Directives Files to the Driver	10
Generating Other Output File Types	10
Creating Libraries	11
Adding and Updating Files in Libraries	12
Driver Options for Intermediate Forms of Output	12
Output File Types	14
Controlling Driver Information	16

Using a Driver Options File	17
Using Makefiles	18
Driver Options for Individual Compiler Passes	22
Green Hills Equivalents to GNU Tools	23
Generating Dependency and Header File Information	23

2. Developing for ARM64 25

ARM64 Characteristics	26
C and C++ Data Type Sizes and Alignment Requirements	27
Register Usage	28
Calling Conventions	29
Structure Packing	30
Understanding Structure Packing	31
Using #pragma pack to Pack All Instances of a Structure Type	32
Using the Packing Builder Option to Pack All Structures	33
Pointing to Packed Structures with the __packed Type Qualifier	34
Specifying an ARM64 Target	35
ARM64 Processor Variants	37
Enabling Debugging Features	38
Generating Debugging Information	38
Generating Debugging Information for Applications Compiled with Third-Party Compilers	40
Enabling Command Line Function Calls	45
Function Entry/Exit History	45
Obtaining Profiling Information	46
Enabling Run-Time Error Checking	51
Enabling Run-Time Memory Checking	51
Using Your Own Header Files and Libraries	56
Instructing the Compiler to Search for Your Headers	57
Instructing the Compiler to Link Against Your Libraries	59
Controlling the Assembler	59
Controlling the Linker	59
Working with Linker Directives Files	60

Text and Data Placement	61
Default Program Sections	61
Custom Program Sections	62
Aligned Data Sections	64
Individual Function and Variable Sections	66
Assigning Program Sections to ROM and RAM	69
Storing Global Variables in Registers	70
Near and Far Function Calls	72
Customizing the Green Hills Run-Time Environment	73
Other Topics	74
Renaming the Output Executable	74
Specifying an Alternate Program Start Address	74
Symbolic Memory-Mapped I/O	74
Verifying Program Integrity	76

3. Builder and Driver Options 77

Target Options	79
Endianness	79
Memory Functions	79
Enable Toolchain Versions of Certain Memory Functions	81
Annotate Call/Return Instructions	81
Memory Model	81
Workaround ARM64 Erratum 1183550 (DMB)	82
Workaround ARM64 Erratum 1183550 (NOP)	82
Straight-line Spectre Compiler Mitigation	82
Straight-line Spectre Linker Mitigation Check	83
Straight-line Spectre Assembler Mitigation	83
Floating-Point	83
Text and Data Placement	90
Operating System	106
Project Options	113
Object File Output Directory	113
Emulate Behaviors of a Specific Compiler Version	113
Source Root	114
Include Directories	114
Library Directories	115
RTOS Library Directories	115

Libraries	116
Output Filename	116
Source Directories Relative to This File	117
Intermediate Output Directory Relative to Top-Level Project	117
Optimization Options	119
Optimization Strategy	119
Intermodule Inlining	121
Automatic Vector Optimization	121
Linker Optimizations	122
Interprocedural Optimizations	122
Optimization Scope	123
Individual Functions	130
Debugging Options	132
Debugging Level	132
INTEGRITY TimeMachine	132
Function Entry/Exit Logging	133
EAGLE Logging	133
Profiling - Block Coverage	134
Function Entry/Exit History Instruction Threshold	134
Profiling - Target-Based Timing	135
Run-Time Error Checks	135
Run-Time Memory Checks	138
Number of Callers to Track with Run-Time Memory Checks	139
Preprocessor Options	140
Define Preprocessor Symbol	140
Undefine Preprocessor Symbol	140
Display includes Preprocessor Directives Listing	140
C/C++ Compiler Options	142
C Language Dialect	142
C Japanese Automotive Extensions	143
C++ Language Dialect	143
C++ Libraries	145
C++ Exception Handling	145
Allow C++ Style Slash Comments in C	146
ANSI Aliasing Rules	146
Thread-Local Storage	147

MISRA C	147
MISRA C 1998	149
MISRA C 2004	159
MISRA C 2012	175
Data Types	204
Alignment and Packing	207
C/C++ Data Allocation	208
Special Tokens	209
C++	212
Assembler Options	224
Source Listing Generation	224
Source Listing Generation Output Directory	224
Preprocess Assembly Files	224
Preprocess Special Assembly Files	225
Interleaved Source and Assembly	225
Additional Assembler Options	225
Assembler Command File	226
Support for C Type Information in Assembly Files	226
Linker Options	227
Output File Type	227
Generate Additional Output	228
Executable Stripping	228
Start Address Symbol	229
Append Comment Section with Link-Time Information	229
Preprocess Linker Directives Files	229
Linker Warnings	230
Raw Import Files	230
Linker Directive Files with Non-standard Extensions	231
Additional Linker Options (beginning of link line)	231
Additional Linker Options (before start file)	232
Additional Linker Options (among object files)	233
Linker Command File	233
Extract Whole Archive	234
Report an Error When Multiple Libraries on a Link Line Contain a Definition of the Same Symbol	234
When Rejecting Duplicate Symbols, Allow Certain Symbols Added by the compiler	234

When Rejecting Duplicate Symbols, Allow Symbols in .ghs.linkonce sections	235
Duplicate Whitelist	235
Linker Optimizations	235
Start and End Files	238
Symbols	239
Linker Output Analysis	242
Link-Time Checking	245
Compiler Diagnostics Options	251
Warnings	251
Remarks	251
Maximum Number of Errors to Display	251
Redirect Error Output to File	252
Quit Building if Warnings are Generated	252
Display Version Information	252
Write Version Info to .comment Section	252
Green Hills Standard Mode	253
Coding Standard Profile	253
Varying Message Format	253
C/C++ Messages	255
Toolchain Messages	261
DoubleCheck (C/C++) Options	262
DoubleCheck Level	262
DoubleCheck Report File	262
DoubleCheck Config File	263
DoubleCheck Errors to Ignore	263
DoubleCheck Errors to Enable	263
DoubleCheck Output that Stops Builds	264
Advanced Options	265
Target Options	265
Project Options	269
Optimization Options	286
Debugging Options	294
Preprocessor Options	304
C/C++ Compiler Options	308
Assembler Options	313
Linker Options	314
Archiver Options	318

Compiler Diagnostics Options	320
Support Diagnostics Options	321
HTML Compiler Options	322

4. Optimizing Your Programs 325

Inlining Optimizations	326
When You Should Consider Inlining Optimizations	327
Advantages of Inlining	327
Using Inlining Optimizations	328
Manually Indicating Functions for Single-Pass Inlining	329
Indicating Functions for Two-Pass Inlining	331
Combining the Inlining Option Variants	332
Inlining Options You Should Generally Leave Alone	333
Additional C++ Inlining Information and Caveats	334
Interprocedural Optimizations	335
When You Should Consider Interprocedural Optimizations	336
Disabling Specific Interprocedural Optimizations	338
Using Standard Interprocedural Optimizations	338
Using Wholeprogram Interprocedural Optimizations	338
Selectively Excluding Compilation Units	339
Libraries and Interprocedural Optimizations	340
Loop Optimizations (Including SIMD Vectorization)	340
When You Should Consider Loop Optimizations	340
Automatic Loop Optimization	341
Manual Loop Optimization	341
Loop Optimization Processes	342
SIMD Vectorization	346
General Optimizations	350
When You Should Consider General Optimizations	350
Peephole Optimization	351
Pipeline Instruction Scheduling	351
Common Subexpression Elimination	352
Tail Calls	353
Constant Propagation	353
C/C++ Minimum/Maximum Optimization	354
Memory Optimization	354

Dead Code Elimination	355
Static Address Elimination	355
5. The DoubleCheck Source Analysis Tool	357
Introduction	358
Enabling DoubleCheck	359
Specifying a DoubleCheck Report File	360
Using Custom Functions with DoubleCheck	361
Specifying Function Properties in a DoubleCheck Configuration	
File	361
Specifying Function Properties with Pragma Directives	362
Property Types	363
Viewing DoubleCheck Reports	366
Launching the Report Viewer from the MULTI Project	
Manager	366
Launching the Report Viewer From the Command Prompt ...	366
Controlling DoubleCheck Output	367
Promoting Source Analysis Errors and Warnings	367
Ignoring and Enabling Source Analysis Errors and	
Warnings	368
Source Analysis Error and Warning Messages	369
Error Messages	370
Warnings	383
Part II. Using Advanced Tools	387
6. The asarm64 Assembler	389
Running the Assembler from the Builder or Driver	390
Generating Assembly Language Files	391
Running the Assembler Directly	391
Assembler Options	391
General Assembler Options	391
Assembler Syntax	392
Identifiers	392

Reserved Symbols	393
Source Statements	395
Expressions	396
Assignment Statements	396
Integral Expression Operators	397
C Type Information Operators	398
Expression Types	400
Type Combinations	401
Labels	401
Current Location	401
Reading the Compiler's Assembly Output	402
Header Comment File	402
Function Comment Section	403
File Comment Section	403
Source Line Comments	403
 7. Assembler Directives	 405
Alignment Directives	406
Conditional Assembly Directives	407
Data Initialization Directives	408
File Inclusion Directives	409
Macro Definition Directives	409
Repeat Block Directives	412
Section Control Directives	413
Source Listing Directives	415
Symbol Definition Directives	415
Symbolic Debugging Directives	419
Miscellaneous Directives	420
 8. The elxr Linker	 423
Linker-Specific Options	424
Specifying the Program Entry Point	429

Configuring the Linker with Linker Directives Files	429
Linker Directives File Syntax	430
Setting Linker Options with the OPTION Directive	430
Setting Defaults with the DEFAULTS Directive	431
Defining a Memory Map with the MEMORY Directive	432
Defining a Section Map with the SECTIONS Directive	434
Modifying your Section Map for Speed or Size	450
Customizing the Run-Time Environment Program Sections	451
Symbol Definitions	455
Beginning, End, and Size of Section Symbols	455
End of Function Symbols	456
Linker Generated Tables	457
Forcing the Linker to Pull in a Module From a Library	458
Deleting Unused Functions	459
Advanced Linker Features	460
Code Factoring	460
Run-Time Clear and Copy Tables	463
Address Space Layout Randomization	464
 9. The ax Librarian	 465
Creating a Library from the Compiler Driver	466
Modifying Libraries	467
Librarian Commands	467
Librarian Command Modifiers	468
Librarian Options	469
Examples	470
Creating and Updating a Table of Contents	471
The Green Hills Large Archive Format	471
 10. Utility Programs	 473
The gaddr2line Utility Program	475
The gasmlist Utility Program	476
The gbin2c Utility Program	478
Global Options	478

Module Inclusion Options	481
Local Options	481
The gbincmp Utility Program	486
The gbuild Utility Program	488
Using gbuild	495
Implicit Dependency Analysis	497
The gcolor Utility Program	499
Color Configuration Files	500
Rules Files	501
The gcompare Utility Program	502
The gdump Utility Program	505
Debugging File Options	505
ELF File Options	507
COFF File Options	509
BSD a.out File Options	510
The gfile Utility Program	511
The gfunsize Utility Program	511
The ghide Utility Program	512
Using ghide	513
The gmemfile Utility Program	513
Data Splitting	516
The gnm Utility Program	520
General Options	521
ELF File Options	521
BSD a.out File Options	523
COFF File Options	523
Output Formats	524
The gpatch Utility Program	526
The gpjmodify Utility Program	527
Editing Existing .gpj Files	527
Creating a New .gpj File	530
Generating a Makefile	530
The gsize Utility Program	531
The gsrec Utility Program	532
S-Record Output Format	536

Data and Termination Records	537
Data Splitting	537
gsrec Examples	538
The gstack Utility Program	539
Preparing Your Program for gstack Analysis	540
Annotating Assembly Functions for gstack	541
Annotating C Functions for gstack	542
Working With Recursive Clusters	544
Specifying Interrupts	552
gstack Report Format	553
Using a gstack Options File	555
gstack Options	556
Specifying Functions as Arguments to gstack	559
The Miscellaneous Problems Report Section	559
Caveats	560
The gstrip Utility Program	561
The gversion Utility Program	563
The gwhat Utility Program	564
Version Extract Mode	565
Version Update Mode	566
The protrans Utility	567

Part III. Language Reference 573

11. Green Hills C 575

Specifying a C Language Dialect	576
ANSI C	577
C Extensions for ANSI C	577
ISO C99	583
Preprocessor Support in C99 or Later Dialects	584
Enforced Requirements in C99 or Later Dialects	584
Differences Between ANSI C and C99 or Later Dialects	584
ISO C11, ISO C17, and ISO C18	584
Notable Differences in ISO C11, ISO C17, and ISO C18	585
ISO C11, ISO C17, and ISO C18 Feature Support	586

GNU C	588
GNU C Only Extensions	589
GNU C/C++ Extensions	591
Japanese Automotive C Extensions	595
Sequence Related Undefined Behavior Checking	596
Type Qualifiers	597
__bytereversed	597
__bigendian and __littleendian	598
__packed	598
volatile	598
Using Volatiles with INTEGRITY TimeMachine Enabled ...	599
const	600
Assignment and Comparisons on struct and union Types	600
Bitfields	601
ANSI C Limitations	601
Signedness of Bitfields	601
Size and Alignment of Bitfields	603
Enumerated Types	604
Functions with Variable Arguments	605
The <stdarg.h> Facility	605
The asm Statement	606
The Preprocessor	607
Preprocessor Output File	607
Extended Characters	608
Compiler Support for Multibyte Characters	609
Kanji Character Support	610
Compiler Limitations	611
C Implementation-Defined Features	613
J.2 Undefined Behavior	613
J.3.1 Translation	614
J.3.2 Environment	614
J.3.3 Identifiers	616
J.3.4 Characters	616
J.3.5 Integers	618
J.3.6 Floating Point	619

J.3.7 Arrays and Pointers	620
J.3.8 Hints	620
J.3.9 Structures, Unions, Enumerations and Bit-fields	621
J.3.10 Qualifiers	622
J.3.11 Preprocessing Directives	623
J.3.12 Library Functions	623
J.3.13 Architecture	627
J.4 Locale-Specific Behavior	628
Implementation-Defined Behavior for Annex K	634
Non-Conformance with C Standards	634
All C Versions	634
C11 or Later	635
Attributes	637
Floating-Point Behavior	652
Basic	652
Accuracy of Floating-Point Results	653
Special-Case Values	655
Floating-Point Exceptions	658
Rounding Modes	659
Complex Numbers	660
Target and Configuration Caveats	661
Other General Notes	662
C Cautions	662
General Undefined Behavior	662
Preprocessing a File Twice May Produce Unexpected Output	662

12. Green Hills C++ 665

Specifying a C++ Language Dialect	666
Specifying C++ Libraries	667
Features Supported by Each C++ Dialect	668
Standard C++	669
Extensions Accepted in Normal C++ Mode	670
C++11	672
C++11 Feature Support	672
Important Backward Incompatibilities	675

C++14	676
C++14 Feature Support	676
Important Backward Incompatibilities	676
C++17	677
C++17 Feature Support	678
Implementation-Defined Features	679
Embedded C++	679
Differences Between Standard C++ and Embedded C++	679
Extended Embedded C++	681
Standard C++ with ARM Extensions	681
GNU C++	681
GNU C++ Extensions	682
Template Instantiation	685
Link-Once Template Instantiation	685
Instantiation Pragma Directives	686
Namespace Support	688
Dependent Name Lookup	688
Lookup Using the Referencing Context	689
Argument Dependent Lookup	690
Linkage	691
Post Processing in C++	691
The C++ decode Utility	692
C++03 Implementation-Defined Features	693
Deprecated C++ Headers	698
Non-Conformance with C++ Standards	699
All C++ Versions	699
C++03 or Later	699
C++11 or Later	705
C++14 or Later	708
C++17 or Later	709
General C++ Cautions	715
C++ Compatibility Across Green Hills Compiler Versions	715

13. Coding Standards 717

GHS Standard Mode	718
MISRA C 1998	718
Enforcing MISRA C 1998 Rules	720
MISRA C 1998 Rules Enforced by the Linker	720
MISRA C 2004	721
Enforcing MISRA C 2004 Rules	721
MISRA C 2004 Rules Enforced by the Linker	722
MISRA C 2004 Implementation	723
MISRA C 2012	723
Enforcing MISRA C 2012 Guidelines	724
MISRA C 2012 Guidelines Enforced by the Linker	725
MISRA C 2012 Guidelines Enforced at Run-Time	726
MISRA C 2004 and 2012 Implementation	726
Using MISRA C 2012 with Operating Systems and Libraries	728
Creating Your Own Coding Standards with Coding Standard Profiles	728
Specifying a Coding Standard Profile	729
Coding Standard Profile Syntax	729

14. Macros, Pragma Directives, and Intrinsic 733

Predefined Macro Names	734
Macro Formats	734
Macro Names Required by ANSI C and C++	734
Language Identifier Macros	736
Green Hills Toolchain Identification Macros	737
ARM64-Specific Predefined Macro Names	738
Endianness Macros	738
Data Type Macros	738
Floating-Point Macros	740
MISRA 1998 Macros	741
Memory Model Macros	742
Alignment and Packing Macros	742
Current File and Function Macros	743

Object Format Macros	743
Optimization Predefined Macro Names	744
C++ Macros	745
Operating System-Specific Macros	745
Miscellaneous Macros	746
Deprecated Macros	746
Pragma Directives	747
General Pragma Directives	748
Green Hills Extension Pragma Directives	752
_Pragma Operator	761
Intrinsic Functions	762
Exception Intrinsics	763
Arithmetic Operation Intrinsics	763
Synchronization Primitives	764
NEON Intrinsics	764
Cryptographic Extension Intrinsics	766
Endianness Intrinsics	766
Memory Barrier Optimization Intrinsics	766
Barrier Instruction Intrinsics	767
Hint Intrinsics	768
System Intrinsics	769
Built-In Intrinsics	770
Floating-Point Bit Manipulation Intrinsics	771
Static Assertions	772

15. Libraries and Header Files 773

The Green Hills Header Files and Standard Libraries	775
C and C++ Header File Directories	775
C and C++ Header Files	776
Library Directories	776
Libraries	776
Advanced Library Topics	779
Less Buffered I/O	779
64-bit Integer Arguments and printf	781
Calling printf from Assembly Code	782
Memory Allocation and malloc()	782
Using Thread-Safe Library Functions	783

Customizing Thread-Safe Library Functions	787
Customizing the Value of FOPEN_MAX	787
Passing Previously Closed FILE to stdio.h Functions	788
Customizing the Runtime-Constraint Handler for Annex K Functions	789
Green Hills Standard C Library Functions	790
Standard C Functions	791
Wide Character Functions	810
libmath.a Functions	811
libind.a Functions	823
libstartup.a Functions	823
Standard Function Names Converted by the Linker	823
Customizing the Run-Time Environment Libraries and Object Modules	825
Creating a Project with a Customizable Run-Time Environment	825
Startup Module crt0.o	827
Low-Level Startup Library libstartup.a	828
Low-Level System Library libsys.a	829
Board Initialization Library libboardinit.a	834
Features that Depend on the Default Green Hills Startup Code	835

16. Mixing Languages and Writing Portable Code 837

Mixing Languages	838
Initialization of Libraries	838
Examples of main() Programs	839
Performing I/O on a Single File in Multiple Languages	839
C Routines and Header Files In C++	840
Using C++ in C Programs	841
Function Prototyping in C vs. C++	842
Writing Portable Code	843
Compatibility Between Green Hills Compilers	843
Word Size Differences	844
Endianness Problems	845
Alignment Requirements	845

Structures, Unions, Classes, and Bitfields	846
Assumptions About Function Calling Conventions	847
Pointer Issues	848
NULL Pointer	849
Character Set Dependencies	849
Floating-Point Range and Accuracy	850
Operating System Dependencies	850
Assembly Language Interfaces	850
Using Assembly Code with INTEGRITY TimeMachine Enabled	851
Evaluation Order	852
Machine-Specific Arithmetic	852
Illegal Assumptions About Compiler Optimizations	853
Memory Optimization Restrictions	854
Problems with Source-Level Debuggers	855
Problems with Compiler Memory Size	856

17. Enhanced asm Macro Facility for C and C++ **857**

Definition of Terms	859
Using and Defining the asm Macro Facility	859
Pseudo-Code Definition	860
Use	860
Using asm Macros	861
Definition	863
Storage Modes	863
asm Body	864
ARM64 asm Macros	865
Guidelines for Writing asm Macros	866

18. GNU Extended Assembly **869**

Extended GNU Support	870

19. The ELF File Format **873**

Formats of Relocatable and Executable Files	874

32-Bit ELF Data Types	875
ELF Headers	876
ELF Identification	879
ELF Sections	881
Section Headers	881
Symbol Tables	889
Symbol Binding	890
Symbol Type	891
Symbol Values	891
String Tables	892
Program Headers	893
Program Types	894
Program Attribute Flags	895

Part IV. Appendices **897**

A. Coding Standard Profile Reference **899**

GHS Standard Mode	900
Declarations, Definitions, and Initialization	900
Expressions and Control Flow	901
Preprocessing Directives	902
Character Set Portability	902
Standard Libraries	903

B. GNU Options **905**

C. Green Hills Project File Format **909**

Green Hills Project File Syntax	910
The Header Section	911
The Project Options Section	912
The Children Section	912
Top Project Directives	914
Group 1 Directives	914
Group 2 Directives	918

Conditional Control Statements	918
Valid Conditions	919
Special Conditions for Select One Projects	921
Macro Functions	922
File Types	923

D. Secure Coding Practices 927

Summary of Available Features	928
Static Checks	928
Run-Time Checks	929
Recommended Coding Practices	929
Avoid setjmp(), longjmp(), and Exceptions	929
Avoid Threads	929
Limit Dynamic Memory Allocation to System Initialization	930
Initialize Locals at Declaration	930
Strategies for Ensuring Control-Flow Integrity	930
Stack Canary	931
Shadow Stack	931
Forward-Edge CFI Attacks and Indirect Function Call Checks	932
Supporting CERT C Rules	932
Green Hills Secure Coding Options Files	944

E. Porting an Existing Project to Use Build Configurations 947

Build Configuration Quick Start	948
Setting Configuration-Specific Files and Options Outside of Build Configuration Files	949
Defining Unique Output Directories for Build Configurations	950
Updating Output Directories	950
Updating Input Directories	951
Handling Custom File Types	952
Verifying Correctness	952
Helpful Tricks	953

Additional Options to Check	954
F. Customization Files	955
Components of Customization Files	956
Customization File Definitions	957
Extended Examples	961
Customization File Syntax	967
FileType Syntax	967
Context Sensitive Variables	969
G. The MULTI Eclipse Plug-in	977
Prerequisites to Installing MULTI for Eclipse	979
Installing MULTI for Eclipse	980
Creating a Green Hills Project from Within Eclipse	981
Creating an INTEGRITY Project	982
Adding a Source File to Your Project	984
Changing Compiler Options	985
Building and Running Your Project	986
Debugging Your Project with the MULTI Debugger	988
H. Third-Party Software Licensing Information	991
Edison Group License	992
Other Licenses	992
Index	995
Index of Driver Options	1043

Preface

Contents

About This Book	xxvi
The MULTI Document Set	xxvi
Conventions Used in the MULTI Document Set	xxvii
The Green Hills Toolchain	xxx

This preface discusses the purpose of the manual, the MULTI documentation set, and typographical conventions used. It also provides an introduction to the Green Hills toolchain.

About This Book

This book provides comprehensive documentation of 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 compiler features, and descriptions of 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 (C, C++), including macros, `#pragma` directives, libraries, headers, and optimizations.
- *Part IV: Appendices* — Documents ancillary features and options, such as how to use GNU compiler options with the Green Hills toolchain.



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* — Provides an introduction to the MULTI Integrated Development Environment and leads you through a simple tutorial.
- *MULTI: Licensing* — Describes how to obtain, install, and administer MULTI licenses.

- *MULTI: Managing Projects and Configuring the IDE* — Describes how to create and manage projects and how to configure the MULTI IDE.
- *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 configure connections to your target.
- *MULTI: Debugging* — Describes how to set up your target debugging interface for use with MULTI and how to use the MULTI Debugger and associated tools.
- *MULTI: Debugging Command Reference* — Describes how to use Debugger commands and provides a comprehensive reference of Debugger commands.
- *MULTI: Scripting* — Describes how to create MULTI scripts. Also contains information about the MULTI-Python integration.
- *MULTI: Toolchain Error Messages and Other Diagnostics* — Provides a sequential listing of MULTI diagnostic numbers and messages for the compiler, assembler, and linker. Where possible, includes information about associated resolutions and workarounds.

For a comprehensive list of the books provided with your MULTI installation, see the **Help** → **Manuals** menu accessible from most MULTI windows.

All books are available in one or more of the following formats:

- Print.
- Online help, accessible from most MULTI windows via the **Help** → **Manuals** menu.
- PDF, available in the **manuals** subdirectory of your MULTI or compiler 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).

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
<i>italic type</i>	Replaceable text	-o filename
	A new term	A task may be called a <i>process</i> or a <i>thread</i>
	A book title	<i>MULTI: Debugging</i>
monospace type	Text you should enter as presented	Type <code>help command_name</code>
	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	<code>int a = 3;</code>
	Input/output	<code>> print Test</code> <code>Test</code>
	A function	<code>GHS_System()</code>
ellipsis (...) (in command line instructions)	The preceding argument or option can be repeated zero or more times.	debugbutton [name]...
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.	<code>> print Test</code> <code>Test</code>

Convention	Indication	Example
pipe () (in command line instructions)	One (and only one) of the parameters or options separated by the pipe or pipes should be specified.	call <i>func</i> <i>expr</i>
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.	<i>.macro name [list]</i>

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

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

The formatting indicates that:

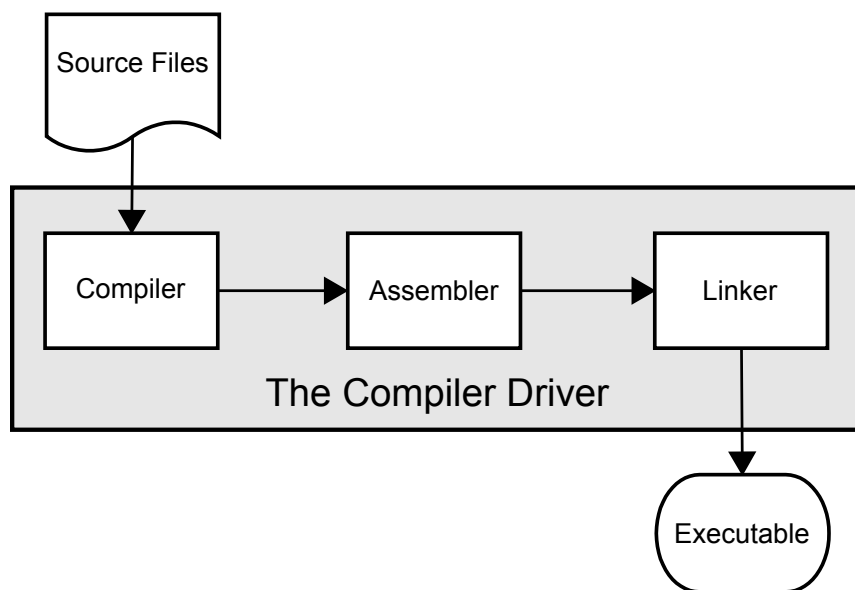
- **gxyz** is a command.
- The option *-option* should either be replaced with one or more appropriate options or be omitted.
- The word *filename* should be replaced with the actual filename of an appropriate file.

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

The Green Hills Toolchain

The C and C++ Compiler Driver

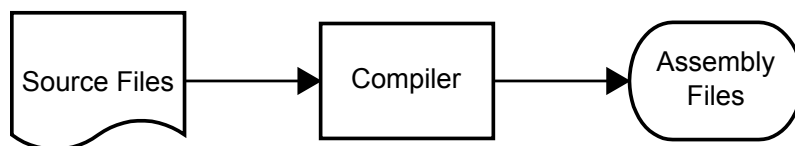
The compiler driver directs and coordinates the actions of the compiler, assembler, and linker. The driver is the primary command line interface to the Green Hills toolchain and is commonly used to transform high-level source files into an executable, as shown below.



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 files.



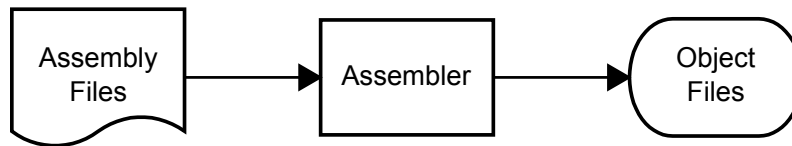
There are separate compilers for C and C++, each composed of a language-specific front end, a global optimizer, and a target-specific code generator. Compatible

subroutine calling conventions allow modules written in different languages to be combined.

The compilers are automatically invoked by the driver whenever necessary. Importantly, unlike other toolchain components, the compilers are designed to be invoked only through the driver or a higher level interface to the driver.

The asarm64 Assembler

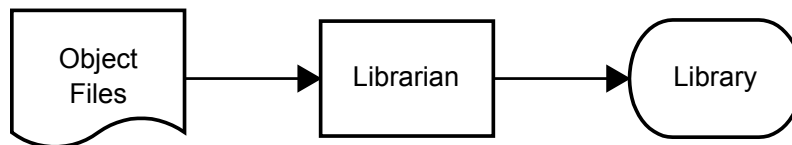
The Green Hills assembler, **asarm64** produces object files by translating ARM64 assembly language into ARM64 machine code and data formats.



The assembler is automatically invoked by the driver whenever necessary. For a detailed description of the assembler's operations and information about invoking it directly, see Chapter 6, “The asarm64 Assembler” on page 389.

The ax Librarian

The Green Hills librarian, **ax**, combines object files into libraries.

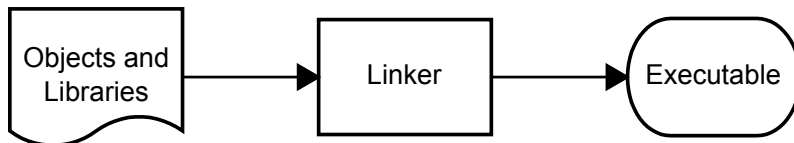


Created libraries can be searched by the linker to resolve external references. The linker will extract any needed object files and incorporate them into the executable.

The librarian is automatically invoked by the driver whenever necessary. For a detailed description of the librarian's operations and information about invoking it directly, see Chapter 9, “The ax Librarian” on page 465.

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 automatically invoked by the driver whenever necessary. For a detailed description of the linker's operations and information about invoking it directly, see Chapter 8, “The elxr Linker” on page 423.

Optimized Libraries and Header Files

Green Hills provides optimized libraries and header files that support the standard libraries specified by each language. The driver selects the appropriate libraries when linking, and header files when compiling. For more information, see Chapter 15, “Libraries and Header Files” on page 773.

Utility Programs

The Green Hills utility programs provide additional ways to analyze and transform the object files, libraries, and executables produced with the Green Hills toolchain. For more information, see Chapter 10, “Utility Programs” on page 473.

The Builder

The Builder (also called the MULTI Builder) is an important higher level interface to the driver. The term “Builder” is used to encompass the functionality provided by the **gbuild** utility and the graphical MULTI Project Manager. The Builder exists to make the actions of the toolchain more intelligent and user-friendly. For example, the Builder can assist in speeding recompilation and in managing large, complex programming projects. For more information about the MULTI Project Manager, see *MULTI: Managing Projects and Configuring the IDE*. For more information about the **gbuild** utility, see “The gbuild Utility Program” on page 488.

Part I

Using the MULTI Compiler

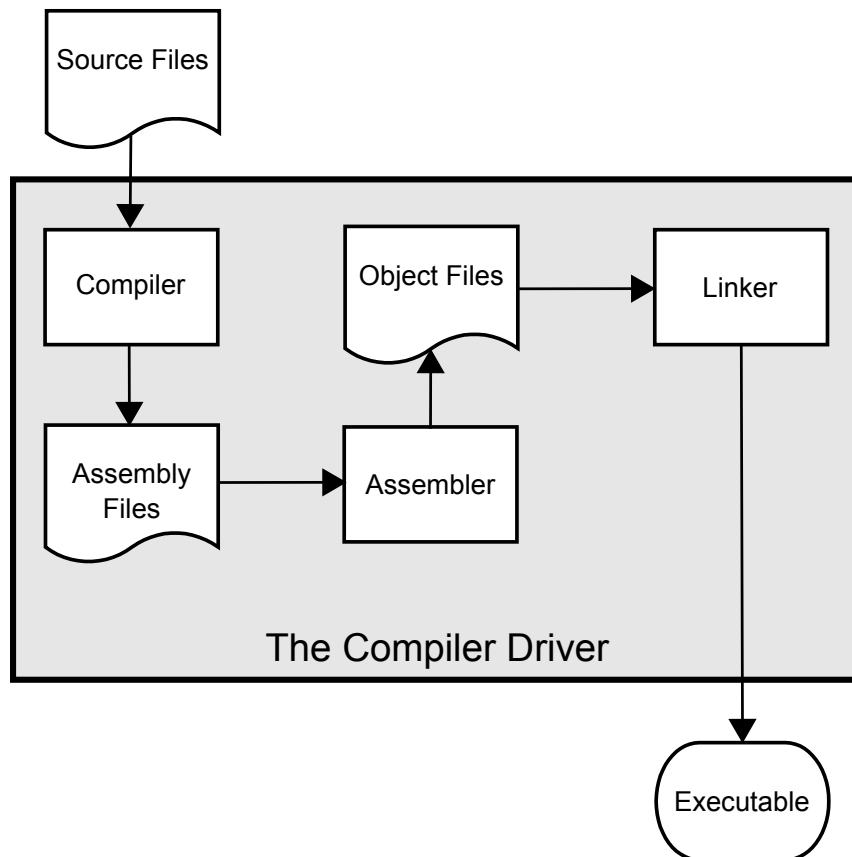
Chapter 1

The Compiler Driver

Contents

Compiler Driver Syntax	5
Building an Executable from C or C++ Source Files	6
Working with Input Files	7
Generating Other Output File Types	10
Controlling Driver Information	16
Using a Driver Options File	17
Using Makefiles	18

The compiler driver directs and coordinates the Green Hills toolchain. Using the driver, you can employ the compiler, assembler, and linker to process a variety of different input files into a variety of different output files. This chapter explains how to use the driver from the command line and introduces some of its most important tasks. For example, the driver can produce an executable from high-level source files by coordinating the toolchain, as illustrated below.



1. The appropriate compiler generates ARM64 assembly language files.
2. The **asarm64** assembler translates the ARM64 assembly files into object files.
3. The **elxr** linker combines the object files to create an executable (including object files in libraries).

The driver accepts many types of input files (see “Working with Input Files” on page 7) and can produce many types of output files (see “Generating Other Output File Types” on page 10).

The driver can also be 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 “The Builder” on page xxxii.

Regardless of how you invoke the driver, it should be your primary means of accessing toolchain components. The driver (especially in combination with the Builder) will intelligently resolve a variety of situations that individual toolchain components, when accessed directly, will not. Thus, although you can invoke the assembler and linker directly, you should do so as little as possible.

Compiler Driver Syntax

The syntax for using the compiler driver is:

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

where:

- ***driver*** is one of the following:
 - **ccarm64** — For programs with only C and assembly code.
 - **cxarm64** — For programs with at least some C++ code.
- *file* is one or more of the following file types:
 - 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 77.

When specifying multiple files or options, separate them with spaces.

In general, the same driver options should be passed across all build steps, regardless of whether you are currently compiling, assembling, archiving, or linking. Exceptions

to this rule include the options **-L** and **-I**, which can safely be omitted on non-link steps. In addition, the options **-D** and **-I** may vary between compilations and can safely be omitted during the link step if preprocessing is not involved.

The driver processes files and options in the following manner:

- First, the driver reads all options. If two options represent different choices for the same feature, the later choice will override the earlier one. If the driver encounters an unrecognized or invalid option, it will ignore the option and issue a warning or error.
- Second, 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. If the driver encounters a file with an unrecognized file extension, it will pass the file to the linker. The linker will attempt to identify the file and use it appropriately.

Building an Executable from C or C++ Source Files

When the driver receives source language files as input, it invokes the appropriate components to produce an executable.

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

```
ccarm64 hello.c
```

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

```
ccarm64 hello.cxx
```

or

```
cxarm64 hello.cxx
```

Either driver may be used because the **.cxx** extension tells the driver that this program contains C++. 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 that 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 38.

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

```
ccarm64 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**.

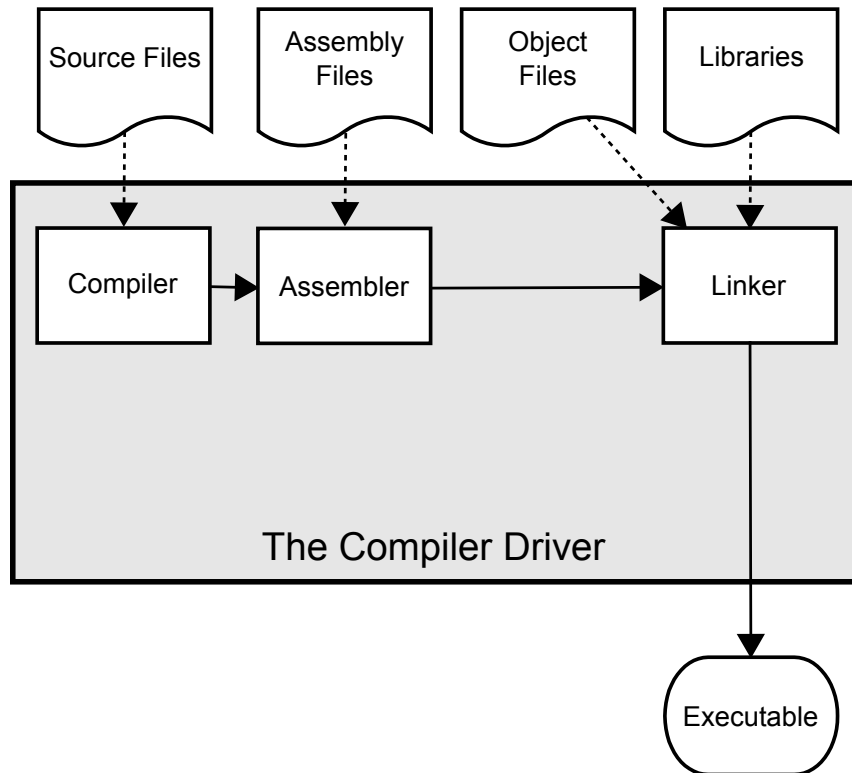
Working with Input Files



Note

The example command lines in this section, and in the remainder of this chapter, are shown using the C driver, **ccarm64**. They will also work with the C++ driver, **cxarm64**.

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 actions to perform on it) from its extension. For example, a file with a **.c** is treated as a C source file and will be compiled with the C compiler, regardless of whether the C or C++ driver is invoked.

The following is a partial list of recognized file extensions:

- C source files: **.c**, **.i**
- C++ source files: **.C**, **.cc**, **.cpp**, **.cxx**
- Assembly language files: **.s**, **.asm**
- Assembly language files with C preprocessor directives: **.arm64**, **.a64**, **.sx**, **.cppasm**
- Object files: **.o**
- Library files: **.a**

- Linker directives files: **.ld**

If the driver passes an input file to the compiler, note that the DOS and Windows EOF marker will be regarded as a true end-of-file indicator (^Z, decimal character 26).

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:

```
ccarm64 -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:

```
cxarm64 -dotciscxx files
```

Note that these options should not be used in the Builder.

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:

```
ccarm64 hello.c foo.cc bar.s baz.o libfoo.a
```

This works because the C and C++ drivers handle source files based on each source file's extension, not the name of the driver. A source file ending in **.cc** will always be compiled as a C++ source file and a source file ending in **.c** will always be compiled as a C source file.

Specifying the Language When Linking a Program

When a program contains at least some C++ code, one or more of the following are necessary to link the program properly:

- Use the C++ driver, **cxarm64**.

- Pass `-language=cxx`.
- Name 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 272.

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.

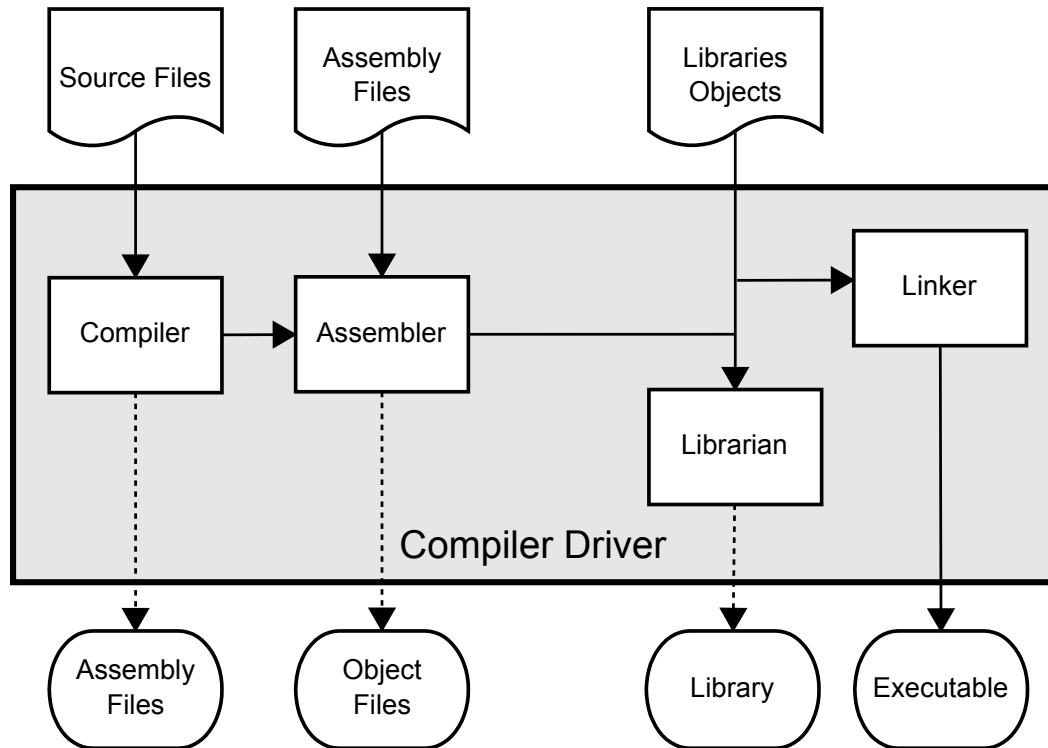
To pass a linker directives file to the driver, add it to the command line as follows (no option is required; the file is recognized by its extension):

```
ccarm64 hello.c mylinkfile.ld
```

This command overrides the default, included linker directive files that are automatically selected by the driver (see “Working with Linker Directives Files” on page 60). 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 429.

Generating Other Output File Types

The driver can be instructed to halt a compilation process at various stages in order to produce assembly files, object files, libraries, or other forms of output. The diagram below shows possible intermediate stopping points in a traditional compilation process.



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

```
ccarm64 hello.c -S
```

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

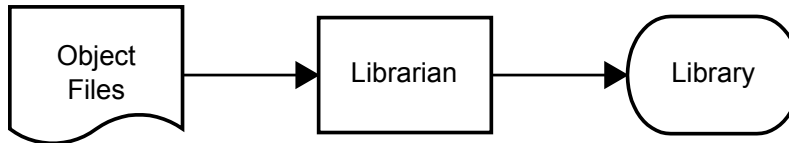
To instruct the driver to halt the build process after the compiler has generated assembly files and the assembler has translated them into object files, use the **-c** option as follows:

```
ccarm64 hello.c -c
```

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

Creating Libraries

You can direct the driver to collect high-level source files, assembly language files, and object files into a library. The driver will invoke the compiler and/or the assembler (if required) to produce object files, and then invoke 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:

```
ccarm64 hello.c foo.o -archive -o libfoo.a
```

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.

For more information, see “Creating a Library from the Compiler Driver” on page 466.

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:

```
ccarm64 hello.c -archive -o libfoo.a
```

To perform other operations on libraries, you must invoke **ax** directly. For more information, see Chapter 9, “The ax Librarian” on page 465.

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. To set them manually, see Appendix C, “Green Hills Project File Format” on page 909.

-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 466.
-c
Produces an object file (called input-file.o) for each source file.
-E
Runs the preprocessor and writes the preprocessed file output to <code>stdout</code> unless the -o option is specified. This option can be useful when debugging preprocessor macros and <code>#include</code> files. Each preprocessing directive is replaced with a blank line, unless there are several lines in a row. In this case, the lines are replaced with a single <code>#line</code> marker that specifies the line number of the next line of code in the original file.
-dM --list_macros
--no_list_macros (default)
Used to modify -E and ignored otherwise. The -dM or --list_macros option will substitute the normal output of -E with a list of <code>#define</code> symbols active after a source file is preprocessed. By default, output is sent to <code>stdout</code> . If there is only one source file, output may be sent to a file using the -o option. If the output file has a suffix, it must be .txt or a suffix unknown to the driver.
-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 466.
-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 <code>#include</code> 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**

If a **.i** file produced by **-E** or **-P** is passed back into the compiler, it will be preprocessed a second time. In rare cases, this can result in different output than if the original source file were compiled directly (see “Preprocessing a File Twice May Produce Unexpected Output” on page 662). To avoid such cases, do not pass the output of **-E** or **-P** back through the compiler. Instead, use **-keepcppfiles** or pass the original source file to the compiler twice: first to preprocess and second to compile.

**Note**

See also the driver options associated with the **Linker**→**Generate Additional Output** option in “Linker Options” on page 227.

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. For more information, see “Generating Dependency and Header File Information” on page 23.

.dep
Program dependency files generated by the linker. For more information, see “Generating Dependency and Header File Information” on page 23.
.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 132.
.env
A file used to determine whether environment variables or macros affecting a project have changed. Files of this type are read and written only by the toolchain.
.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
An INTEGRITY-only file that is 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.
.lst
An assembler source listing that is written by the toolchain, and read only by the user. See “Assembler Options” on page 224.
.map
A map file generated by the linker that contains information about how the program is laid out in memory. Files of this type are written by the toolchain, and read only by the user. See “Linker Output Analysis” on page 242.

.mem
An additional copy of the executable image that has been translated by the gmemfile utility. Files of this type are written by the toolchain, and read by non-GHS utility programs. See “Linker Options” on page 227.
.run
An additional copy of the executable image that has been translated by the gsrec utility. Files of this type are written by the toolchain, and read by non-GHS utility programs.
.time
File generated by the Project Manager when a file being processed does not generate an output file. This allows the Project Manager to record if and when the item was last processed for dependency tracking. Files of this type are 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.

@filename
Instructs the driver to read command line arguments from <i>filename</i> .
-#
Displays the command lines generated by the driver without executing them. These command lines are used to call the compiler, assembler, and linker for processing input files. 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 they are being executed. These command lines are used to call the compiler, assembler, and linker for processing input files.

Using a Driver Options File

When using multiple driver arguments, you may find that your command lines become overly long and hard to read. To address this issue, you can pass most arguments to the driver using a plain text file and the *@filename* option. A file passed using this option will be parsed according to the following rules:

- All characters other than spaces, tabs, newline characters, and double quotes will be treated as literal. As such, do not include comments.
- 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.
- If an argument begins with a double quote, everything on the line up to the next double quote will be part of the argument. The quotes will be discarded. If a double quote is in the middle of an argument, it will not be discarded, but will cause any whitespace on the line up to the next double quote to be included in the argument.

The following argument types cannot be included in driver options files:

- Linker directives files.
- Linker options or assembler options. Instead, use **-asmcmd=filename** (see “Assembler Options” on page 224) or **-lnkcmd=filename** (“Linker Options” on page 227) to pass the arguments directly to the assembler or the linker, respectively.
- The **-bsp** and **-layout** options.

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 arguments contained in **myoptfile**, enter:

```
ccarm64 @myoptfile
```

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

- Pass files **hello.c**, **foo.c**, and **bar.c** to the toolchain.
- 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**).

As noted earlier, **-bsp** and **-layout** cannot be used in a driver options file. They can be specified separately, however. The following command shows their combined use to generate code tailored to the instruction set associated with the `generic` board. This command also assigns the program to memory regions of the board using a board-specific linker directives file:

```
ccarm64 @myoptfile -bsp=generic -layout=romrun
```

Using Makefiles

This section describes how to port existing makefiles for use with Green Hills compilers.

In each of the following examples, a makefile is ported from a fictional compiler (**qcc**) to the Green Hills compiler (**ccarm64**). The fictional compiler operates using a recognizably 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 **ccarm64**, make the following changes:

- Change the invocation of **qcc** to **ccarm64**.
- Change the optimization option **-O2** to a Green Hills option, such as **-Ogeneral** (which enables general optimizations).

The resulting **ccarm64** makefile will look like the following:

```
scanprog: scanprog.c
    ccarm64 -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
LFLAGS=-L../lib -lxyzlib
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 **ccarm64**, change the macros as follows:

- Change:

```
CC=qcc
```

to:

```
CC=ccarm64
```

- 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 ARM64 assembly file, and one C++ file are linked together into a single executable:

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

```
myprog: $(OFILES)
    gcc $(OFILES) -o myprog
```

```
myprog.o: myprog.c utils.h
    gcc -c myprog.c
```

```
utils1.o: utils1.c utils.h
    gcc -c utils1.c
```

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

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

Porting this example to use **cxarm64** is easy because the Green Hills drivers recognize filename extensions and invoke the correct compiler or assembler automatically. Consequently, the only changes needed are to replace references to **gcc**, **qasm**, and **qc++** with **cxarm64**. The corrected makefile will read as follows:

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

```
myprog: $(OFILES)
    cxarm64 $(OFILES) -o myprog
```

```
myprog.o: myprog.c utils.h
    cxarm64 -c myprog.c
```

```
utils1.o: utils1.c utils.h
    cxarm64 -c utils1.c
```

```
utils2.o: utils1.s
    cxarm64 -c utils1.s
```

```
interface.o: interface.cpp
    cxarm64 -c interface.cpp
```


Example 1.5. Using Wholeprogram Interprocedural Optimizations With Makefiles

When building with **-Owholeprogram** enabled (see “Interprocedural Optimizations” on page 122), the compiler is invoked in at least three separate passes during its execution:

- The first pass produces an inline file for each C or C++ source file. These files are named *input-file.inf*.
- The middle pass reads the *.inf* files, analyzes them for interprocedural optimization opportunities, and writes them back out with additional information.
- The final pass reads the *.inf* files, along with the original source files. The compiler uses the information added in the second pass to perform additional optimizations as it compiles the source to object code or assembly.

When building a program using makefiles, each of these compiler passes may be specified individually to provide greater control over how each pass is performed. In the following example, three C files are compiled together using wholeprogram optimizations, which can increase program speed while reducing size:

```
CC=carm64
SRCS=src/file1.c src/file2.c src/file3.c
INFS=objs/file1.inf objs/file2.inf objs/file3.inf
OBJS=objs/file1.o objs/file2.o objs/file3.o
CFLAGS=-object_dir=objs -Owholeprogram -Ogeneral -I.

myprog: ${OBJS}
    ${CC} -o myprog ${OBJS}

# This is the first pass, where an .inf file is generated for
# each C file. The compiler driver is invoked with the -Q option
# to tell it to generate the .inf file instead of compiling the
# source file to object code.
objs/%.inf: src/%.c ${SRCS}
    ${CC} ${CFLAGS} $< -Q

# This is the middle pass, where the .inf files are processed.
# The driver is invoked with the -Qnormalize option and a list
# of all of the .inf files for the program. Note that no source
# files are specified. A dummy file "qnormalize" is used to
# guarantee that this step is completed before moving on to the
# final pass.
qnormalize: ${INFS}
    ${CC} ${CFLAGS} ${INFS} -Qnormalize
    touch qnormalize
```

```
# This is the final pass, where individual source files are
# compiled to object code or assembly. The driver is invoked
# with the -Qfinal and -c options and a list of all of the
# .inf files for the program.
objs/%.o: src/%.c objs/%.inf qnormalize
    ${CC} ${CFLAGS} ${INFS} $< -c -Qfinal
```

Note that when using wholeprogram optimizations with makefiles, as with **gbuild**, you must make all parts of your program available to the compiler for analysis (see “Selectively Excluding Compilation Units” on page 339).

Wholeprogram optimizations require that the compiler is provided a list of all externally referenced functions and variables. These symbols are found in object files or libraries that are linked into the program but do not have corresponding **.inf** files (see “Symbols Referenced Externally” on page 124). When building with makefiles, these pre-compiled objects and libraries can be passed to the driver for the middle pass, along with **-Qnormalize** and the list of **.inf** files. Doing so will instruct the compiler to automatically analyze the objects for external references. (This is in lieu of using the **-external** option to specify the external references directly.) While the **-Qnormalize** approach is not required, it is *strongly* encouraged. It will reduce the chances of error due to incongruities between possible source code changes and the list of externally referenced symbols. For more information about **-Qnormalize**, see the next section.

Driver Options for Individual Compiler Passes

The following options should be used when invoking the driver to perform the compiler passes related to interprocedural optimizations, as shown in the previous example. (See also the **-Q** option in “Driver Options for Intermediate Forms of Output” on page 12.)

-Qnormalize

Instructs the driver to process a set of inline (**.inf**) files. The set of **.inf** files provided as input should be identical to the set provided to the **-Qfinal** option.

Build options such as optimization settings should be identical across passes. This option is not compatible with **gbuild**.

-Qfinal

Instructs the driver to compile a list of one or more source files to object code (with **-c**) or assembly (with **-S**). Along with the source files to compile, the driver should be provided a list of **.inf** files identical to that used in the preceding pass by the **-Qnormalize** option.

Build options such as optimization settings should be identical across passes. This option is not compatible with **gbuild**.

Green Hills Equivalents to GNU Tools

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

GNU	Green Hills
CC=gcc	CC=ccarm64
CXX=g++	CXX=cxarm64
LD=ld	LD=cxarm64
AR=ar	AR=ax AR=cxarm64 -archive (recommended)

Note that because **AR=ax** invokes the librarian directly, it will not produce debug information.

Generating Dependency and Header File Information

These options provide information about your makefile structure:

-H | --trace_includes

Prints to `stderr` a list of files opened by `#include` directives during 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 | **--make_depends**

--no_make_depends

-MMD | **--user_make_depends**

-MF=*filename*

The option **--no_make_depends** disables the generation of makefile dependency files. Note that this is the default option.

The **-MD** option 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 file extensions to prevent file overwrites due to duplicate names. Note that **-MF** has no effect if **-syntax** is passed.

The **-MMD** option behaves the same as **-MD**, but does not include the standard C or C++ headers or headers from system `#include` directories (see “System Include Directories” on page 307).

The **-MF**=*filename* option changes the generated dependency filename. This option is only effective when specified with **-MD** or **-MMD**. If specified for multiple source files, this option is ignored with a warning.

Note that the above options only apply to the compiler driver. The Builder will automatically use these options to track dependencies.

Chapter 2

Developing for ARM64

Contents

ARM64 Characteristics	26
Structure Packing	30
Specifying an ARM64 Target	35
Enabling Debugging Features	38
Using Your Own Header Files and Libraries	56
Controlling the Assembler	59
Controlling the Linker	59
Text and Data Placement	61
Customizing the Green Hills Run-Time Environment	73
Other Topics	74

This chapter describes the ARM64 target environment as well as some of the Builder and driver options that control features of that environment.

ARM64 Characteristics

Addressing — Memory is byte-addressed with 64-bit addresses.

Byte Ordering — By default, bytes are ordered in the little endian format, where the least significant byte of a multiple-byte value is stored at the lowest address:

Address 0	Address 1	Address 2	Address 3
Least Significant Byte			Most Significant Byte

The reverse byte order, big endian, is supported for data only. Instructions are always output in little endian format regardless of data endianness. For more information, see “Endianness” on page 79.

Bitfields — 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.

Character Encoding — ASCII.

Stack Characteristics — The stack is 16 byte aligned.

Floating-Point Support — Floating-point values use IEEE-754 format (32- and 64-bit) and are in the same byte order as other values. Although the compiler uses the IEEE-754 floating-point format, floating-point operations may not generate the precise values required by the specification. For general and architecture-specific details, see “Floating-Point Behavior” on page 652.

Instruction Set Extensions — The ARM64 architecture includes the NEON instruction set, which allows single instruction multiple data (SIMD) operations on vectors of integer and floating-point values. In addition to allowing for powerful vectorization opportunities, NEON intrinsics can be manually specified to explicitly perform several operations in parallel. For more information, see “NEON Intrinsics” on page 764 and “SIMD Vectorization” on page 346. The ARMv8.1, ARMv8.2, and Cryptographic Extension instructions are recognized by the assembler.

C and C++ Data Type Sizes and Alignment Requirements

This section lists the data type sizes and alignment requirements for C and C++. The size of a data type is the number of bits the type requires. The alignment requirement of a data type tells the compiler how to control the type's starting address in memory.

An instance of a data type is aligned if the compiler places it at an address offset $a*n$ bits from the beginning of the section, where a is the type's alignment requirement, and n is an integer greater than or equal to 0. To ensure that multiple instances of a data type remain aligned when placed in an array, a data type's size is always a multiple of its alignment requirement.

Note that each structure, union, and array is aligned to the maximum alignment requirement of any of its components.

C/C++ data type	Size	Alignment Requirement
char	8	8
short	16	16
int	32	32
long	64	64
long long	64	64
float	32	32
double	64	64
long double	64	64
* (pointer)	64	64
enum (default)	32	32



Note

To reduce the enum type's size and alignment requirement 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 206.

Register Usage

The ARM64 processor has thirty-one 64-bit registers accessible for integer values. There are thirty-two 128-bit registers available for floating-point values and SIMD vectors.

The general registers are as follows:

General Register	Alternate Name	Use
r0-r7		Argument registers, not saved across calls.
r8		Used as a return pointer for functions that return a structure greater than 128 bits in size. Otherwise, treated as a temporary register; not saved across calls.
r9-r18		Temporary registers, not saved across calls.
r19-r28		Permanent registers, saved across calls.
r29	FP	Either permanent register or frame pointer.
r30	LR	Link register.
SP		Stack pointer.
v0-v7		Argument registers, not saved across calls.
v8-v15		Permanent registers, only bottom 64 bits saved across calls.
v16-v31		Temporary registers, not saved across calls.

In addition, r0-r30 may be referred to as x0-x30 to indicate that the full 64 bits of the register are being used, or as w0-w30 to indicate that only 32 bits are used. The registers v0-v31 may be referred to with the b, h, s, d, or q prefix, to indicate sizes of 8, 16, 32, 64, or 128 bits, respectively.

Calling Conventions

This section provides an overview of ARM64 calling conventions.

Argument Type Conversion

After evaluation, arguments are converted as follows.

- If a function prototype exists and there is a corresponding formal parameter for an integral or floating-point argument, then the argument is converted to the type of the formal.
- If there is no visible prototype, or if the argument corresponds to the “...” parameter, then a 32-bit floating-point value is extended to 64 bits and integers less than 32-bits are extended to 32-bits.

Argument Passing

The compiler conforms to the Procedure Call Standard for the ARM 64-bit architecture. Arguments are passed according to the following rules:

- Each integral argument is passed in the argument registers (if possible).
 - When the size of an argument is less than 64 bits, it is not necessary to sign or zero-extend the high bits of the register.
- Floating-point arguments are passed based on whether hardware floating point is disabled (**-fsoft**):
 - If hardware floating point is enabled, each floating-point argument is passed in floating-point argument registers.
 - If hardware floating point is disabled, floating-point arguments are passed using integral argument registers.
- Structures are passed based on the following rules:
 - Homogeneous aggregates are passed in float/vector registers.
 - Structures of 128 bits or less are passed in one or two integer registers.
 - Structures greater than 128 bits are copied by the caller, and a pointer to the copy is passed instead.

- If no registers are available for the argument, the argument is passed on the stack.

Accessing the Stack Frame

Accesses to parameters or local stack storage are normally relative to the stack pointer, `sp`.

Return Value Passing

Values are returned based on the following rules:

- Excepting homogeneous aggregates, structures of more than 128 bits are returned to the location pointed to by the `r8` register.
- Other return types (including homogeneous aggregates) are returned as in the same manner that an argument of that type would be passed to a void function that takes a single argument of the same type.

Structure Packing



Note

Support for structure packing in C++ is limited. See the release notes for more information.

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” on page 31
- “Using `#pragma pack` to Pack All Instances of a Structure Type” on page 32
- “Using the Packing Builder Option to Pack All Structures” on page 33
- “Pointing to Packed Structures with the `__packed` Type Qualifier” on page 34

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 variable, 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 variable so that it is naturally aligned.

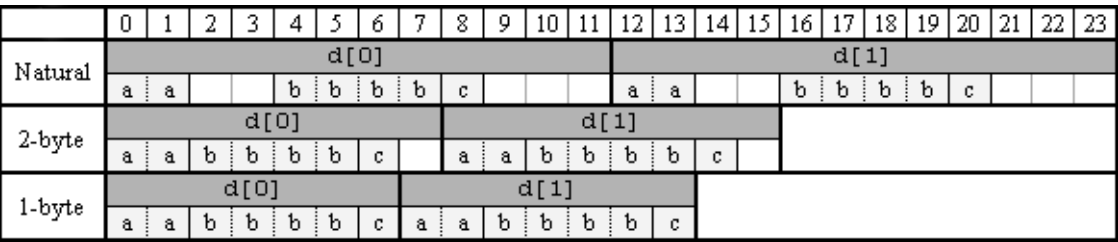
Structure packing reduces the amount of padding the compiler inserts between fields in a structure. When using structure packing, you give the compiler a *packing alignment*. When the compiler places a field in a packed structure, it aligns that field to its alignment requirement, or the packing alignment, 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



Note

Support for structure packing in C++ is limited. See the release notes for more information.

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 33 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`.



Warning

When building applications for INTEGRITY, you must ensure that there is no structure packing in effect when including an INTEGRITY or system header.

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, 8, or 16.

**Note**

When using this Builder option on a project, you should 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 34 for more information.

**Warning**

This option is not supported with INTEGRITY; it may cause your program to use structure layouts that are incompatible with the rest of INTEGRITY. If you need to use structure packing in an INTEGRITY application, you should use either the `__packed` qualifier or the `#pragma pack` variants to mark only those structures that require packing. For more information, see “Pointing to Packed Structures with the `__packed` Type Qualifier” on page 34 and “Using `#pragma pack` to Pack All Instances of a Structure Type” on page 32.

Pointing to Packed Structures with the `__packed` Type Qualifier

**Note**

Support for structure packing in C++ is limited. See the release notes for more information.

The `__packed` type qualifier tells the compiler that a particular pointer might point to a misaligned structure or field. Because ARM64 processors handle these accesses in hardware, the Green Hills compiler will not generate additional code for them, and so declaring a pointer with `__packed` does not affect programs compiled for ARM64. We recommend that you use `__packed` to make your code portable; if you compile your code for a different architecture that does not handle misaligned accesses in hardware, the 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 should 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.

Specifying an ARM64 Target

The Builder and compiler drivers support many of the most popular ARM64 boards and the entire ARM64 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 ARM64 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 69.

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 **Target Board** 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=help`.

ARM64 Processor Variants

The following ARM64 processor variants are supported:

Processor	Driver Option -cpu=
ARMv8	armv8
Cortex-A35	cortexa35
Cortex-A53	cortexa53
Cortex-A55	cortexa55
Cortex-A57	cortexa57
Cortex-A72	cortexa72
Cortex-A73	cortexa73
Cortex-A75	cortexa75
Cortex-A76	cortexa76
Cortex-A76AE	cortexa76ae
Cortex-A77	cortexa77
Cortex-A78	cortexa78
Cortex-R82	cortexr82
Cortex-X1	cortexx1
Kryo	kryo
Kryo250-Gold	kryo250g
Kryo250-Silver	kryo250s
Kryo260-Gold	kryo260g
Kryo260-Silver	kryo260s
Kryo280-Gold	kryo280g
Kryo280-Silver	kryo280s
Kryo360-Gold	kryo360g
Kryo360-Silver	kryo360s
Kryo385-Gold	kryo385g
Kryo385-Silver	kryo385s
Kryo460-Gold	kryo460g
Kryo460-Silver	kryo460s

Processor	Driver Option -cpu=
Kryo465-Gold	kryo465g
Kryo465-Silver	kryo465s
Kryo468-Gold	kryo468g
Kryo468-Silver	kryo468s
Kryo470-Gold	kryo470g
Kryo470-Silver	kryo470s
Kryo475-Gold	kryo475g
Kryo475-Silver	kryo475s
Kryo485-Gold	kryo485g
Kryo485-Silver	kryo485s
Kryo490-Gold	kryo490g
Kryo490-Silver	kryo490s
Kryo495-Gold	kryo495g
Kryo495-Silver	kryo495s
Kryo585-Gold	kryo585g
Kryo585-Silver	kryo585s

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 132 and “Debugging Options” on page 132.

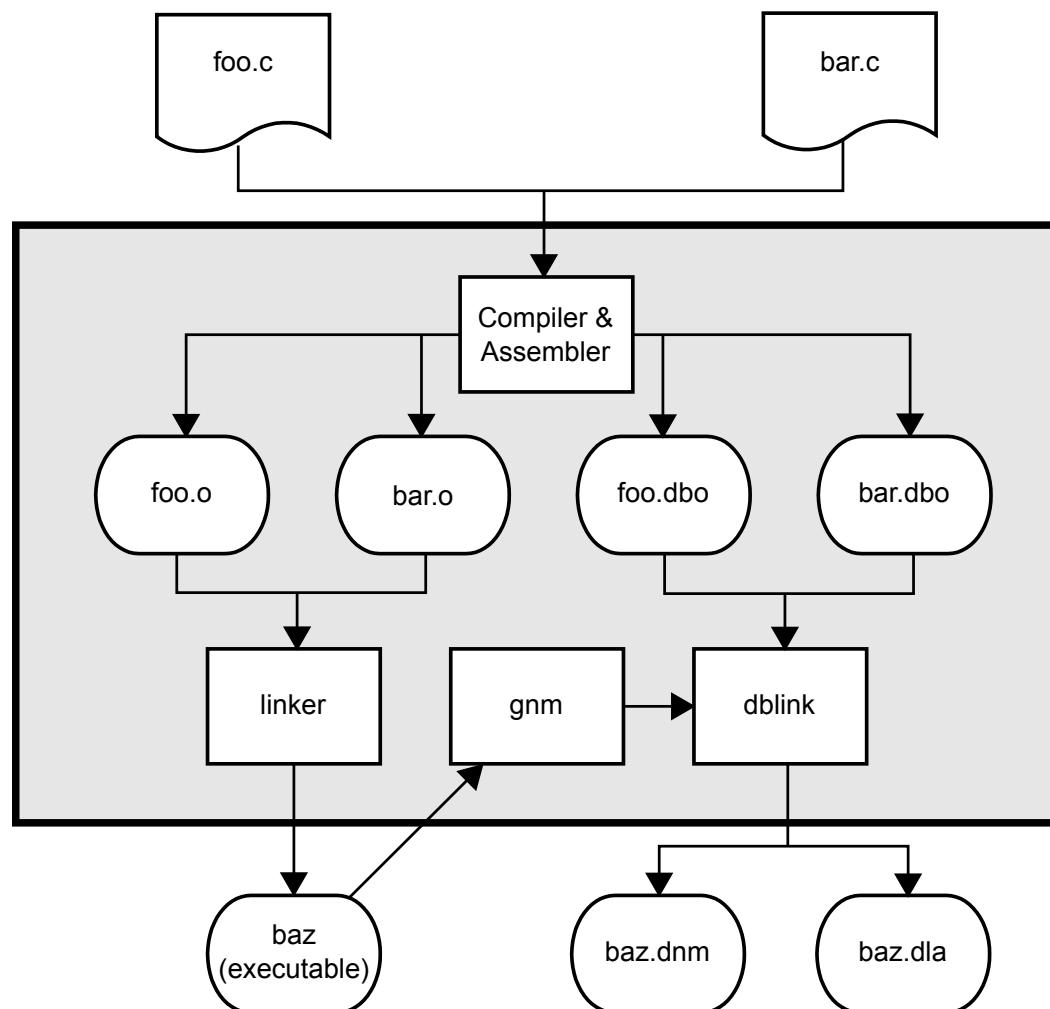
Debugging information is only created or updated when you create an executable or a library with the Builder or driver. It is not created when the linker or librarian are invoked separately. If you move an executable, library, or object file, you must also move the associated debugging information files. 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 the **dwarf2dbo** debug translator to 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 **dwarf2dbo** debug translator allows you to convert DWARF debugging information generated by third-party compilers into

the format required by the MULTI Debugger. If you need to use **dwarf2dbo**, we recommend that you integrate it with your build process.



Note

The **dwarf2dbo** debug translator is a separately licensed product of Green Hills Software. Contact your sales representative for information about licensing.

Translating DWARF Debugging Information Automatically

The **dwarf2dbo** Debug Translator converts DWARF debugging information (4.0 or prior) into the special debugging information required by the MULTI Debugger.

If you have licensed **dwarf2dbo**, you can configure MULTI to automatically convert DWARF debugging information. 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 Information Manually

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

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

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 (**-dwarf2dbo**) to the compiler driver during the final link phase. The general syntax for doing this is:

```
driver -dwarf2dbo link_options \  
    object_files -o executable_name
```

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 that you want to generate

a link map as well as the final executable, **a.out**. From the command line, run the Green Hills compiler driver:

```
driver -dwarf2dbo -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 42), 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** 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 Information Manually” on page 41) 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 **dwarf2dbo** can be used to improve the code browsing features of MULTI when using code compiled by third-party compilers. This feature causes the **dwarf2dbo** 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** 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 Information Manually” on page 41), 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 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 function calls cannot be made unless the executable has been linked against **libmulti.a**. For more information, see “Enabling Command Line Function Calls” on page 45.
- Static call graph browsing is not available unless the program is translated using the source scanning feature (see “Source Scanning” on page 42).
- 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 function 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 42), 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 42) 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.

Special Considerations for GNU Compilers

Green Hills Software recommends generating DWARF 4.0 information for sources compiled with GNU compilers. The standard **-g** GNU option may have been configured to generate any one of several debug information formats, so the full option names are used here for clarification. To generate DWARF 4.0 information from the GNU compiler, enter:

```
gcc -gdwarf-4
```

If your GNU compiler cannot generate DWARF 4.0 information using the **-gdwarf-4** option, DWARF version 2.0 is recommended. Generate DWARF 2.0 by entering:

```
gcc -gdwarf-2
```


Enabling Command Line Function Calls

You can call functions 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.

Function Entry/Exit History

Function entry/exit history is a separately licensed feature that tracks each entry and exit from a function. Control of function entry/exit history is available at compile time, link time, and run-time. At compile time, the compiler can generate instrumentation in the prologue and epilogue of each function (see “Generate Function Entry/Exit Instrumentation” on page 294). At link time, this instrumentation can be enabled, disabled, or stripped (see “Function Entry/Exit Logging” on page 133). At run-time, entry/exit history can be used by debuggers that support this feature, and toggled on or off by the **eagleconfig** MULTI command. For information about this command, see the MULTI: Debugging Command Reference.

This implementation allows you to compile a program with function entry/exit history enabled, then disable or strip the instrumentation in the linker without having to recompile. To use function entry/exit history on a function-by-function basis, use the `#pragma directive ghs entry_exit_history` (see “Green Hills Extension Pragma Directives” on page 752) or attribute `entry_exit_history` (see `entry_exit_history` on page 639).



Note

Only library functions that can be explicitly called by the user can be instrumented using function entry/exit history.

Using Function Entry/Exit History with INTEGRITY TimeMachine

- When INTEGRITY TimeMachine and Function Entry/Exit History are used in concert, some or all functions may derive entry and exit information from INTEGRITY TimeMachine instrumentation. Consequently, the times shown for entry/exit events in History will be less accurate than if INTEGRITY TimeMachine were disabled. To improve accuracy, consider passing `-lnk=-precise_fee_timestamps`. Note that using `-precise_fee_timestamps` will incur additional run-time performance overhead.
- Combining INTEGRITY TimeMachine and Function Entry/Exit History provides a high-level picture of a program's internal systems and activity. This perspective can allow for substantial performance improvements. However, once performance tuning becomes about smaller, more fine-grained changes, it may be advisable to disable INTEGRITY TimeMachine instrumentation to exclude its associated overhead. Consider also disabling Function Entry/Exit History for any frequently called functions. For information about disabling INTEGRITY TimeMachine, see “INTEGRITY TimeMachine Enabled” on page 297. For information about selectively disabling functions from Function Entry/Exit History, see the `#pragma` directive `ghs entry_exit_history` in “Green Hills Extension Pragma Directives” on page 752, and the attribute `entry_exit_history` on page 639.

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 567.

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 49.	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.	Medium

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. (A basic block is a set of instructions where if the first instruction executes, the rest will also execute.) There are three 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.
- *64-bit Count* profiling (**-coverage=count64**) is similar to count profiling, but uses 64-bit counters.

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 134. 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 six sections:

<code>.ghcovfa</code>	-coverage=flag block addresses
<code>.ghcovfz</code>	-coverage=flag block flags
<code>.ghcovca</code>	-coverage=count block addresses
<code>.ghcovcz</code>	-coverage=count block counts
<code>.ghcovda</code>	-coverage=count64 block addresses
<code>.ghcovdz</code>	-coverage=count64 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. If they are listed in the section map, they should not be specified with an explicit section contents field or section inclusion commands, nor should they be referenced in the contents of other sections. 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%
count64	30%	40%

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 752.

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 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 834 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 825. 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 834 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 135.

In order to use run-time error checking, you must be able to single step your target without triggering an interrupt.

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 an error, the program calls `exit` with a value greater than 0 (or `HaltTask`, if it is an INTEGRITY program), terminating itself. After 101 warnings are produced, it prints a message indicating too many errors and calls `exit` with a value greater than 0 (or `HaltTask`), 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*.



Note

If you are using run-time memory checking with an INTEGRITY project, shared libraries must be disabled. In the top-level project file, set the **Target→Operating System→Shared Libraries** build option to **No Shared Libraries (-non_shared)**. If you have disabled this option as described and still receive compile-time errors when building with run-time memory checking enabled, the option may be enabled in another project file. Make sure that no project file sets the **Target→Operating System→Shared Libraries** build option to **Link With Shared Libraries (-call_shared)**.

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 53). It generates run-time errors for the following cases:

- Memory that has not been allocated with `malloc()` is subsequently freed
- The same area of memory is freed twice
- Recently freed memory has been written to (may be detected when calls to the `malloc` library are made)

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 52). It generates run-time errors for the following cases:

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



Note

When debugging your program with MULTI 7, a write to recently freed memory may instead be reported as a read. This does not apply when running your program outside of the debugger.

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 52 or “Enabling Intensive Memory Allocation Checking” on page 53). 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 54.

For a demonstration of the use of these memory checking functions, see Example 2.2. Enabling and Disabling All Checks on page 55.

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 54.

**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 54.

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()`, `aligned_alloc()`, 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 2.2. 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 15, “Libraries and Header Files” on page 773.

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 (**-Idirectory**).

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 host	Windows host
A	<code>#include "/src/h/header.h"</code>	<code>#include "f:\src\h\header.h"</code>
B	<code>#include </src/h/header.h></code>	<code>#include <f:\src\h\header.h></code>
C	<code>#include "header.h"</code>	<code>#include "header.h"</code>
D	<code>#include <header.h></code>	<code>#include <header.h></code>

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 (cases C and D above), the compiler searches in the following directories, in order:
 1. Only if quotes are used (case C above), the directory of the source file that contains the `#include` directive.
 2. Directories specified with the **Project→Include Directories** option (**-Idirectory**), in the order in which they appear on the command line.
 3. The default directories (see “C and C++ Header File Directories” on page 775).
 4. Directories specified with the **Project→Advanced→Preprocessor Options→Advanced Include Directories→System Include Directories** option (**-sys_include_directory**), in the order in which they appear on the command line.

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 `-Idirectory`, 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 (**-Llibrary_directory**).
- Specify the library names to link against in the **Project→Libraries** option (**-llibrary**). 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 automatically invokes the assembler, **asarm64**, to process assembly files generated by the compiler. 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[=filename]**).

For a list of all the assembler-specific Builder and driver options, see “Assembler Options” on page 224. For detailed documentation of the assembler, see Chapter 6, “The asarm64 Assembler” on page 389.

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 227. For detailed documentation of the linker, see Chapter 8, “The elxr Linker” on page 423.

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 429.

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 69), 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/arm64* directory with a board-specific memory map taken from *install_dir/target/arm64/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), 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 91). 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.



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 434.

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



Note

To change the compiler's default section mappings, see “Rename Section” on page 97.

Example 2.3. 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 413.

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 451.

Aligned Data Sections

By default the compiler assigns all data to one of the default data sections according to its access type, see “Default Program Sections” on page 61. When data with different alignment requirements are assigned to the same section, gaps between data may be present in the section. To avoid such gaps, default data sections can be split by alignment. Each aligned data section is named `sectname.aN`, where `sectname` is the name of the original data section, and `N` is the alignment in bytes. To enable aligned data sections:



Set the **Target→Text and Data Placement→Aligned Data Sections** option to **On** (`-split_data_sections_by_alignment`).

For example, the variable `char x = 5;`, normally be assigned to `.data`, would now be assigned to `.data.a1`, assuming the `char` type is 1 byte aligned. All other data which is 1 byte aligned and would normally be assigned to `.data` would also be assigned to the `.data.a1` section.

The aligned sections feature only takes effect for variables that belong to the default non-COMMON data sections, such as `.data`, `.bss` and `.rodata`. For a complete list of default sections, see “Default Program Sections” on page 61. Variables that are in a user-named section by pragma `#pragma ghs section, attribute __attribute__((section("name")))`, or by driver option **-section** are not assigned to an aligned data section, and are kept it in the user-named section.



Note

The aligned data section feature does not take effect for uninitialized global variables in C that are COMMON. To assign uninitialized global variables in C to aligned sections, use **--no_commons** (see “Allocation of Uninitialized Global Variables” on page 91) to disable commons if possible.

At link time, the linker coalesces sections `sectname.aN` into section `sectname` in the executable output file automatically, unless you explicitly list the aligned section in the linker directives file, in which case the aligned section will be placed in the output file according to the section map specified.

If an object file containing an aligned section is linked to a relocatable object file instead of to an executable program (with the **-relobj** option), the linker keeps the aligned section in the relocatable object file and does not merge it into the section which the aligned section derived from. When more than one object file contains the same aligned section, they are merged into the same aligned section in the relocatable object file.

**Note**

The `SORT` linker directive can be used to sort `sectname.aN` sections across different modules. For more information, see “Using Wildcards in Section Inclusion Commands” on page 438.

Individual Function and Variable Sections

By default, the compiler assigns all functions to a single text section: `.text`. All variables are assigned to data sections according to access type: initialized, uninitialized, or read-only. The individual section feature allows you to change these default assignments to locate functions and non-COMMON variables in their own unique sections. When enabled, section names are formed by appending the symbol name of the function or variable to the name of the original section (`sectname.symname`). For example, the individual section for a function `foo` in `.text` will be named `.text.foo`. Likewise, the individual section for an initialized global variable `x` in section `.data` will be named `.data.x`. At link time, the linker will coalesce individual sections into sections named `sectname`. You can also explicitly specify a section map by listing individual sections in a linker directives file. Doing so gives you the option of assigning particular functions or variables to arbitrary addresses.

There are a number of important considerations when using this feature:

- The placement of a function or variable in the final executable depends on the order in which object files are processed and the order of the sections in an object file's symbol table. As such, enabling individual sections may cause the addresses assigned to functions and variables to be different than if individual sections were disabled. To avoid this possibility, manually assign specific addresses in a linker directives file.
- If an object file containing an individual section is linked to a relocatable object file instead of an executable program (**`-relobj`**), the linker will keep the individual section in the relocatable object file. If more than one object file contains the same individual section, they will be merged into a unified section in the relocatable object file.
- The symbol name component of an individual section name (`.symname`) matches the name in the symbol table of the compiled object file. Normally, it is the name of the function or variable. But this is not always the case. For example, for a function in C++, the symbol name is a compiler-generated

mangled name. This is also true for a local static variable with function scope. As such, if you need to explicitly reference an individual section's name, check the symbol table of the compiled object file to confirm the name's correctness. You can look up the symbol table by running **gnm** on the file in question.

- By default, the individual section feature only takes effect for functions or variables that belong to the default text or non-COMMON data sections, such as `.text`, `.data` and `.bss`. However, options are available to enable the feature for functions or variables that belong to user-defined sections. For a complete list of default sections, see “Default Program Sections” on page 61.

For a complete list of the driver options related to the individual section feature, see:

- “Individual Variable Sections for Default Data Sections” on page 98
- “Individual Variable Sections for Renamed Data Sections” on page 98
- “Individual Variable Sections for User-Defined Data Sections” on page 99
- “Individual Function Sections for Default Text Section” on page 100
- “Individual Function Sections for Renamed Text Sections” on page 100
- “Individual Function Sections for User-Defined Text Sections” on page 101



Note

The individual section feature does not take effect for uninitialized global variables in C that are COMMON. To assign uninitialized global variables in C to individual sections, use **--no_commons** (see “Allocation of Uninitialized Global Variables” on page 91).



Note

By default one single period (".") is used to separate base section names and function or variable symbol names in individual section names. One can use the driver option **-individual_section_name_extra_dot** to have an extra single period in individual section names. For more information about the option, see “Extra Dot in Individual Section Name” on page 102.

Example 2.4. Individual Sections for Local Static Variables

In the following sample C source file `test.c`, the function `main` has five local static variables.

```
volatile unsigned int dummy1_u32, dummy2_u32;

int main(void)
{
    /* differently named */
    static unsigned int static_x, static_y, static_z;

    /* same name as below, but different scope */
    static unsigned int static_u32;
    static_u32 = static_u32 + dummy1_u32;
    dummy1_u32 = static_u32;
    {
        /* same name as above / below, but different scope */
        static unsigned int static_u32;
        static_u32 = static_u32 + dummy2_u32;
        dummy2_u32 = static_u32;
    }
    {
        /* same name as above, but different scope */
        static unsigned int static_u32;
        static_u32 = static_u32 + dummy2_u32;
        dummy2_u32 = static_u32;
    }

    static_x = static_x + 1;
    static_y = static_y + 1;
    static_z = static_z + 1;

    return 0;
}
```

When compiled with **-individual_data_sections**, the five local static variables are assigned to five individual sections. The variables `static_x`, `static_y` and `static_z` use unique names in the function, and so their mangled symbol names do not have a number. In contrast, three variables share the name `static_u32`, and thus include a number for differentiation.

```
ccarm64 -individual_data_sections --no_commons test.c -c
nm test.o | grep OBJ
0|      0|OBJT |LOCL |0   |.text                |..gh.f.a.1.e.0
0|      0|OBJT |LOCL |0   |.bss                  |.L54
0|      4|OBJT |LOCL |0   |.bss.static_x.main    |static_x.main
0|      4|OBJT |LOCL |0   |.bss.static_y.main    |static_y.main
0|      4|OBJT |LOCL |0   |.bss.static_z.main    |static_z.main
0|      4|OBJT |LOCL |0   |.bss.static_u32.1.main|static_u32.1.main
```



```
0|      4|OBJT |LOCL |0      |.bss.static_u32.2.main|static_u32.2.main
0|      4|OBJT |LOCL |0      |.bss.static_u32.3.main|static_u32.3.main
0|      4|OBJT |GLOB |0      |.bss.dummy1_u32      |dummy1_u32
0|      4|OBJT |GLOB |0      |.bss.dummy2_u32      |dummy2_u32
```

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** — 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**. If compiling from the command line and using a **-bsp** option, the equivalent driver option is **-layout=ram**.
- **Link to and execute out of ROM** — 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 function calls, memory leak detection, and run-time error checking. It appears in the Target

Resources project as **standalone_romrun.ld**. If compiling from the command line and using a **-bsp** option, the equivalent driver option is **-layout=romrun**. Note that not all BSPs offer this **.ld** file or support this layout. 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** — 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**. If compiling from the command line and using a **-bsp** option, the equivalent driver option is **-layout=romcopy**. Note that not all BSPs offer this **.ld** file or support this layout.

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 429. For further discussion about optimizing your program by editing your linker directives file, see “Modifying your Section Map for Speed or Size” on page 450.

If you are using the driver with the **-layout** option, you must also specify the **-bsp=** option (see “Specifying an ARM64 Target” on page 35). If no appropriate BSP is available, specify **-bsp=generic**. The **-layout=** option is ignored if you specify your own linker directives files.

The **-layout** driver option is not supported with INTEGRITY. If you have an INTEGRITY project, set the program layout in the Project Manager or specify your linker directives files manually.

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). Depending on the type of program being built, you can reserve up to 8 global registers (r18 to r25 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.



Note

If you are building INTEGRITY code, you cannot modify the default value of this option. If you are building C++ code, this option cannot be set higher than 3.

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

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

The usage of the `register` keyword is otherwise illegal.



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.

The Green Hills stand-alone libraries are compiled with only the first global register reserved. The remaining global registers are permanent registers, so functions compiled with a higher **-globalreg** value can call (but not be called by) those library functions. A function may not access variables allocated to global registers unless

all functions in the call chain before it were compiled with the global registers reserved. As such, it is not safe to use variables allocated to global registers in library callback routines (such as the user-supplied comparison function called by the **stdlib.h** functions `bsearch()` and `qsort()`).

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.



Note

When `longjmp()` is called, the second and subsequent global registers are reset to their values as of the matching `setjmp()` call. When an exception is thrown, they are reset to their values as of the start of the catching `try` block. The first global register is not reset on `longjmp()` or `throw`.

Near and Far Function Calls

A function call is normally performed with an instruction that contains a 26-bit value that is shifted left by 2 and 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 28 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

In most cases, the linker is able to detect out of range calls, and inserts veneers to the far function as necessary. This method is the default behavior of the linker and does not require the setting of options or modification of source code. A far call using this method requires two branches, one to the veneer and one to the function itself, which may reduce execution speed for the call. Nonetheless, this is generally the recommended method for making far function calls on the ARM architecture.

**Note**

Functions declared with the `static` keyword cannot be patched in this way, but far calls will be generated for them if you use a `#pragma`.

Function Declaration

The keywords `__nearcall` and `__farcall` 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 the command line option. For example:

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

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

The `__nearcall` and `__farcall` 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.

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 825). 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 451).

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**).

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

```
ccarm64 -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.

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 `.bss` 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

Target Options	79
Project Options	113
Optimization Options	119
Debugging Options	132
Preprocessor Options	140
C/C++ Compiler Options	142
Assembler Options	224
Linker Options	227
Compiler Diagnostics Options	251
DoubleCheck (C/C++) Options	262
Advanced Options	265

This chapter lists all Builder and compiler driver options. These options control numerous aspects of the compilation, assembly, linking, and management of projects. Builder and driver options differ in the following ways:

- Driver options control the behavior of the compiler driver.
- Builder options control the behavior of **gbuild** and the project manager.

The following information is listed for each option:

- The name of the option as it appears in the MULTI Project Manager's **Build Options** window.
- A description of the option.
- Any alternate forms of the option (such as an "on" and "off" version).

Unless stated otherwise in an option's description, it is a driver option. For information about setting options through the Project Manager, see the documentation about setting Builder options in the *MULTI: Managing Projects and Configuring the IDE* book.

Target Options

This is a top-level option category.

Target options control aspects of compilation specific to your target, such as available memory models and forms of floating-point support.

For additional, more specialized options, see “Target Options” on page 265.

Endianness

Controls the endianness of generated code (the byte ordering of memory addresses). Endianness fundamentally impacts how your processor reads memory, and should rarely need to be manually changed. The settings below are automatically selected by the MULTI New Project Wizard based on your board. For details about your processor's endianness, see “ARM64 Characteristics” on page 26. The permitted settings for this option are:

- **Big Endian (-bigendian)** — Causes the most significant byte of an integer to appear at the lowest address. This setting will result in big endian byte ordering for data and little endian byte ordering for code.
- **Little Endian (-littleendian)** — [default] Causes the most significant byte of an integer to appear at the highest address.



Note

To function correctly, this option's setting must be uniformly applied to all parts of a program.

Memory Functions

Controls the use of specific memory functions. Permitted settings for this option are:

- **Fast (-mem_funcs=fast)** — Specifies the use of memory functions designed for overall performance.
- **Safe (-mem_funcs=safe)** — Redirects `memcpy()` and `memset()` to alternative implementations with additional constraints. These alternative implementations are designed to trade-off performance to constrain their operation for restricted

environments (such as when certain requirements must be met for memory functions to interact with memory-mapped I/O). The alternative implementations of `memcpy()` and `memset()` have the following constraints:

- These versions will only use general-purpose registers (no floating-point or SIMD registers).
- All memory accesses will be sufficiently aligned to not cause an alignment fault.
- Assuming non-overlapping arrays, a function will access each source and destination byte exactly once.
- Accesses to the source and destination array will be in ascending order by address.
- Functions will not access memory outside of the source and destination arrays.

The **-mem_funcs=safe** setting works by redirecting function calls in the linker. For more information, see “Standard Function Names Converted by the Linker” on page 823.

The aliases `__ghs_fast_memcpy` and `__ghs_fast_memset` can be called to explicitly use the **Fast** versions of `memcpy()` and `memset()`. The aliases `__ghs_safe_memcpy` and `__ghs_safe_memset` can be called to explicitly use the **Safe** versions of `memcpy()` and `memset()`. If you intend to call either `memcpy()` variant using the explicit aliases, you should not provide your own `memcpy()` implementation. Doing so may lead to incorrect behavior in `memmove()`, which depends on certain properties of `memcpy()`.

When upgrading your Green Hills compiler, it is always advisable to rebuild the INTEGRITY shared libraries. If you use **-mem_funcs=safe**, however, it is especially important to rebuild the INTEGRITY libraries. Doing otherwise may cause those memory functions to be unavailable, in addition to other incorrect behavior.

The default setting is **-mem_funcs=fast**. If **-kernel**, **-record_timemachine**, or **-no_misalign_access** is used, the default is **-mem_funcs=safe**.

This option is only supported when using the Green Hills C library.

Enable Toolchain Versions of Certain Memory Functions

Controls whether the toolchain's memory functions are considered when linking a program. When enabled, the toolchain's memory functions may be preferred over those provided by the operating system. Otherwise, the operating system's memory functions will be used. Depending on INTEGRITY version, **-toolchain_mem_funcs** may have no effect on what memory functions are linked in. Permitted settings for this option are:

- **On (-toolchain_mem_funcs)** — This setting is the default when **-act_like** is set to 2019.1 or later.
- **Off (-no_toolchain_mem_funcs)**



Note

This option is only supported when building an INTEGRITY Virtual AddressSpace.

Annotate Call/Return Instructions

Controls whether to annotate call/return instructions for improved debugging. This involves adding the non-loading sections `.ghcalltbl`, `.ghtailcalltbl`, and `.ghrettbl`. These sections contain address lists of call, tail-call, and return instructions. Permitted settings for this option are:

- **On (-annotate_call_return)** — [default]
- **Off (-no_annotate_call_return)**

Memory Model

Controls whether to use a 33-bit or 64-bit memory model for variable and function names. By default, the compiler uses 33-bit relative relocations, the "small" memory model. This setting increases performance for suitably compact programs. If your program's variable and function locations are not within 33-bits of one another, use the large memory model instead. Permitted settings for this option are:

- **Small Memory Model (-small_memory_model)** — [default] Specifies the use of signed, 33-bit relative relocations when loading the addresses of code and data.

- **Large Memory Model (-large_memory_model)** — Specifies the use of 64-bit absolute relocations when loading addresses.

Workaround ARM64 Erratum 1183550 (DMB)

Mitigates ARM Cortex-A53 Erratum ID 1183550. May insert a `DMB` instruction to work around a potential hardware problem. Please contact ARM for further details on the erratum. Permitted settings for this option are:

- **Off (-no_arm64_errata_1183550)** — [default]
- **On (-arm64_errata_1183550)**

Workaround ARM64 Erratum 1183550 (NOP)

Mitigates ARM Cortex-A53 Erratum ID 1183550. May insert `NOP` instructions to work around a potential hardware problem. Please contact ARM for further details on the erratum. Permitted settings for this option are:

- **Off (-no_arm64_errata_1183550_nop)** — [default]
- **On (-arm64_errata_1183550_nop)**

Straight-line Spectre Compiler Mitigation

Controls whether toolchain will generate code with mitigation instructions inserted for straight-line Spectre vulnerabilities (CVE-2020-13844). Permitted settings for this option are:

- **Off (-no_straightline_spectre_mitigation)** — [default]
- **On (-straightline_spectre_mitigation)**



Note

This option does not affect assembly code. To generate mitigation instructions for assembly code, see “Straight-line Spectre Assembler Mitigation” on page 83.

**Note**

The setting **-straightline_spectre_mitigation** is incompatible with **-gen_timemachine** and **-arm64_aapcs=v1**.

Straight-line Spectre Linker Mitigation Check

Controls whether the linker will give a warning if instructions vulnerable to straight-line Spectre vulnerabilities (CVE-2020-13844) appear without mitigation code. This is useful to verify that all modules have the expected mitigation code in the final link. Permitted settings for this option are:

- **Off (-no_check_straightline_spectre_mitigation)** — [default]
- **On (-check_straightline_spectre_mitigation)**

Straight-line Spectre Assembler Mitigation

Controls whether the toolchain will insert mitigation instructions for straight-line Spectre vulnerabilities (CVE-2020-13844) in assembly files. This is a separate mitigation option as the toolchain cannot know when it is safe to make such insertions. For example, hand-coded assembly may make assumptions about the size of functions, instruction regions for hard-coded branch offsets, or interrupt vector sizes. If assumptions such as these are not being made, this option should be safe to enable. Permitted settings for this option are:

- **Off (-no_straightline_spectre_assembler_mitigation)** — [default]
- **On (-straightline_spectre_assembler_mitigation)**

Floating-Point

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

Floating-Point Mode

Indicates whether floating-point operations should be calculated using software, hardware, or a combination thereof. This option also allows you to disable floating-point operations entirely. In general, the appropriate floating-point setting

will be automatically set by the compiler based on your hardware, and will not need to be changed. This option exists for those times when the automatic setting is inappropriate or non-optimal for a specific task. Permitted settings for this option are:

- **No Floating-Point (-fnone)** — Disallows all floating-point operations. In addition to its use with processors lacking an FPU, this setting may also be appropriate if you are attempting to minimize program size. To understand why, consider the two primary effects of **-fnone** on program compilation. First, the C and C++ compilers will give an error for any use of floating-point variables or constants. This will greatly reduce the size of such library functions as `printf` and `scanf`. Second, the compiler will pass `-D__NoFloat__` to allow source code and header files to hide references to floating-point objects. For more information, see “Predefined Macro Names” on page 734.

Note that code related to floating-point operations may be linked in to your executable even if it does not use floating-point variables. The exclusion of this code can cause an additional non-trivial reduction in program size. Finally, because this setting requires support from system libraries and header files, it is not supported for environments using embedded operating systems with their own C or C++ headers and libraries. Other targets may use target-specific header files that require the use of floating-point types.

This setting implies **-no_float_scanf**.

- **Hardware Floating Point (-fhard)** — [default] Specifies hardware floating point (HFP) mode. This setting causes your target to store floating-point data in floating-point registers and use floating-point instructions to do floating-point operations. This is in contrast to a software-simulated floating-point mode (see **-fsoft** below).
- **Software Emulation (-fsoft)** — Specifies software floating-point (SFP) mode. This setting causes your target to use integer registers to hold floating-point data and use library subroutine calls to emulate floating-point operations.



Note

To function correctly, this option's setting must be uniformly applied to all parts of a program.

Floating-Point Rigor

Indicates a set of accuracy/performance tradeoffs in calculating and comparing floating-point values. This is an omnibus option, and thus specifies a number of individual floating-point options. Related options (such as **-precise_signed_zero**) can still be enabled or disabled for fine-grained control. Permitted settings for this option are:

- **Default (-frigor=default)** — Specifies a balanced approach to floating-point accuracy and performance.
- **Accurate (-frigor=accurate)** — Specifies favoring accuracy over floating-point performance.
- **Fast (-frigor=fast)** — Specifies favoring performance over floating-point accuracy.

Note that the **-frigor** option offers no more utility than the individual options that comprise it. For example, one effect of **-frigor=accurate** is that NaNs will compare as unordered, because **-frigor=accurate** implies **-fNaN_cmp_unordered**. But **-fNaN_cmp_unordered** requires the use of hardware floating point. As such, passing **-frigor=accurate** while using software floating point will cause no such change in comparison behavior.

The following list shows the options that are set by the three **-frigor** settings:

- **-frigor=default** is equivalent to passing no **-frigor** option on the command line.
- **-frigor=fast** implies the following options (if these options are not otherwise set on the command line):
 - **-float_associativity**
 - **-ffunctions**
 - **-fused_madd**
 - **-no_precise_signed_zero**
 - **-Oinline_constant_math**
 - **-fno_NaN_cmp_unordered**

- **-frigor=accurate** implies the following options (if these options are not otherwise set on the command line):
 - **-no_float_associativity**
 - **-fnofunctions**
 - **-no_fused_madd**
 - **-precise_signed_zero**
 - **-Ono_inline_constant_math**
 - **-fNaN_cmp_unordered**

NaN Compares as Unordered

Controls whether floating-point comparisons using the C relational operators properly handle NaN operands as unordered. Properly handling such comparisons may require generating additional or alternative instructions, and these instructions can degrade floating-point comparison performance. This option requires the use of hardware floating point (**-fhard**). Permitted settings for this option are:

- **On (-fNaN_cmp_unordered)**
- **Off (-fno_NaN_cmp_unordered)** — [default]

Floating Point Associativity

Controls whether to apply the mathematical associativity property to floating-point expressions. When enabled (the default state), the compiler may apply the associativity property to floating-point expressions. This may cause floating-point values to be rounded in a non-standard manner. Disabling this option may lead to more accurate floating-point calculations at the cost of reduced performance. Permitted settings for this option are:

- **On (-float_associativity)** — [default]
- **Off (-no_float_associativity)**

Floating-Point Functions

Controls whether calls to math library functions (such as `sqrt()`) can be replaced with inline instructions. As with other forms of inlining, this can improve performance at the potential cost of increasing code size. Note that substituted instructions may not be exactly equivalent to the function calls they are replacing. For example, they may not exhibit identical floating-point rounding or set `errno`. Even if this option is disabled, other optimizations that replace calls to math library functions may be performed under **-Oinline_constant_math**. Permitted settings for this option are:

- **On (-ffunctions)**
- **Off (-fnofunctions)** — [default]

Floating-Point Fused Multiply and Add

Controls whether the compiler may use fused multiply and add instructions. These are multiply instructions where a floating-point multiply is combined with a subsequent addition or subtraction into a single operation. While this is generally faster, the resulting operation may have different rounding characteristics than the original sequence of operations. Note also that this option is only functional on hardware that explicitly supports it.

Permitted settings for this option are:

- **On (-fused_madd)** — [default]
- **Off (-no_fused_madd)**

Implicit Floating-Point/SIMD Register Use

Controls whether the compiler can use FPU/SIMD registers for general purpose code generation on targets that support hardware floating point. If disabled, these registers will only be used for functions that have explicit references to floating point or SIMD data types or are SIMD vectorized. For more information, see “SIMD Vectorization” on page 346. Permitted settings for this option are:

- **Off (-no_implicit_float_reg)**
- **On (-implicit_float_reg)** — [default]

**Note**

This option has no effect when compiling for INTEGRITY.

Floating-Point Precise Signed Zero

Controls whether the compiler can substitute a sequence of operations that could produce a negative zero (-0.0) outcome with one that could produce a positive zero outcome (0.0), and vice versa. Allowing this substitution does not strictly conform to the IEEE floating-point standards, but can improve performance and reduce code size. Additionally, enabling this option should have no impact on the accuracy of your code when the distinction between signed zero quantities is irrelevant (the compiler treats such quantities as equal). Permitted settings for this option are:

- **On** (`-precise_signed_zero`)
- **Off** (`-no_precise_signed_zero`) — [default]

Half-Precision Floating-Point Support

Permitted settings for this option are:

- **On** (`-half_precision_type`) — Enable compiler support for `__fp16`, a half-precision floating-point type.
- **Off** (`-no_half_precision_type`) — [default] Disable compiler support for `__fp16`

The `__fp16` data type is a 16-bit (half-precision) IEEE 754-2008 floating-point type, which consists of a 5-bit exponent and an 11-bit signed fraction. The `__fp16` data type is capable of representing NaN and INFINITY, as well as finite and subnormal numbers. The `__fp16` type is intended for reducing memory usage in cases where the reduced range and precision are acceptable. Operations on `__fp16` types are automatically promoted to `float`, so it is generally preferable to use `float` variables for intermediate results to avoid unnecessary conversions. For compatibility, `__float16` is accepted as a synonym for `__fp16`. No syntax is provided for representing floating-point literals in the `__fp16` type. Instead single or double precision floating-point literals should be used.

The `__fp16` type can be used in the same kinds of places as `float`, including typedefs, structures, arrays, function arguments, and return types. On targets that support it, conversion between `__fp16` and other types will be performed using hardware floating-point instructions. Otherwise, the conversion will be performed via library routines.

For example:

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

/* Each element of this array consumes 2 bytes of
 * memory, for a total of 10 bytes. */
__fp16 array[] = {1.1, -200.3, 4000.5, -60000.7, 54321.9};

void func1(__fp16 *p, float scale)
{
    int i;
    for (i=0; i<5; i++){
        /* The following operation will be performed
         * as float, requiring a conversion from __fp16
         * to float, a single-precision multiply, and *
         * then a conversion from float to __fp16.
         *
         * Values that are out of range of the __fp16
         * type will be flushed to +/- inf when converted
         * from float.
         */
        p[i] *= scale;
    }
}

int main()
{
    int i;
    func1(array, 1.5);

    for (i=0; i<5; i++){
        /* __fp16 variables are automatically converted to
         * double when passed to variadic functions (just
         * like float are).
         *
         * Note that after scaling in func1(), two of the
```

```
        * elements in the array are out of range and will
        * return true for __isinfh() and will print as
        * +/- inf.
        */
    printf("array[%d] = %f  (isinf=%d, signbit=%d)\n",
           i, array[i], __isinfh(array[i]),
           __signbith(array[i]));
}

return 0;
}
```

For convenience, the following half-precision floating-point functions/macros are defined in `math.h`:

- `int __isnormalh(__fp16);`
- `int __isfiniteh(__fp16);`
- `int __isnanh(__fp16);`
- `int __isinfh(__fp16);`
- `int __signbith(__fp16);`
- `int __fpclassifyh(__fp16);`



Note

The generic macros in `math.h`, such as `isnormal`, do not directly support the half-precision type. They may work in some cases, since the half-precision value will be converted to float, but it is possible to get different results than by directly calling the half-precision versions. A subnormal half-precision type may be normalized or flushed to zero during conversion to float, for example, leading to a different result from `isnormal` than from `__isnormalh`. It is recommended that the specific half-precision versions be used for half-precision values.

Text and Data Placement

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

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

Allocation of Uninitialized Global Variables

Controls the allocation of uninitialized global variables. Permitted settings for this option are:

- **Treat as Common Data (`--commons`)** — 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.



Note

Common data is not supported in C11 or later C language modes. This applies both to this option setting and to the `common` attribute (see “Attributes” on page 637).

- **Treat as Unique Definitions (`--no_commons`)** — [default] 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 */
```

Linker-Based Far Call Patching

Controls a linker-based optimization that attempts to generate far calls for out-of-range calls that would otherwise fail. If a single module contains code in a

single section that is itself larger than the allowable function call range, then **-farcallpatch** may be insufficient to resolve far calls from that module and will result in a link time error. 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 global variables and arrays that are explicitly initialized to zero. Allocating such objects to zero-initialized sections will generally reduce the size of an executable ROM image. Permitted settings for this option are:

- **Place Zero-Initialized Data in BSS Sections (-discard_zero_initializers)** — [default] Causes explicitly zero-initialized global variables to be allocated as if they were defined but not initialized. For example, if the statement `int x = 0;` normally places `x` in the `.data` section, the compiler may place it in the `.bss` section instead. In either case, `x` is still initialized to zero.
- **Place Zero-Initialized Data in Data Sections (-no_discard_zero_initializers)** — Causes explicitly zero-initialized global variables to be allocated in the same way as variables 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.



Note

When a global variable explicitly initialized to zero is placed in a user-named section (using `__attribute__((section("name")))`), the section will still be treated as `.data`-like. This will remain true even if **-discard_zero_initializers** is set. The initializer will need to be removed for the user-named section to be treated as a `.bss`-like section.

Global Registers

Reserves up to 8 registers (`r18` to `r25` in that order) to hold global variables. For more information, see “Storing Global Variables in Registers” on page 70.

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
```



Note

If you are building C++ code, this option cannot be set higher than 3. If you attempt to do so, the compiler will issue the following message:

```
Error: -globalreg=n invalid. Cannot set higher than 3
for a C++ program.
```



Note

To function correctly, this option's setting must be uniformly applied to all parts of a program.

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 825). 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 645).

Check for Stack Smashing Attacks

Controls code instrumentation that provides protection against stack-smashing attacks. Stack-smashing attacks are attempts to override function return values. When this option is enabled, instrumentation will be added to functions that call `alloca()` or that have a local aggregate or array variable greater than 8 bytes. In the prologue of each instrumented function, the compiler will place a canary value adjacent to a set of saved registers. In the epilogue, this value will be compared against the original value. If the values do not match, the program will be terminated with an error function. For implementation details, see the code example offered below.

Although this option offers protection against stack-smashing attacks, a motivated attacker can overcome the protection using a multi-phased attack. The best protection is thus to write a program that contains no possibility of an array overrun. More generally, note that the **-stack_protector** setting is not a substitute for well-designed software. Permitted settings for this option are:

- **On (-stack_protector)**
- **Off (-no_stack_protector) — [default]**

The following pseudo code illustrates canary value randomization and customized error handling.

```
#pragma ghs startnoinline
// Do not allow main2() to be inlined or main() may require a canary.
static int main2(void){
    // Code that would normally be in main() should be placed here.
}
#pragma ghs endnoinline

void __stack_chk_fail(void){
    // Log the failure
    Exit(1);
}
extern uintptr_t *__ghs_get_stack_chk_guard_address(void);

int main() {
    // Initialize the stack canary before any code
    // that may require a stack canary
    *__ghs_get_stack_chk_guard_address() = <random_value>;
    // The real program starts in main2
    return main2();
}
```

On stand-alone targets, the canary value can alternatively be initialized by overriding the function `__ghs_set_stack_chk_guard()`:

```
void __ghs_set_stack_chk_guard(void)
{
    *__ghs_get_stack_chk_guard_address() = <random_value>;
}
```

This avoids the need for `main2()`. For more information, see `src/libsys/ind_stackcanary2.c` and `src/libsys/ind_stackcanary.c`.

Indirect Function Call Checks

Controls whether to instruct the toolchain to mitigate against forward-edge attacks. Forward-edge attacks attempt to subvert the behavior of a program by corrupting and then manipulating the values of function pointers. Attackers may use corrupted function pointers to point to any executable component of a program, including the middle of a function.

Enabling this option will cause the toolchain to create one or more tables populated with descriptors of each function that has addresses taken in the program. Function pointers within the program will become pointers to these descriptors. When the program would normally perform an indirect call using a function pointer, a call will be made instead to a routine where the function descriptor is passed as an extra argument. This function, `__ghs_ifc_check()`, will check that the descriptor is a valid descriptor within the table, dispatching the code to the indicated destination if valid, and terminating if invalid. This ensures that forward-edge attacks can only branch to actual function locations.

Note the following caveats:

- Code that assumes that a function pointer points directly to a function's memory address may not work properly with this option. When aliases or weak symbols are used to reference the same function using different symbol names, the function pointer addresses may differ (multiple function pointers can point to multiple descriptors) but the descriptors will point back to the same target function.
- Code built with this option must only call through function pointers also built with this option. (Code built without this option, however, can call function pointers built with it.) Be aware that when linking with this option, any library

code will also be built with it. Any function pointers used by library code, including C++ dtors and ctors, must also be built with this option.

- This option is not currently supported with INTEGRITY TimeMachine or with assembly code that attempts to create function pointers.

Permitted settings for this option are:

- **On** (**-indirect_function_call_checks**)
- **Off** (**-no_indirect_function_call_checks**) — [default]

Instrument Prologue/Epilogue

Controls whether the compiler will insert calls to `__ghs_prologue` into the prologue of each function, and calls to `__ghs_epilogue` into the epilogue of each function. By providing a definition for these subroutines, you can perform tasks such as logging functions or stack buffer overflow detection. If you do not provide a definition for these subroutines, a non-functional default definition will be provided. Calls to the default routines can be optimized away with **-inline_trivial**.

Permitted settings for this option are:

- **On** (**-instrument_prologue_epilogue**)
- **Off** (**-no_instrument_prologue_epilogue**) — [default]

The subroutines used by this option use a different, more restrictive version of the default ARM64 EABI calling convention. As such, assembly code should be used to write definitions. The following are additions to the default EABI:

- The register `x16` contains the return address of the caller when entering the subroutine.
- The registers `x14`, `x15`, and `x17` may be modified. All other registers must be preserved, including `x16`.

To use instrument prologue/epilogue on a function-by-function basis, use the `#pragma directive` `ghs_instrument_prologue_epilogue` (see “Green Hills Extension Pragma Directives” on page 752) or attribute `instrument_prologue_epilogue` (see `instrument_prologue_epilogue` on page 643).

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 62). 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 61 (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`.

Aligned Data Sections

Controls whether the compiler will assign a variable in a default data section to an aligned data section whose name is *sectname.aN*, where *sectname* is the name of the default data section, and *N* is the alignment in bytes of the variable. Permitted settings for this option are:

- **Split Data Sections by Alignment (-split_data_sections_by_alignment)** — Allocates a variable in a default data section to an aligned data section according to its alignment. However, if the variable is in a user-named section either by `#pragma` or by `__attribute__`, do not allocate it to an aligned data section and keep it in the user-named section.
- **Do Not Split Data Sections by Alignment (-no_split_data_sections_by_alignment)** — [default] Do not allocate variables to aligned data sections.

This option should be passed both when compiling and when linking. For more information, see “Aligned Data Sections” on page 64.

Individual Variable Sections for Default Data Sections

Controls whether to enable the Individual Variable Sections support and whether the compiler will assign a non-COMMON global or static variable in a default data section to its own individual section, whose name is *sectname.symname*, where *sectname* is the default data section's name, and *symname* is the variable's symbol name. Permitted settings for this option are:

- **Allocate Variables in Default Data Sections to Individual Variable Sections (-individual_data_sections)**
- **Do Not Allocate Variables in Default Data Sections to Individual Variable Sections (-no_individual_data_sections) — [default]**

This option should be passed both when compiling and when linking. For more information, see “Individual Function and Variable Sections” on page 66.



Note

This option does not assign variables in custom data sections to individual sections. To assign variables in custom data sections to individual sections in addition to variables in default data sections, see “Individual Variable Sections for Renamed Data Sections” on page 98 and “Individual Variable Sections for User-Defined Data Sections” on page 99.

Individual Variable Sections for Renamed Data Sections

When the Individual Variable Sections support is enabled, controls whether the compiler will assign a global or static variable in a renamed data section declared by `#pragma ghs section settype="sectname"` directive to its own individual section, whose name is *sectname.symname*, where *sectname* is the renamed data section's name, and *symname* is the variable's symbol name. By default, the Individual Variable Sections support only assigns variables in default data sections to individual sections. Permitted settings for this option are:

- **Allocate Variables in Data Sections Renamed by #pragma to Individual Variable Sections (-individual_pragma_data_sections)**
- **Do Not Allocate Variables in Data Sections Renamed by #pragma to Individual Variable Sections (-no_individual_pragma_data_sections) — [default]**

For more information, see “Individual Function and Variable Sections” on page 66.

**Note**

This option is only effective if **-individual_data_sections** (see “Individual Variable Sections for Default Data Sections” on page 98) is also used.

Individual Variable Sections for User-Defined Data Sections

When the Individual Variable Sections support is enabled, controls whether the compiler will assign a global or static variable in a user-defined data section declared by `__attribute__((section("sectname")))` to its own individual section, whose name is `sectname.symname`, where `sectname` is the renamed data section's name, and `symname` is the variable's symbol name. By default, the Individual Variable Sections support only assigns variables in default data sections to individual sections. Permitted settings for this option are:

- **Allocate Variables in User-Defined Data Sections by `__attribute__` to Individual Variable Sections** (**-individual_attribute_data_sections**)
- **Do Not Allocate Variables in User-Defined Data Sections by `__attribute__` to Individual Variable Sections** (**-no_individual_attribute_data_sections**) — [default]

For more information, see “Individual Function and Variable Sections” on page 66.

**Note**

This option is only effective if **-individual_data_sections** (see “Individual Variable Sections for Default Data Sections” on page 98) is also used.

Individual Variable Sections for Nameless Variables

When used with **-individual_data_sections**, controls whether compiler-created objects that would be allocated to data sections are allocated to their own individual sections therein. Note that this only applies to objects not declared by the user. Permitted settings for this option are:

- **Individual Variable Sections for Nameless Variables** (**-individual_data_sections_for_nameless_vars**) — [default]

- **No Individual Variable Sections for Nameless Variables**
(**-no_individual_data_sections_for_nameless_vars**)



Note

This option is only effective if **-individual_data_sections** (see “Individual Variable Sections for Default Data Sections” on page 98) is also used.

Individual Function Sections for Default Text Section

Controls whether to enable the Individual Function Sections support and whether the compiler will assign a function (including static function) in a default text section to its own individual section, whose name is *sectname.symname*, where *sectname* is the default text section's name, and *symname* is the function's symbol name. Permitted settings for this option are:

- **Allocate Functions in Default Text Section to Individual Function Sections**
(**-individual_function_sections**)
- **Do Not Allocate Functions in Default Text Section to Individual Function Sections** (**-no_individual_function_sections**) — [default]

If this option is used in any compilation unit, it must also be passed to the linker. Note that this option does not need to be applied across all compilation units. For more information, see “Individual Function and Variable Sections” on page 66.



Note

This option does not assign functions in custom text sections to individual sections. To assign functions in custom text sections to individual sections in addition to functions in the default text section, see “Individual Function Sections for Renamed Text Sections” on page 100 and “Individual Function Sections for User-Defined Text Sections” on page 101.

Individual Function Sections for Renamed Text Sections

When the Individual Function Sections support is enabled, controls whether the compiler will assign a function (including static function) in a text section renamed by `#pragma ghs section secttype="sectname"` directive to its own individual section, whose name is *sectname.symname*, where *sectname* is the renamed text section's name, and *symname* is the function's symbol name. By default,

the Individual Function Sections support only assigns functions in the default text section to individual sections. Permitted settings for this option are:

- **Allocate Functions in Text Sections Renamed by #pragma to Individual Function Sections (-individual_pragma_function_sections)**
- **Do Not Allocate Functions in Text Sections Renamed by #pragma to Individual Function Sections (-no_individual_pragma_function_sections)** — [default]

For more information, see “Individual Function and Variable Sections” on page 66.



Note

This option is only effective if **-individual_function_sections** (see “Individual Function Sections for Default Text Section” on page 100) is also used.

Individual Function Sections for User-Defined Text Sections

When the Individual Function Sections support is enabled, controls whether the compiler will assign a function (including static function) in a user-defined text section by `__attribute__((section("sectname")))` directive to its own individual section, whose name is `sectname.symname`, where `sectname` is the user-defined text section's name, and `symname` is the function's symbol name. By default, the Individual Function Sections support only assigns functions in the default text section to individual sections. Permitted settings for this option are:

- **Allocate Functions in User-Defined Text Sections by __attribute__ to Individual Function Sections (-individual_attribute_function_sections)**
- **Do Not Allocate Functions in User-Defined Text Sections by __attribute__ to Individual Function Sections (-no_individual_attribute_function_sections)** — [default]

For more information, see “Individual Function and Variable Sections” on page 66.



Note

This option is only effective if **-individual_function_sections** (see “Individual Function Sections for Default Text Section” on page 100) is also used.

Extra Dot in Individual Section Name

Specifies whether the compiler will use an extra dot to separate base section names from variable or function names in individual section names. For example, if enabled, the individual section name for function `<func>` in `.text` would be `.text..<func>`. Normally, the individual section name would be `.text.<func>`. Permitted settings for this option are:

- **Extra Dot in Individual Section Name**
(**-individual_section_name_extra_dot**)
- **No Extra Dot in Individual Section Name**
(**-no_individual_section_name_extra_dot**) — [default]

Individual Link-Once Sections

Specifies whether functions or objects that would otherwise be allocated to link-once sections (such as template instantiations) are allocated to link-once versions of individual sections. This option is designed to be used in conjunction with options such as **-individual_function_sections** or **-individual_data_sections**. Permitted settings for this option are:

- **Individual Sections for Link-Once** (**-individual_linkonce_sections**) — [default]
- **No Individual Sections for Link-Once** (**-no_individual_linkonce_sections**)

Malloc Version

Indicates which `malloc` implementation to use. For most applications, **-malloc_version=arena** is recommended. For information about alternatives, see the option settings and the usage advice that follows. Note that if **-check=alloc** or **-check=memory** is passed, **-malloc_version** must be omitted or set to **-malloc_version=memcheck**. Permitted settings for this option are:

- **Legacy** (**-malloc_version=legacy**) — Specifies an older `malloc` implementation that was included with prior Compiler versions. This setting is not supported with INTEGRITY.

- **Small Block Pool (-malloc_version=smallblock)** — Specifies a `malloc` implementation that maintains a separate allocation pool for small blocks. This setting results in faster allocation times at the expense of increased heap fragmentation. This setting is the default when **-act_like** is set to `2018.5` or earlier.
- **Arena (-malloc_version=arena)** — Specifies a `malloc` implementation where freed blocks (small and large) are tracked in separate allocation pools for different size ranges. Operations on small blocks are done locklessly when there is low contention, which further decreases allocation times when `__ghsLock` and `__ghsUnlock` are not trivial (for example, on INTEGRITY). This version can also divide the heap into arenas. Different threads can allocate from different arenas, reducing contention in multi-threaded environments. The creation of multiple arenas is dependent on the target operating system. Contact Green Hills Support for more information about which operating systems support creating multiple arenas. This setting is the default unless **-act_like** is set to `2018.5` or earlier.
- **Real-Time (-malloc_version=realtime)** — Specifies a `malloc` implementation where freed blocks (small and large) are tracked in separate allocation pools for different size ranges. This allows `malloc`, `aligned_alloc`, and `free` to run in bounded time, with some caveats (see the list of **Real-Time** `malloc` caveats below). Operations on small blocks are done locklessly when there is low contention, which further decreases allocation times when `__ghsLock` and `__ghsUnlock` are not trivial (for example, on INTEGRITY).
- **Memory Checking Enabled (-malloc_version=memcheck)** — Specifies a version of `malloc` instrumented to detect certain types of memory allocation problems. This option is equivalent to **-check=alloc**.

Legacy, Small Block Pool, and Arena are general-purpose `mallocs`:

- **Arena** is typically the fastest, at the cost of slightly worse heap utilization and a larger library size (`.text` and `.data`) than the others.
- **Legacy** typically provides the best heap utilization and smallest library size, but is also significantly slower than the others.
- **Small Block Pool** has performance, heap utilization, and a library size that is generally between that provided by **Legacy** and **Arena**.

Real-Time and **Memory Checking Enabled** are special-purpose `mallocs`:

- **Real-Time** is appropriate if you need to dynamically allocate memory in a real-time system. Review the **Real-Time** `malloc` caveats list before using (see below). Note that this implementation has a similar heap utilization and library size to **Arena**.
- **Memory Checking Enabled** is intended for debugging application memory issues, and should not be used in a deployed system. For more information, see `-check=alloc` in “Run-Time Memory Checks” on page 138.

Real-Time `malloc` caveats:

- Bounded run-time does not apply when there are other threads in the same process that can call `__ghsLock` and `__ghsUnlock`. Except for lockless small block operations, accesses to `malloc`'s data structures are synchronized using that lock. This means that a call to `malloc` can be blocked indefinitely if another thread holds that lock. Various library functions call `__ghsLock` and `__ghsUnlock`, so this can occur even if the other thread never calls `malloc` or `free`. For help with avoiding all library functions which use that lock, contact Green Hills Support.
- `malloc` calls `__ghsLock`, `__ghsUnlock`, and `sbrk`. As such, in order for `malloc` to run in bounded time, these called functions must also run in bounded time. On stand-alone, the default implementations of all three functions do so. On INTEGRITY, the default implementations of `__ghsLock` and `__ghsUnlock` do so, provided that there is only one thread per process. For information regarding INTEGRITY's `sbrk` and bounded time, consult the *INTEGRITY Implementation-Dependent Behavior Guide* or contact Green Hills Support.
- Only the functions `malloc`, `aligned_alloc`, and `free` run in bounded time. The related functions `calloc` and `realloc` can run in linear time due to (respectively) zeroing or copying the allocated array.



Note

The **Arena** and **Real-Time** settings do not support MULTI's **heapview** command in MULTI 7.

**Note**

The options **-fast_malloc** and **-no_fast_malloc** are recognized for backward compatibility, but their use is deprecated. If you previously passed **-no_fast_malloc**, the equivalent setting is **-malloc_version=legacy**. If you previously passed **-fast_malloc**, the equivalent setting is **-malloc_version=smallblock**.

Number of Arenas for Malloc

Overrides the number of arenas created with the arena `malloc`. This requires an operating system that supports such configuration. The equivalent driver option is:

- **-malloc_num_arenas=***value*

The number of arenas can vary between 1 and 32, inclusive. Pass `-malloc_num_arenas=default` to let the operating system choose the number of arenas.

This option is only accepted with the arena `malloc` on INTEGRITY, and is ignored on INTEGRITY versions that do not support it. Contact Green Hills Support for information about the INTEGRITY versions that support this option.

Program is Single-Threaded

Controls whether to signal to the toolchain that the current address space is single-threaded. When enabled, the toolchain can make certain optimizations that would be invalid in a multi-threaded program, such as selectively omitting locking or synchronization code in thread-safe C library routines like `malloc`. Enabling this option when building a multi-threaded address space will result in incorrect behavior. Permitted settings for this option are:

- **On (-single_threaded)**
- **Off (-no_single_threaded)** — [default]

The **-single_threaded** setting is not supported with INTEGRITY shared libraries.



Note

Certain INTEGRITY modules may create additional threads, even if your code does not. Do not enable this option if any such modules have been included in your address space.

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.

Shared Libraries

Controls whether the application may link in shared libraries. Permitted settings for this option are:

- **No Shared Libraries** (**-non_shared**) — [default]
- **Link With Shared Libraries** (**-call_shared**)

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 hosts, we recommend that you use forward slashes (/) in the path. The equivalent driver option is:

- **-os_dir directory**

Kernel Project

This option is available for INTEGRITY projects only.

Controls whether the project is to be linked as a kernel or as a standard application. Permitted settings for this option are:

- **On (Link with the default kernel) (-kernel)** — Specifies linking with the default INTEGRITY kernel library. This kernel library is fully featured and provides full debug support.
- **Off (-no_kernel)** — Specifies that no INTEGRITY kernel library be linked in.
- **Link with kernel optimized for production use (-kernel=kernel_opt)** — Specifies linking with a production-optimized INTEGRITY kernel library that does not include debugging features. This option can only be used if you are running INTEGRITY version 11.4 or earlier.
- **Link with kernel with checked debug output enabled (-kernel=kernel_checked)** — Specifies linking with an INTEGRITY kernel library that is fully featured, and provides extra debug output and internal error checking for debug purposes. This option can only be used if you are running INTEGRITY version 11.4 or earlier. To replicate this behavior in later versions of INTEGRITY, use **-kernel** and build your project in the Checked configuration.
- **Link with velOSity kernel (-kernel=kernel_vel)**
- **Link with velOSity kernel with checked debug output enabled (-kernel=kernel_vel_checked)**
- **Link with velOSity kernel optimized for production use (-kernel=kernel_vel_opt)**

POSIX DLL

Generates an ELF executable format that is appropriate for INTEGRITY POSIX DLLs. Permitted settings for this option are:

- **On (--posix_dll)**
- **Off (--no_posix_dll)**

Full POSIX-2001 Compilation Support

Enables the C99 keyword support for POSIX. Permitted settings for this option are:

- **On** (`--integrity_posix`)
- **Off** (`--no_integrity_posix`)

INTEGRITY Posix Mode

Simplifies what libraries to link in, especially when dealing with shared libraries and POSIX. Permitted settings for this option are:

- **No POSIX Libraries** (`:integrity_posix_mode=noposix`)
- **Single Address Space POSIX Libraries** (`:integrity_posix_mode=single`)
- **Full POSIX System Libraries** (`:integrity_posix_mode=system`)

For more information, see “Introduction to POSIX on INTEGRITY” in the *INTEGRITY Libraries and Utilities User's Guide*.

Generated Header File Directory

This option is available for INTEGRITY projects only.

Sets the base directory to use for header file generation. It must be an existing directory. By default, it is the directory where the **.bsp** file resides. The equivalent **intex** option is:

- **-headerfiledir** *dir*

Dynamic Download Project

This option is available for INTEGRITY projects only.

Controls whether the project will be linked to be suitable for downloading to a running kernel. Permitted settings for this option are:

- **On** (**-dynamic**) — Is ignored if an **.int** file is specified.

- **Off (-no_dynamic)** — [default]

Display BootTable

This option is available for INTEGRITY projects only.

Controls the build-time display of the BootTable that the kernel processes at run time. Permitted settings for this option are:

- **On (-display_boot)**
- **Off (-no_display_boot)** — [default]

Delete Unused Kernel Calls

This option is only available for INTEGRITY projects and is no longer supported as of INTEGRITY 11.

Controls the deletion of unused kernel calls from the final image. Permitted settings for this option are:

- **On (-generatemonolithdeletions)**

The deletion is based on physical and virtual kernel calls referenced in the component `AddressSpaces`. This option may cause unpredictable errors if used in the compilation of a dynamic download project which could use a deleted kernel call.

Create Raw Binary Image

Controls the creation of a binary image in addition to the standard ELF image. This option applies only to projects that contain an Integrate (**.int**) file. If you want to generate a binary image for anything else, see “Generate Additional Output” on page 228. Permitted settings for this option are:

- **Generate Default Binary Image (-memory)** — Generates a binary memory image output file with the name of the ELF executable plus a **.bin** extension.
- **Generate User-Specified Binary Image (-memory=*filename*)** — Creates a binary image with the specified *filename*.

Create S-Record File Image

Creates an S-Record file image. For more information, see “The gsrec Utility Program” on page 532. The equivalent driver option is:

- **-srec**

Additional Intex Options

This option is available for INTEGRITY projects only.

Specifies additional **intex** options. The equivalent **intex** option is:

- **-intexoption** *option*

Generated Header File Name

This option is for INTEGRITY projects only.

Sets the name for the default header file to use for header file generation. Default is *basename_of_.int_file_integrate.h*. The equivalent **intex** option is:

- **-headerfilename**

For more information about header file generation, see the "Header File Generation" section in the *Integrate User's Guide*.

Include File Search Directory

This option is available for INTEGRITY projects only.

Adds the specified directory to the search path for `#include` files. The equivalent **intex** option is:

- **-includefiledir**

INTEGRITY Application Stripping

This option is available for INTEGRITY projects only.

Controls *stripping* of the executable at the conclusion of linking. Permitted settings for this option are:

- **On** (**-strip_application**)
- **Off** (**-no_strip_application**) — [default]

Intex Variable Definition

This option is available for INTEGRITY projects only.

Overwrites an entry in a `Variables` section. *name* indicates the name of the *Variable* to be overwritten. *value* indicates the new value of the variable. The equivalent **intex** option is:

- **-intex_var** *name=value*

Quit Building if Intex Warnings are Generated

This option is available for INTEGRITY projects only.

Controls whether building will cease upon the generation of any warning. Permitted settings for this option are:

- **On** (**--quit_after_intex_warnings**)
- **Off** (**--no_quit_after_intex_warnings**) — [default]

SHA-1

This option is available for INTEGRITY projects only.

Prepares the final image for an SHA-1 consistency check upon boot. Use this option in conjunction with **libsha1.a** in the INTEGRITY kernel. Permitted settings for this option are:

- **On (-sha1)**

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 269.

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. In general, you should not change the default **-act_like** setting for products in which it is already set (such as INTEGRITY). Exceptions include being explicitly instructed to do so by Green Hills documentation or Green Hills Support.

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.
- **2012.1 (-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.
- **2013.1** (**-act_like=2013.1**) — Act similarly to Green Hills Compiler 2013.1.
- **2013.5** (**-act_like=2013.5**) — Act similarly to Green Hills Compiler 2013.5.
- **2014.1** (**-act_like=2014.1**) — Act similarly to Green Hills Compiler 2014.1.
- **2015.1** (**-act_like=2015.1**) — Act similarly to Green Hills Compiler 2015.1.
- **2016.5** (**-act_like=2016.5**) — Act similarly to Green Hills Compiler 2016.5.
- **2017.1** (**-act_like=2017.1**) — Act similarly to Green Hills Compiler 2017.1.
- **2017.5** (**-act_like=2017.5**) — Act similarly to Green Hills Compiler 2017.5.
- **2018.1** (**-act_like=2018.1**) — Act similarly to Green Hills Compiler 2018.1.
- **2018.5** (**-act_like=2018.5**) — Act similarly to Green Hills Compiler 2018.5.
- **2019.1** (**-act_like=2019.1**) — Act similarly to Green Hills Compiler 2019.1.
- **2019.5** (**-act_like=2019.5**) — Act similarly to Green Hills Compiler 2019.5.
- **2020.1** (**-act_like=2020.1**) — Act similarly to Green Hills Compiler 2020.1.
- **2020.5** (**-act_like=2020.5**) — Act similarly to Green Hills Compiler 2020.5.
- **2021.1** (**-act_like=2021.1**) — Act similarly to Green Hills Compiler 2021.1.
- **Latest** (**-act_like=latest**) — [default for non-INTEGRITY] 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:

- **-Idirectory**

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 57.

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:

- **-Llibrary_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.

RTOS Library Directories

Specifies directories in which the linker searches for system libraries. This option may be specified multiple times. Any directory specified without an absolute path

will be located relative to the directory specified by **-os_dir**. The equivalent driver option is:

- **-rtos_library_directory** *directory*

Libraries

Specifies a *library* for the builder to link against. The equivalent driver option is:

- **-llibrary**

If *library* does not have an extension or directory path, it is assumed to be an abbreviation for **liblibrary.a**. If *library* does not have a directory path, the driver and linker search for **liblibrary.a** in the directories specified by the **-Llibrary_directory** 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 *library* has a directory path, *library* 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 **-llibrary** 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**.



Note

Some output files are dependent on their root name: their filename up to the first instance of a period. For example, the root name of **Foo3.5.gpj** is **Foo3**. Consequently, programs, libraries, and other build product files should be given unique root names. Doing otherwise may produce unexpected behavior.

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 a wide range of optimizations to help you produce smaller and/or faster executables.

Optimization Strategy

Specifies the overall optimization strategy used by the compiler. When deciding how to optimize your program, begin by selecting the optimization strategy that is the best fit for your project. To determine fit, read the available optimization strategies below, then compare their relative impacts in the table that follows.

To learn more about optimizations or to fine-tune a strategy to your specifications, see Chapter 4, “Optimizing Your Programs” on page 325 and “Optimization Options” on page 286. Permitted settings for this option are:

- **Maximum Debugging And No Inlining (-Omaxdebug)** — Disables inlining and all optimizations. Consider this strategy if you are debugging code that has many inlined functions (but see the note listed below).
- **Maximum Debugging And Limited Optimizations (-Omoredebug)** — [default] Enables optimizations that do not affect debugging. Consider this strategy if you are not size-restricted and you use the same build for both production and development. Note that **-Omoredebug** does not allow the selection of individual optimizations.
- **Optimize for Debuggability (-Odebug)** — Enables optimizations that do not affect debugging. Consider this strategy if you are not size-restricted and you use the same build for both production and development. Note that **-Odebug** does not allow the selection of individual optimizations.
- **Optimize for General Use (-Ogeneral or -O)** — Enables optimizations that improve both size and performance. Consider this strategy if program size and speed are of equal priority, and you have separate builds for production and development.
- **Optimize for Size (-Osize or -Ospace)** — Enables optimizations that improve size and performance, while favoring additional size optimizations that limit performance gains. Consider this strategy if size minimization is your highest

priority, and you have separate builds for production and development. Note that, to be effective, **-Osize**, should be applied as globally as possible.

- **Optimize for Speed (-Ospeed)** — Enables optimizations that improve performance, but may increase code size. Consider this strategy if execution speed is your highest priority. Note that, unlike other strategies, **-Ospeed** is typically employed selectively, targeting files and functions that use the most program execution time. This is because optimizing for speed tends to markedly reduce debuggability and significantly increase compilation time.
- **No Optimizations (-Onone)** — Disables all optimizations and provides the most straightforward code generation. Inlining of explicitly marked functions is performed at this level. Consider this strategy if compilation speed is your highest priority.



Note

-Omaxdebug may cause undefined symbol errors at link time for non-conforming uses of non-static inline functions in C code. This is because when the `inline` keyword is applied to a function, **-Omaxdebug** inhibits the inlining that would otherwise occur, but it does not modify the effects that the `inline` keyword has on the linkage of non-static inline functions. In particular, code that defines and calls imported inline C functions without supplying an external definition will often build properly when **-Omaxdebug** is not used. This is because the `inline` request is normally honored and, therefore, the lack of an exported definition will not be detected. However, when **-Omaxdebug** is used, the inlining is inhibited, so the linker will output undefined symbol errors if the code is rebuilt with **-Omaxdebug**.

In the table below, the impact of optimization strategies on the improvement (+) or worsening (-) of program characteristics is shown relative to a non-optimized compilation (**-Onone**). An equals sign (=) signifies no improvement.

Strategy	Setting	Debugging Ability	Code Reduction	Compilation Speed	Execution Speed
Maximum Debugging and No Inlining	-Omaxdebug	+++	-	=	-

Maximum Debugging and Limited Optimizations	-Omoredebug	++	+	-	+
Optimize for Debuggability	-Odebug	=	++	-	++
Optimize for General Use	-Ogeneral	--	++++	--	++++
Optimize for Size	-Osize	--	+++++	--	+++
Optimize for Speed	-Ospeed	---	+++	---	+++++
No Optimizations	-Onone	=	=	=	=

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 326.

Automatic Vector Optimization

Controls Vector Optimizations. If no optimization strategy is selected, this option also enables **-Ospeed**. Permitted settings for this option are:

- **On (-OV)**
- **Off (-Onovector)** — [default]

For more information, see “SIMD Vectorization” on page 346.

Linker Optimizations

Controls the linker optimizations listed in “Linker Optimizations” on page 235. Permitted settings for this option are:

- **On (-Olink)** — Enables certain linker optimizations depending on your overall optimization strategy. Many of these optimizations make complex changes to your code. These changes may slow down the link stage, produce code that is harder to debug, or have other drawbacks. To disable individual linker optimizations, set them to **Off** in combination with this option.

Enabling **-Olink** implicitly enables **-delete** and **-uvfd**. In addition, enabling **-Olink** implicitly enables **-codefactor** if **-Osize** is also enabled. When building shared objects, this could result in unresolved symbols. For more information about linker-specific options, see “Linker-Specific Options” on page 424. For more information about **-delete**, see “Deleting Unused Functions” on page 459. For more information about **-codefactor**, see “Code Factoring” on page 237.

- **Off (-Onolink)** — [default]



Note

The setting **-Olink** may cause DWARF debugging information to be out of sync with the program in memory.

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 Interprocedural Optimizations (-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**.
- **Standard Interprocedural Optimizations (-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 335.



Note

To function correctly, this option's setting must be uniformly applied to all parts of a program.

Optimization Scope

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

These options provide fine-grained control over major optimization processes.

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 considers larger functions when inlining and larger loops when unrolling, where size is determined by approximate number of expressions. If no optimization strategy is selected, this option enables **-Ospeed**

and implies **-Oloop**. If optimization strategies **-Osize** or **-Ogeneral** are selected, this option implies **-Oloop**. If **-Oloop** is enabled, this option implies **-Ounrollmore**. 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.

Unroll Loops More

Controls the amount of loop unrolling performed by the compiler. When this option is enabled, larger loops will be considered for unrolling, and loops will sometimes be unrolled with a larger unrolling factor. This option has no effect unless loop optimizations and loop unrolling are enabled, such as by selecting **-Ospeed** as an optimization setting. This option defaults to **On** when using **-Ospeed** or when using both **-Oloop** and **-OB**. In all other circumstances, it defaults to **Off**. Permitted settings for this option are:

- **On (-Ounrollmore)**
- **Off (-Onounrollmore)**

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 424.

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 that lists 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 520 for more information).

Interprocedural Optimization Scope

This option category is contained within **Optimization**→**Optimization Scope**.

The following interprocedural optimization options are designed for debugging unforeseen interactions with your code. They should only be used when indicated by Green Hills Support. For more information about interprocedural optimizations, see “Interprocedural Optimizations” on page 335.

IP One-Site Inlining

When using **-Ointerproc** or **-Owholeprogram**, controls whether single call site functions can be inlined. Permitted settings for this option are:

- **On (-Oiponesiteinlining)** — [default]
- **Off (-Onoiponesiteinlining)**



Note

To function correctly, this option's setting must be uniformly applied to all parts of a program.

IP Delete Functions

When using **-Ointerproc** or **-Owholeprogram**, controls whether unreachable functions can be deleted. Even if functions are not deleted as part of an interprocedural optimization pass, they may be deleted due to other optimization passes. Permitted settings for this option are:

- **On (-Oipdeletefunctions)** — [default]
- **Off (-Onoipdeletefunctions)**

IP Delete Globals

When using **-Owholeprogram**, controls whether constant-value, directly referenced global variables can be deleted. Permitted settings for this option are:

- **On (-Oipdeleteglobals)** — [default]

- **Off (-Onoipdeleteglobals)**

IP Localize Globals

When using **-Ointerproc** or **-Owholeprogram**, controls whether some global variables are replaced with local register variables. For example:

```
int foo = 0;

int main(void)
{
    foo = rand();
    return foo;
}
```

In the above case, the global variable `foo` can be replaced with the local variable `foo` in `main()`, as `foo` is only used in this function. Note that this optimization is most effective when used with **-Owholeprogram**. Permitted settings for this option are:

- **On (-Oiplocalizeglobals)** — [default]
- **Off (-Onoiplocalizeglobals)**

IP Constant Globals

When using **-Ointerproc** or **-Owholeprogram**, controls whether some initialized, unchanging global variables are replaced with their constant values. This optimization requires all such global variables to be directly referenced. Permitted settings for this option are:

- **On (-Oipconstglobals)** — [default]
- **Off (-Onoipconstglobals)**

IP Small Inlining

When using **-Ointerproc** or **-Owholeprogram**, controls whether to inline small functions. This option applies to functions whose inlined size may be less than the size of their equivalent function call. Permitted settings for this option are:

- **On (-Oipsmallinlining)** — [default]
- **Off (-Onoipsmallinlining)**

IP Constant Propagation

When using **-Owholeprogram**, controls whether some function parameters are replaced by constant values. This option applies to function parameters that are always set to the same constant value by callers. Permitted settings for this option are:

- **On (-Oipconstprop)** — [default]
- **Off (-Onoipconstprop)**

IP Remove Parameters

When using **-Owholeprogram**, controls whether some non-required function parameters can be removed. Other functions will not pass arguments using removed parameters. Permitted settings for this option are:

- **On (-Oipremoveparams)** — [default]
- **Off (-Onoipremoveparams)**

IP Limit Inlining

When using **-Ointerproc** or **-Owholeprogram**, controls whether to limit the amount of intermodule inlining performed (**-OI**). Permitted settings for this option are:

- **On (-Oiplimitinlining)** — [default]
- **Off (-Onoiplimitinlining)**

IP Constant Returns

When using **-Ointerproc** or **-Owholeprogram**, controls whether functions returning constant values can have their values propagated to callers. Permitted settings for this option are:

- **On (-Oipconstantreturns)** — [default]
- **Off (-Onoipconstantreturns)**

IP Remove Returns

When using **-Owholeprogram**, controls whether unused function call results can have their return statements removed. This only applies to return variables that are never used at any call sites, or constant return values. Permitted settings for this option are:

- **On (-Oipremovereturns)** — [default]
- **Off (-Onoipremovereturns)**

IP Alias Reads

When using **-Ointerproc** or **-Owholeprogram**, controls what memory can be read by function calls. This setting is strictly for testing under the direction of Green Hills Support. Permitted settings for this option are:

- **On (-Oipaliasreads)** — [default]
- **Off (-Onoipaliasreads)**

IP Alias Writes

When using **-Ointerproc** or **-Owholeprogram**, determines what memory can be written by function calls. This setting is strictly for testing under the direction of Green Hills Support. Permitted settings for this option are:

- **On (-Oipaliaswrites)** — [default]
- **Off (-Onoipaliaswrites)**

IP Alias Lib Functions

When using **-Ointerproc** or **-Owholeprogram**, controls the use of interprocedural alias analysis on some common library functions (such as `printf`, `strlen`, and `fopen`). Permitted settings for this option are:

- **On** (**-Oipaliaslibfuncs**) — [default]
- **Off** (**-Onoipaliaslibfuncs**)

IP Allocate Variables

When using **-Ointerproc** or **-Owholeprogram**, controls whether global and static variables may be allocated during interprocedural analysis. This can improve code quality when functions reference global variables or are inlined into callers that are defined in different modules. Note that this optimization will not function without using **--no_commons**. Permitted settings for this option are:

- **On** (**-Oipallocvars**) — [default]
- **Off** (**-Onoipallocvars**)

Individual Functions

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

These options allow you to specify individual functions for optimization.

Optimize Specific Functions for Speed

Prefer speed over size when optimizing specific functions. The equivalent driver option is:

- **--fast=***function1*[*function2*]...

This option has no effect when used with the optimization strategies **-Onone**, **-Omoredebug**, **-Odebug**, and **-Omaxdebug**. The speed optimizations enabled by **--fast** will not always be identical to those enabled by **-Ospeed**.

Optimize Specific Functions for Size

Prefer size over speed when optimizing specific functions. The equivalent driver option is:

- **--small**=*function1[,function2]*...

This option has no effect when used with the optimization strategies **-Onone**, **-Omoredebug**, **-Odebug**, and **-Omaxdebug**. The size optimizations enabled by **--small** will not always be identical to those enabled by **-Osize**.

Inline Specific Functions

Enables two-pass inlining for specific functions. If no optimization strategy is selected, this option enables **-Ospeed**. The equivalent driver option is:

- **-OI**=*function1[,function2]*...

For more information, see “Inlining Optimizations” on page 326.

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 (Including SIMD Vectorization)” on page 340.

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 38 and “Obtaining Profiling Information” on page 46.

For additional, more specialized options, see “Debugging Options” on page 294.

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 from the MULTI Debugger's command line.
- **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.

INTEGRITY TimeMachine

Controls whether the linker keeps INTEGRITY TimeMachine instrumentation. Permitted settings for this option are:

- **Off (-strip_timemachine)** — [default]
- **On (-record_timemachine)**

To enable INTEGRITY TimeMachine at compile time, see “INTEGRITY TimeMachine Enabled” on page 297. This feature is licensed separately from the Green Hills toolchain.

Function Entry/Exit Logging

Controls how the linker treats function entry/exit instrumentation. Permitted settings for this option are:

- **Enable (-record_entry_exit_history)** — Function entry/exit instrumentation is not replaced or removed by the linker.
- **Disable (-mute_entry_exit_history)** — Function entry/exit instrumentation is disabled by the linker and replaced with `NOP` instructions.
- **Strip (-strip_entry_exit_history)** — [default] Function entry/exit instrumentation is disabled by the linker and stripped from the executable. (Stripped code is similar though not identical to code without function entry/exit instrumentation.)

For more information, see “Function Entry/Exit History” on page 45. This feature is licensed separately from the Green Hills toolchain.



Note

The settings **-record_entry_exit_history** and **-strip_entry_exit_history** may cause DWARF debugging information to be out of sync with the program in memory.



Note

In rare cases, the setting **-record_entry_exit_history** can interfere with trace features in MULTI 7. Similar problems may occur with **-instrument_prologue_epilogue**. For more information about these issues, see “Instrument Prologue/Epilogue” on page 96).

EAGLE Logging

Controls the global inclusion of EAGLE functionality at link time. Permitted settings for this option are:

- **Off (-strip_eagle_log)** — [default] EAGLE logging is stripped from the code by the linker. (Stripped code is similar though not identical to code without EAGLE logging.)
- **On (-enable_eagle_log)** — Enables EAGLE logging.

This feature is licensed separately from the Green Hills toolchain.



Note

The setting **-strip_eagle_log** may cause DWARF debugging information to be out of sync with the program in memory.

Profiling - Block Coverage

Controls *block coverage* profiling, which records a flag or count for each basic block in your program. Basic blocks are instruction sequences with no branches in or out save at entry and exit. Permitted settings for this option are:

- **Flag (-coverage=flag)** — Records whether each block is executed.
- **Count (32-bit counters) (-coverage=count)** — Records how many times each block is executed using a 32-bit counter.
- **Count (64-bit counters) (-coverage=count64)** — Records how many times each block is executed using a 64-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 47.

Function Entry/Exit History Instruction Threshold

Sets the minimum number of instructions required for entry/exit history instrumentation. Functions with fewer instructions will be excluded from entry/exit instrumentation, which can assist in reducing logging overhead at the cost of coverage. Setting this option to 0 will cause entry/exit history instrumentation to be generated for all functions. The equivalent driver option is:

- **-entry_exit_history_inst_threshold=value**

**Note**

Instruction counts are made using the compiler's internal representation of instructions, which may not match the number of instructions in the final function.

Profiling - Target-Based Timing

Controls *timing* profiling, which records the time spent in each function. See your operating system's documentation for relevant compatibility information. 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 49.

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*]...

To enable run-time error checks for all types, pass:

-check=all

To enable all checks except those of a certain type, pass either of the following:

-check=all, *notype*

-check=all **-check**=*notype*

**Note**

Run-time error checks are also enabled by some of the rules specified in the Motor Industry Software Reliability Association (MISRA) standards. For more on MISRA, see “MISRA C 1998” on page 718, “MISRA C 2004” on page 721, and “MISRA C 2012” on page 723.

The various checks generate 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 an error if a value assigned to an integral or enumerated object is out-of-bounds for said object. (For example, assigning `long long` to an `int` object.) Errors to `enum` assignments are issued only when the value is smaller than the smallest `enum` value, or larger than the largest `enum` value. When this checking is enabled, certain instances of diagnostic 68 (`integer_sign_change`) and diagnostic 69 (`integer_truncated`) will be issued with a default severity of *warning*, rather than *remark*.

This check generates the following messages at run-time:

- Assignment out of range of integral/enum type
- Value outside of type

Note that when debugging your program with MULTI 7, the first run-time message may be replaced with a different message with identical meaning:

- Assignment out of bounds

This behavior will not occur when running your program outside of the Debugger.

- **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 parameters declared with array types. For example, 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];
}
```

This check generates the following message at run-time:

- Array index out of bounds

- **Case Label Bounds (-check=switch)** — Generates an error if the `switch` expression does not match any of the `case` labels. This does not apply when a default case label is used.

This check generates the following message at run-time:

- Case/switch index out of bounds

- **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.

This check generates the following message at run-time:

- Divide by 0

- **Nil Pointer Dereference (-check=nilderef)** — Generates an error for dereferences of null pointers.

This check generates the following message at run-time:

- Nil pointer dereference

- **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.

This check generates the following message at run-time:

- Write to watchpoint

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. 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. Note that this setting is equivalent to **-malloc_version=memcheck**.
- **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 file 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 target-walkable stack. If you enable memory checking, you must use **-gtws** when compiling any function that may call `malloc` (directly or indirectly). Note that **-gtws** is the default when **-check=alloc** or **-check=memory** is enabled. As such, if either is passed to your entire project, **-gtws** will be enabled automatically.

For detailed information, see “Enabling General Memory Allocation Checking” on page 52, “Enabling Intensive Memory Allocation Checking” on page 53, and the documentation about viewing memory allocation information in the *MULTI: Debugging* book

Number of Callers to Track with Run-Time Memory Checks

When run-time memory checking is enabled, specifies the number of callers that should be tracked and reported for each allocation. This affects the number of callers displayed in the **Allocations** and **Leaks** tabs of MULTI's **Memory Allocations** window. For more information, see the documentation about viewing memory allocation information in the *MULTI: Debugging* book. The equivalent driver option is:

- **-memcheck_num_callers**=*n*, where *n* is the number of callers.

You can revert back to the default number of callers (8) by passing **-memcheck_num_callers=default**. If you are not using memory checking, passing this will serve to disable this option and any associated diagnostics.

Note the following caveats:

- Information about callers for each allocation is stored on the heap, so increasing this value will increase your program's heap usage.
- This option may not be used to decrease the number of callers tracked below 5.
- The number of callers displayed will be limited to what is supported by MULTI. With MULTI 7, any additional callers beyond 5 will not be displayed.

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 304.

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:

- **-Dsymbol[=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:

- **-Usymbol**

This is similar 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 an `#include` directive during normal compilation. Files are compiled and linked normally.

Permitted settings for this option are:

- **On** (**-H** | **--trace_includes**)
- **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 details about Green Hills C and the differences between the various modes, see Chapter 11, “Green Hills C” on page 575.

For additional, more specialized options, see “C/C++ Compiler Options” on page 308.

C Language Dialect

Controls the version of C accepted by the compiler. For more on the use and selection of C a version, see “Specifying a C Language Dialect” on page 576. Permitted settings for this option are:

- **Strict ANSI C (-ANSI)** — Specifies strict ANSI. This mode is compliant with the ANSI X3.159-1989 standard and disallows most non-standard constructs.
- **ANSI C (-ansi)** — 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 577.
- **Strict ISO C99 (-C99)** — Specifies strict ISO C99. This mode is compliant with ISO/IEC 9899:1999 and disallows most non-standard constructs.
- **ISO C99 (-c99)** — [default] 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 ISO C11 (-C11)** — Specifies strict ISO C11. This mode is compliant with ISO/IEC 9899:2018 and disallows most non-standard constructs.
- **ISO C11 (-c11)** — Specifies ISO C11. This mode provides all features of the ISO C11 language in addition to some extensions. Some non-standard constructs produce warnings rather than errors.
- **Strict ISO C17 (-C17)** — Specifies strict ISO C17. This mode is compliant with ISO/IEC 9899:2018 and disallows most non-standard constructs.
- **ISO C17 (-c17)** — Specifies ISO C17. This mode provides all features of the ISO C17 language in addition to some extensions. Some non-standard constructs produce warnings rather than errors.

- **Strict ISO C18 (-C18)** — Specifies strict ISO C18. This mode is compliant with ISO/IEC 9899:2018 and disallows most non-standard constructs.
- **ISO C18 (-c18)** — Specifies ISO C18. This mode provides all features of the ISO C18 language in addition to some extensions. Some non-standard constructs produce warnings rather than errors.
- **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 588.
- **GNU C99 (-gnu99)** — Specifies a GNU-compatible version of C99 mode, supporting many GNU features not explicitly included in ISO C99.

**Note**

If **-act_like** is set to `2015.1` or earlier, the compiler will default to ANSI (**-ansi**).

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 595.

C++ Language Dialect

Specifies the version of C++ to be accepted by the compiler. For more information, see “Specifying a C++ Language Dialect” on page 666. Permitted settings for this option are:

- **Standard C++ (Violations Give Errors) (--STD)** — Specifies C++ Strict Standard Mode, which gives errors when most non-standard features are used and disables features that conflict with Standard C++. For more information, see “Standard C++” on page 669.

- **Standard C++ (Violations Give Warnings) (--std)** — [default] Specifies C++ Standard Mode, which gives warnings when most non-standard features are used and disables features that conflict with Standard C++. For more information, see “Standard C++” on page 669.
- **C++11 (Violations Give Errors) (--C++11)** — Specifies C++11 Strict Mode, which gives errors when most non-standard features are used and disables features that conflict with C++11. For more information, see “C++11” on page 672.
- **C++11 (Violations Give Warnings) (--c++11)** — Specifies C++11 Standard Mode, which gives warnings when most non-standard features are used and disables features that conflict with C++11. For more information, see “C++11” on page 672.
- **C++14 (Violations Give Errors) (--C++14)** — Specifies C++14 Strict Mode, which gives errors when most non-standard features are used and disables features that conflict with C++14. For more information, see “C++14” on page 676.
- **C++14 (Violations Give Warnings) (--c++14)** — Specifies C++14 Standard Mode, which gives warnings when most non-standard features are used and disables features that conflict with C++14. For more information, see “C++14” on page 676.
- **C++17 (Violations Give Errors) (--C++17)** — Specifies C++17 Strict Mode, which gives errors when most non-standard features are used and disables features that conflict with C++17. For more information, see “C++17” on page 677.
- **C++17 (Violations Give Warnings) (--c++17)** — Specifies C++17 Standard Mode, which gives warnings when most non-standard features are used and disables features that conflict with C++17. For more information, see “C++17” on page 677.
- **Standard C++ with ARM Extensions (--arm)** — Specifies Standard C++ with extensions, which extends the standard with many useful and compatible constructs. This dialect is not supported on INTEGRITY projects. For more information, see “Standard C++ with ARM Extensions” on page 681.
- **GNU C++ (--g++)** — Specifies GNU C++ mode. This dialect is not supported on INTEGRITY projects. For more information, see “GNU C++” on page 681.
- **Embedded C++ (--e)** — Specifies Embedded C++. For more information, see “Embedded C++” on page 679.

- **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 681.

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 667. Note that **--eel**, **--eele**, **--el**, and **--ele** are not supported in language modes for C++11 or later.

C++ Exception Handling

Controls support for exception handling. When enabled, code size and speed may be impacted even when exception handling is not directly used. Likewise, heap usage will increase due to the allocation of exception-handling buffers during program startup. Throwing large exception objects or having many concurrently alive exception objects may cause additional buffers to be allocated from the heap. Note that enabling this option will also enable **--rtti**, even if **--no_rtti** is specified. Permitted settings for this option are:

- **On (--exceptions)**
- **Off (--no_exceptions)** — [default]



Note

To function correctly, this option's setting must be uniformly applied to all parts of a program.

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.

Thread-Local Storage

Controls support for the C11 and C++11 `thread_local` storage class specifier, used to indicate that a variable should reside in thread-local storage. Permitted settings for this option are:

- **On** (`--thread_local_storage`)
- **Off** (`--no_thread_local_storage`) — [default]

For more information, see “Threads, Thread-Local Storage, and Synchronization Primitives” on page 672.



Note

Unless `-act_like` is set to `2015.1` or earlier, this option will default to `--thread_local_storage` for INTEGRITY and u-velOSity in language C11 or later dialects and C++11 or later dialects.

MISRA C

This option category is contained within **C/C++ Compiler**.

MISRA C - Mandatory Rules Level

Controls the diagnostic messages generated upon a violation of the MISRA mandatory rules. Note that mandatory rules are only present in MISRA 2012. Permitted settings for this option are:

- **Errors** (`--misra_mand=error`) — [default]
- **Warnings** (`--misra_mand=warn`)
- **Silent** (`--misra_mand=silent`)



Note

Rules that are checked at run-time are not affected by this setting. Such rules can be individually turned off or collectively disabled by passing `--no_misra_runtime`.

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`)



Note

Rules that are checked at run-time are not affected by this setting. Such rules can be individually turned off or collectively disabled by passing `--no_misra_runtime`.

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 - Enforcement and Run-Time

Controls whether validation is enabled for any MISRA rules or directives that can impact run-time execution.

- **On** (`--misra_runtime`) — [default]
- **Off** (`--no_misra_runtime`)

Guidelines that can impact run-time execution include:

- MISRA 1998 rules 4, 38, and 107.
- MISRA 2004 rules 12.8 and 21.1.
- MISRA 2012 directive 4.1 and rules 7.4, 12.2, 22.1, and 22.2.

MISRA C 1998

This option category is contained within **C/C++ Compiler**.

Specifies the use of compliance checks for the 1998 Motor Industry Software Reliability Association (MISRA) standard. MISRA 1998 is a set of C guidelines designed to enforce good engineering practices when developing embedded automotive systems. For more information, see “MISRA C 1998” on page 718.

To reference the diagnostic numbers listed below, see *MULTI: Toolchain Error Messages and Other Diagnostics*.



Note

MISRA options function as an added layer of checking on top of C90- or C99-conformant code. Consequently, generic compiler errors may supersede MISRA-specific diagnostics, regardless of whether the code violates any MISRA guidelines.



Note

Support for MISRA 1998 is deprecated and may be removed in future versions. It is recommended that you use the MISRA 2004 Rules (see “MISRA C 2004” on page 159).

MISRA C 1998 Rules - Environment

Permitted settings for this option are:

- **1. [R] ISO 9899 C conformance w/o extensions (--saferc=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.
- **2. [A] Code other than C must conform to standard interface (--saferc=2)** — Not enforced.
- **3. [A] Inline assembly only in functions with no other code (--saferc=3)**
- **4. [A] Run-time checking (--saferc=4)** — 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 135. The option `--no_misra_runtime` turns off this rule completely. The option `--no_misra_runtime` disables both this rule and rule 107, as well as the run-time component of rule 38.

MISRA C 1998 Rules - Character Set

Permitted settings for this option are:

- **5. [R] Only ISO C characters and escape sequences used (`--saferc=5`)** — 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 (`--saferc=6`)** — 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 (`--saferc=7`)**
- **8. [R] No multibyte chars and wide strings (`--saferc=8`)** — 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 (`--saferc=9`)**
- **10. [A] No 'commented out' sections of code (`--saferc=10`)** — 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 (--saferc=11)** — 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 (--saferc=112)**

MISRA C 1998 Rules - Types

Permitted settings for this option are:

- **13. [A] Basic types used only in `typedef's (--saferc=13)**
- **14. [R] `char' always used as `signed char' or `unsigned char' (--saferc=14)**
- **15. [A] Floating point implementation complies to a standard (--saferc=15)** — 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 (--saferc=16)** — 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 (--saferc=117)**

MISRA C 1998 Rules - Constants

Permitted settings for this option are:

- **18. [A] Numeric constants need suffixes when appropriate (--saferc=18)**
- **19. [R] No octal constants (other than zero) (--saferc=19)**

MISRA C 1998 Rules - Declarations and Definitions

Permitted settings for this option are:

- **20. [R] All objects and functions declared before used (--saferc=20)**
- **21. [R] No use of same id name in inner and outer scope (--saferc=21)**
- **22. [A] Function scope declarations whenever possible (--saferc=22) —**
Not enforced.
- **23. [A] Static linkage of file scope declarations when possible (--saferc=23)**
— Not enforced.
- **24. [R] No internal and external linkages of same identifiers (--saferc=24)**
- **25. [R] Only one external definition of external identifier (--saferc=25) —**
Enforced in the compiler and elxr.

Associated diagnostics include 1797.

- **26. [R] Compatible multiple declarations of the same object/function (--saferc=26) —** 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 (--saferc=27) —**
Not enforced.
- **28. [A] The 'register' storage class specifier should not be used (--saferc=28)**
- **29. [R] The use of tag shall agree with its declaration (--saferc=29)**

MISRA C 1998 Rules - Initialization

Permitted settings for this option are:

- **30. [R] Automatic variables initialized before used (--saferc=30) —** Enforced for simple cases.
- **31. [R] Braces used in non-zero initialization of arrays/structs (--saferc=31)**
- **32. [R] All or only first enumerator may be explicitly initialized (--saferc=32)**

MISRA C 1998 Rules - Operators

Permitted settings for this option are:

- **33. [R] No side effects in right hand operand of `&&` or `||` (`--saferc=33`)**
- **34. [R] Operands of `&&` and `||` shall be primary expressions (`--saferc=34`)**
- **35. [R] Assignment operators not used in Boolean expressions (`--saferc=35`)**
- **36. [A] Logical and bitwise operators should not be confused (`--saferc=36`)**
— Not enforced.
- **37. [R] No bitwise operations on signed integer types (`--saferc=37`)**
- **38. [R] Right hand value of shift operand must be in range (`--saferc=38`)**
— 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 (`--saferc=39`)**
- **40. [A] No side effects in the `sizeof` operand (`--saferc=40`)**
- **41. [A] Implementation of division determined and documented (`--saferc=41`)** — This behavior is dependent on the hardware configuration of the target. Most Green Hills compilers, however, round the division result toward 0.
- **42. [R] No comma operators except in control expr of `for` loops (`--saferc=42`)**

MISRA C 1998 Rules - Conversions

Permitted settings for this option are:

- **43. [R] No implicit conversions which might lose information (`--saferc=43`)**
- **44. [A] Redundant explicit cast should not be used (`--saferc=44`)**
- **45. [R] Type casting to or from pointers should not be used (`--saferc=45`)**

MISRA C 1998 Rules - Expressions

Permitted settings for this option are:

- **46. [R] No expression with values dependent on evaluation order** (`--saferc=46`) — Not enforced.
- **47. [A] No dependence placed on C precedence rules** (`--saferc=47`)
- **48. [A] Mixed precision arithmetic must use explicit casting** (`--saferc=48`)
- **49. [A] Test value against zero unless expression Boolean** (`--saferc=49`) — 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** (`--saferc=50`)
- **51. [A] No wraparound in constant unsigned expressions** (`--saferc=51`)

MISRA C 1998 Rules - Control Flow

Permitted settings for this option are:

- **52. [R] There shall be no unreachable code** (`--saferc=52`)
- **53. [R] All non-null statements shall have a side-effect** (`--saferc=53`)
- **54. [R] Null statement must occur on line by itself** (`--saferc=54`)
- **55. [A] No labels except in 'switch' statements** (`--saferc=55`)
- **56. [R] The 'goto' statement shall not be used** (`--saferc=56`)
- **57. [R] The 'continue' statement shall not be used** (`--saferc=57`)
- **58. [R] No 'break' statement except in 'switch'** (`--saferc=58`)
- **59. [R] Dependent statements always enclosed in braces** (`--saferc=59`)
- **60. [A] All 'if' and 'else if' constructs must have an 'else'** (`--saferc=60`)
- **61. [R] Every non-empty 'case' clause terminated with 'break'** (`--saferc=61`)
- **62. [R] All 'switch' statements should contain a 'default' clause** (`--saferc=62`)
- **63. [A] No Boolean values in 'switch' expressions** (`--saferc=63`)
- **64. [R] 'switch' statements need at least one 'case'** (`--saferc=64`)
- **65. [R] No floating-point variables in loop counters** (`--saferc=65`)

- **66. [A] Only loop control expression in `for' statement header (--saferc=66)**
— Not enforced.
- **67. [A] No modification of control variables in `for' loop body (--saferc=67)**
— Not enforced.

MISRA C 1998 Rules - Functions

Permitted settings for this option are:

- **68. [R] Functions always declared at file scope (--saferc=68)**
- **69. [R] No variable argument functions (--saferc=69)**
- **70. [R] No direct or indirect recursion (--saferc=70)**
- **71. [R] Functions always have prototype declarations (--saferc=71)** — 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 255).
- **72. [R] Function declaration and definition match prototypes (--saferc=72)**
— Always enforced.
- **73. [R] Identifiers given for all or none of function parameters (--saferc=73)**
- **74. [R] Declaration and definition match parameter names (--saferc=74)**
- **75. [R] Every function shall have an explicit return type (--saferc=75)**
- **76. [R] Functions with no parameters should have `void' parameter (--saferc=76)**
- **77. [R] Unqualified arguments of callee and caller compatible (--saferc=77)**
— Not enforced.
- **78. [R] Number of arguments matches the function prototype (--saferc=78)**
— Always enforced.
- **79. [R] Values returned by `void' functions not used (--saferc=79)** — Always enforced.
- **80. [R] No void expressions passed as parameters (--saferc=80)** — Always enforced.
- **81. [A] `const' present when needed on reference parameters (--saferc=81)**
— Not enforced.

- **82. [A] A function should have a single point of exit (--saferc=82)**
- **83. [R] Return expression matches function type (--saferc=83)**
- **84. [R] No expressions returned from 'void' functions (--saferc=84) — Always enforced.**
- **85. [A] Functions called with empty () if no parameters (--saferc=85)**
- **86. [A] If error information returned, it should be tested (--saferc=86) — Not enforced.**

MISRA C 1998 Rules - Preprocessor

Permitted settings for this option are:

- **87. [R] Only preprocessing directives before #include (--saferc=87)**
- **88. [R] Only standard characters in file names for #include (--saferc=88)**
- **89. [R] #include directive followed by <filename> or 'filename' (--saferc=89) — Always enforced.**
- **90. [R] C macros only as constant, function-like, or specifier/qualifier (--saferc=90)**
- **91. [R] Macros not #define'd or #undef'd within a block (--saferc=91)**
- **92. [A] #undef should not be used (--saferc=92)**
- **93. [A] Function used instead of function-like macro when possible (--saferc=93) — Not enforced.**
- **94. [R] Function-like macro called with all of its arguments (--saferc=94) — Always enforced.**
- **95. [R] No preprocessing directives in function-like macros (--saferc=95)**
- **96. [R] Use of parentheses in function-like macros (--saferc=96)**
- **97. [A] Identifiers in pre-processing directives defined before used (--saferc=97)**
- **98. [R] Only one use of '#' or '##' operator in any one macro**
- **99. [R] Uses of #pragmas documented and explained (--saferc=99) — This book lists all #pragma constructs, their usage and effects (see “Pragma Directives” on page 747).**

- **100. [R] Correct use of 'defined' preprocessor operator (--saferc=100) —**
Always enforced.

MISRA C 1998 Rules - Pointers and Arrays

Permitted settings for this option are:

- **101. [A] Pointer arithmetic should not be used (--saferc=101)**
- **102. [A] No more than 2 levels of pointer indirection (--saferc=102)**
- **103. [R] Relational operators not used except on same object (--saferc=103) —**
Not enforced.
- **104. [R] No non-constant pointers to functions (--saferc=104)**
- **105. [R] Function and a pointer to it match in prototypes (--saferc=105)**
- **106. [R] Address of automatic variable not used out of scope (--saferc=106) —**
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 (--saferc=107) —**
Performed via a run-time check (**-check=nilderef**). For more information about run-time error checking, see “Run-Time Error Checks” on page 135. The option **--no_misra_runtime** disables both this rule and rule 4, as well as the run-time component of rule 38.

MISRA C 1998 Rules - Structs and Unions

Permitted settings for this option are:

- **108. [R] All members of structures/union fully specified (--saferc=108) —**
Not enforced.
- **109. [R] Overlapping variable storage should not be used (--saferc=109) —**
Not enforced.
- **110. [R] Unions not used to access sub-parts of larger data types (--saferc=110) —**
Not enforced.
- **111. [R] Bit fields can only have 'unsigned int' or 'signed int' types (--saferc=111)**

- **112. [R] Bit fields of 'signed int' type must be at least 2 bits long**
(--saferc=112)
- **113. [R] All members can be accessed only through their name**
(--saferc=113)

MISRA C 1998 Rules - Standard Library

Permitted settings for this option are:

- **114. [R] No redefinition of reserved words and standard library names**
(--saferc=114) — 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** (--saferc=115)
— Not enforced.
- **116. [R] All library production code conforms to MISRA** (--saferc=116)
— Not enforced.
- **117. [R] Check validity of value passed to library functions** (--saferc=117)
— Not enforced.
- **118. [R] Dynamic heap memory allocation shall not be used** (--saferc=118)
- **119. [R] The error indicator 'errno' shall not be used** (--saferc=119)
- **120. [R] Macro 'offsetof' in <stddef.h> shall not be used** (--saferc=120)
- **121. [R] <locale.h> and 'setlocale' function shall not be used** (--saferc=121)
- **122. [R] The 'setjmp' and 'longjmp' facilities shall not be used**
(--saferc=122)
- **123. [R] The facilities of <signal.h> shall not be used** (--saferc=123)
- **124. [R] No use of <stdio.h> in production code** (--saferc=124)
- **125. [R] Functions 'atof', 'atoi', and 'atol' not used** (--saferc=125)
- **126. [R] Functions 'abort', 'exit', 'getenv', and 'system' not used**
(--saferc=126)
- **127. [R] The facilities of <time.h> shall not be used** (--saferc=127)

MISRA C 2004

This option category is contained within **C/C++ Compiler**.

Specifies the use of compliance checks for the 2004 Motor Industry Software Reliability Association (MISRA) standard. MISRA 2004 is a set of C guidelines designed to enforce good engineering practices when developing embedded automotive systems. For more information, see “MISRA C 2004” on page 721.

To reference the diagnostic numbers listed below, see *MULTI: Toolchain Error Messages and Other Diagnostics*.



Note

MISRA options function as an added layer of checking on top of C90- or C99-conformant code. Consequently, generic compiler errors may supersede MISRA-specific diagnostics, regardless of whether the code violates any MISRA guidelines.

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 a new-line character.
 - Open or close 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 is dependent on the hardware configuration of the target. Most Green Hills compilers, however, round the division result toward 0.
- **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 748).
- **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 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 explicit 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`.

- **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.



Note

To reflect the spirit of the rule, this option will also check for casts that remove qualifiers for C extensions (such as `__bterversed` and `__packed`).

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`) — Enforced.

Associated diagnostics include 2017 and 2018.

- **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.** This option will validate that an `else` statement has a statement in it. If it does not, the MISRA 2004 standard allows for comments "to ensure that a positive indication is given of the desired behaviour." To avoid triggering a diagnostic in this instance, add a statement such as `(void)0;`. If the `else` statement is missing altogether, this will only be diagnosed in the C dialects **-ansi**, **-ANSI**, and **-gcc**.

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.
- **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 135. The `--no_misra_runtime` option disables this rule.

MISRA C 2012

This option category is contained within **C/C++ Compiler**.

Specifies the use of compliance checks for the 2012 Motor Industry Software Reliability Association (MISRA) standard. MISRA 2012 is a set of C guidelines designed to enforce good engineering practices when developing embedded automotive systems. For more information, see “MISRA C 2012” on page 723. MISRA 2012 is disabled by default.

To use these guidelines effectively with the Green Hills toolchain, note the following:

- All guidelines defined in the MISRA 2012 standard as directives or undecidable rules cannot be comprehensively checked by the toolchain (or any other program). For undecidable rules, the toolchain will perform simplified solutions designed to catch common violations. These solutions will tend towards under-reporting errors to minimize reporting errors against rule-compliant code.
- The checking for some MISRA guidelines is affected by language extensions. For example, an unused label can trigger a rule 2.6 violation, but not if the label is annotated with `__attribute__((unused))`. The attribute already indicates that the label is unused, avoiding the problem mentioned in the rule's rationale. The interaction of MISRA and language extensions is outside the scope of both MISRA and the C Standard. Regardless, the compiler will attempt to reconcile the behavior of language extensions and the goal of any given MISRA rule.

- MISRA options function as an added layer of checking on top of C90- or C99-conformant code. Consequently, generic compiler errors may supersede MISRA-specific diagnostics, regardless of whether the code violates any MISRA guidelines.
- Some MISRA 2012 options enable diagnostics that cannot have their severity altered by `--misra_mand`, `--misra_req`, or `--misra_adv`. Most of these diagnostics will not have an associated MISRA header. These diagnostics can still have their severity altered on a generalized diagnostic level, such as with `--diag_warning`. Alternatively, disabling the correct MISRA 2012 guideline will disable the associated diagnostics.
- Unless a MISRA guideline applies to preprocessing specifically, its validations will typically be performed after preprocessing.
- GNU languages modes are not compatible with MISRA C 2012.
- MISRA 2012 checking is permitted with C11 and later language modes. Note, however, that neither the MISRA 2012 standard nor its associated options account for new features added to the language post-C99.
- Enabling various MISRA options may have any of the following effects on your project:
 - Changing the language mode (as with `--misra_2012=R1.1`).
 - Disabling language extensions (as with `--misra_2012=R1.2`).
 - Changing the data type of string literals to `const char *` (as with `--misra_2012=R7.4`).
 - Enabling run-time error checking (see “MISRA C - Enforcement and Run-Time” on page 148).
 - Disabling some optimizations (see “MISRA C 2012 Guidelines - Stricter R8.7” on page 203 and “MISRA C 2012 Guidelines - Stricter R8.9” on page 203).

Furthermore, some MISRA guidelines can be enforced more or less effectively depending on the build options set for your project. For details, consult the documentation for the specific guidelines you enable.

To reference the diagnostic numbers listed below, see *MULTI: Toolchain Error Messages and Other Diagnostics*.

**Note**

When diagnostic messages are given by the toolchain, MISRA rules will be referred to as MISRA Rule 2012 X.Y, and MISRA directives will be referred to as MISRA 2012 Rule DX.Y.

MISRA C 2012 Guidelines - 1 Environment

Permitted settings for this option are:

- **1.1 [R] The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits (--misra_2012=R1.1) — Enforced.** Diagnostics issued as a side-effect of enabling this option may be issued with elevated severity. This can occur even in regions where MISRA checking is suppressed (for example, with `#pragma ghs startnomisra`). Any discretionary diagnostics can still be suppressed with **--diag_suppress**. Note that this option must be used with the strict versions of ANSI C (**-ANSI**) or C99 (**-C99**). This option will also change the default language mode from **-c99** to **-C99**.

Associated diagnostics include 667 and 1546.

- **1.2 [A] Language extensions should not be used (--misra_2012=R1.2) — Enforced.** Diagnostics issued as a side-effect of enabling this option may be issued with elevated severity. This can occur even in regions where MISRA checking is suppressed (for example, with `#pragma ghs startnomisra`). Any discretionary diagnostics can still be suppressed with **--diag_suppress**. Note that this option will disable many extensions (such as **-gcc**) and extension keywords (such as `__inline`). Additionally, this option cannot be used with **-ansi**.

Associated diagnostics include 20, 667, 1546, 2101.

- **1.3 [R] There shall be no occurrence of undefined or critical unspecified behavior — Not enforced.**

MISRA C 2012 Guidelines - 2 Unused Code

Permitted settings for this option are:

- **2.1 [R] A project shall not contain unreachable code (--misra_2012=R2.1)** — Enforced.

Associated diagnostics include 111 and 177.

- **2.2 [R] There shall be no dead code (--misra_2012=R2.2)** — Enforced.

Associated diagnostics include 174, 177, and 550.

- **2.3 [A] A project should not contain unused type declarations (--misra_2012=R2.3)** — Enforced on a per module basis. To avoid issuing false positives, the toolchain will not diagnose ostensibly unused entities declared in headers. (It is possible that such entities are used in other translation units.)

Associated diagnostics include 2102.

- **2.4 [A] A project should not contain unused tag declarations (--misra_2012=R2.4)** — Enforced on a per module basis. To avoid issuing false positives, the toolchain will not diagnose ostensibly unused entities declared in headers. (It is possible that such entities are used in other translation units.)

Associated diagnostics include 2103.

- **2.5 [A] A project should not contain unused macro declarations (--misra_2012=R2.5)** — Enforced on a per module basis. To avoid issuing false positives, the toolchain will not diagnose ostensibly unused entities declared in headers. (It is possible that such entities are used in other translation units.)

Associated diagnostics include 2104.

- **2.6 [A] A function should not contain unused label declarations (--misra_2012=R2.6)** — Enforced.

Associated diagnostics include 177.

- **2.7 [A] There should be no unused parameters in functions** (`--misra_2012=R2.7`) — Enforced. Note that this option will not diagnose unused parameters of `main`.

Associated diagnostics include 826.

MISRA C 2012 Guidelines - 3 Comments

Permitted settings for this option are:

- **3.1 [R] The character sequences `/*` and `//` shall not be used within a comment** (`--misra_2012=R3.1`) — Enforced.

Associated diagnostics include 9.

- **3.2 [R] Line-splicing shall not be used in `//` comments** (`--misra_2012=R3.2`) — Enforced.

Associated diagnostics include 2060.

MISRA C 2012 Guidelines - 4 Character Sets and Lexical Conventions

Permitted settings for this option are:

- **4.1 [R] Octal and hexadecimal escape sequences shall be terminated** (`--misra_2012=R4.1`) — Enforced. Note that this rule is only checked after the preprocessor stage, meaning only strings present after preprocessing will be checked. This validation process operates on octal and hexadecimal escape sequences in string and character literals.

Associated diagnostics include 2069.

- **4.2 [A] Trigraphs should not be used** (`--misra_2012=R4.2`) — Enforced.

Associated diagnostics include 1695.

MISRA C 2012 Guidelines - 5 Identifiers

Permitted settings for this option are:

- **5.1 [R] External identifiers shall be distinct (--misra_2012=R5.1)** — Enforced. Because there is no significant character limit on identifiers, rule 5.1 will always be true.
- **5.2 [R] Identifiers declared in the same scope and name space shall be distinct (--misra_2012=R5.2)** — Enforced. Because there is no significant character limit on identifiers, rule 5.2 will always be true.
- **5.3 [R] An identifier declared in an inner scope shall not hide an identifier declared in an outer scope (--misra_2012=R5.3)** — Enforced. Provides a check for unique identifiers in different scopes and name spaces. This check does not consider significant characters.

Associated diagnostics include 1721.

- **5.4 [R] Macro identifiers shall be distinct (--misra_2012=R5.4)** — Enforced. Because there is no significant character limit on identifiers, rule 5.4 will be always be true.
- **5.5 [R] Identifiers shall be distinct from macro names (--misra_2012=R5.5)** — Enforced. Provides a check for unique identifiers in different scopes and name spaces. This check does not consider significant characters.

Associated diagnostics include 2061.

- **5.6 [R] A typedef name shall be a unique identifier (--misra_2012=R5.6)** — Enforced. Only names mentioned in the translation unit are evaluated.

Associated diagnostics include 1841.

- **5.7 [R] A tag name shall be a unique identifier (--misra_2012=R5.7)** — Enforced. Only names mentioned in the translation unit are evaluated.

Associated diagnostics include 1842 and 469.

- **5.8 [R] Identifiers that define objects or functions with external linkage shall be unique (--misra_2012=R5.8)** — Enforced. Only names mentioned in the translation unit are evaluated.

Associated diagnostics include 1843.

- **5.9 [A] Identifiers that define objects or functions with internal linkage should be unique (--misra_2012=R5.9)** — Enforced. Only names mentioned in the translation unit are evaluated.

Associated diagnostics include 1843.

MISRA C 2012 Guidelines - 6 Types

Permitted settings for this option are:

- **6.1 [R] Bit-fields shall only be declared with an appropriate type (--misra_2012=R6.1)** — Enforced.

Associated diagnostics include 1717 and 230.

- **6.2 [R] Single-bit named bit fields shall not be of a signed type (--misra_2012=R6.2)** — Enforced. Note that the compiler will convert 1-bit plain-typed bitfields to unsigned. Consequently, code like the following will not trigger a diagnostic:

```
struct S {  
    int i:1;  
}
```

Associated diagnostics include 108.

MISRA C 2012 Guidelines - 7 Literals and Constants

Permitted settings for this option are:

- **7.1 [R] Octal constants shall not be used (--misra_2012=R7.1)** — Enforced.

Associated diagnostics include 1718.

- **7.2 [R] A 'u' or 'U' suffix shall be applied to all integer constants that are represented in an unsigned type (--misra_2012=R7.2)** — Enforced.

Associated diagnostics include 1775.

- **7.3 [R] The lowercase character 'l' shall not be used in a literal suffix (--misra_2012=R7.3)** — Enforced.

Associated diagnostics include 2063.

- **7.4 [R] A string literal shall not be assigned to an object unless the object's type is 'pointer to const-qualified char' (--misra_2012=R7.4) — Enforced.** This option is enforced by changing the type of string literals to `const char *`. Because this option modifies the underlying type of string literals, it may affect executable size. To disable this option, pass either **--misra_2012=-R7.4** or **--no_misra_runtime**. This option's diagnostic output will not include the MISRA 2012 rule 7.4 header.

Associated diagnostics include 120, 137, 144, 167, 177, 513, 550, and 1812.

MISRA C 2012 Guidelines - 8 Declarations and Definitions

Permitted settings for this option are:

- **8.1 [R] Types shall be explicitly specified (--misra_2012=R8.1) — Enforced.**

Associated diagnostics include 77, 260, 837, 1051, and 938.

- **8.2 [R] Function types shall be in prototype form with named parameters (--misra_2012=R8.2) — Enforced.** Note that this option cannot be used with **--prototype_silent** or **--prototype_warnings**.

Associated diagnostics include 1727, 1728, 223, and 2096.

- **8.3 [R] All declarations of an object or function shall use the same names and type qualifiers (--misra_2012=R8.3) — Enforced.** To improve coverage for this rule, consider passing **-argcheck** and **-globalcheck**. This will provide a more thorough evaluation of whether types of functions and objects match across different translation units.

Associated diagnostics include 147, 1745, and 1844.

- **8.4 [R] A compatible declaration shall be visible when an object or function with external linkage is defined (--misra_2012=R8.4) — Enforced.**

Associated diagnostics include 2066 and 2067.

- **8.5 [R] An external object or function shall be declared once in one and only one file — Not enforced.**
- **8.6 [R] An identifier with external linkage shall have exactly one external definition (--misra_2012=R8.6) — Enforced.**

Associated diagnostics include 585.

- **8.7 [A] Functions and objects should not be defined with external linkage if they are referenced in only one translation unit (--misra_2012=R8.7) — Enforced.** This option cannot be used in a project with **-relobj** or **-relprog**.

Associated diagnostics include 584.



Note

Enabling interprocedural optimizations may inhibit the linker's ability to validate this rule. Some implicit optimizations may also interfere with this rule. To disable these implicit optimizations, see “MISRA C 2012 Guidelines - Stricter R8.7” on page 203.

- **8.8 [R] The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage (--misra_2012=R8.8) — Enforced.**

Associated diagnostics include 172.

- **8.9 [A] An object should be defined at block scope if its identifier only appears in a single function (--misra_2012=R8.9) — Enforced.**

Associated diagnostics include 437 and 582.

- **8.10 [R] An inline function shall be declared with the static storage class (--misra_2012=R8.10) — Enforced.**

Associated diagnostics include 2064.

- **8.11 [A] When an array with external linkage is declared, its size should be explicitly specified (--misra_2012=R8.11) — Enforced.**

Associated diagnostics include 1824.

- **8.12 [R] Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique (--misra_2012=R8.12) — Enforced.**

Associated diagnostics include 2072.

- **8.13 [A] A pointer should point to a const-qualified type whenever possible — Not enforced.**
- **8.14 [R] The restrict type qualifier shall not be used (--misra_2012=R8.14) — Enforced.**

Associated diagnostics include 643.

MISRA C 2012 Guidelines - 9 Initialization

Permitted settings for this option are:

- **9.1 [M] The value of an object with automatic storage duration shall not be read before it has been set (--misra_2012=R9.1)** — Enforced. Note that rule 9.1 is undecidable. Additionally, checks will not be performed across function boundaries, and complex control flow (such as through the use of `gotos`) may also defeat this option's validation process.

Associated diagnostics include 549.

- **9.2 [R] The initializer for an aggregate or union shall be enclosed in braces (--misra_2012=R9.2)** — Enforced.

Associated diagnostics include 146, 1736, and 991.

- **9.3 [R] Arrays shall not be partially initialized (--misra_2012=R9.3)** — Enforced.

Associated diagnostics include 2089.



Note

This rule is checked late in compilation during a special phase that is precluded by previous errors. As such, you may not see diagnostics for this rule if there are other errors in the translation unit. You must resolve these other errors to properly diagnose this rule.

- **9.4 [R] An element of an object shall not be initialized more than once (--misra_2012=R9.4)** — Enforced.

Associated diagnostics include 2075.



Note

This rule is checked late in compilation during a special phase that is precluded by previous errors. As such, you may not see diagnostics for this rule if there are other errors in the translation unit. You must resolve these other errors to properly diagnose this rule.

- **9.5 [R] Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly (--misra_2012=R9.5) — Enforced.**

Associated diagnostics include 2074.

MISRA C 2012 Guidelines - 10 The Essential Type Model



Note

As an extension, integer literals with a value of 0 or 1 are allowed in contexts that require an essentially Boolean type.

Permitted settings for this option are:

- **10.1 [R] Operands shall not be of an inappropriate essential type (--misra_2012=R10.1) — Enforced.**

Associated diagnostics include 2099.

- **10.2 [R] Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations (--misra_2012=R10.2) — Enforced.**

Associated diagnostics include 2098.

- **10.3 [R] The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category (--misra_2012=R10.3) — Enforced.**

Associated diagnostics include 1864, 1869, 2091, 1868, 68, and 69.

- **10.4 [R] Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category (--misra_2012=R10.4) — Enforced.**

Associated diagnostics include 1863, 1865, 1866, and 2100.

- **10.5 [A] The value of an expression should not be cast to an inappropriate essential type (--misra_2012=R10.5) — Enforced.**

Associated diagnostics include 2086.

- **10.6 [R] The value of a composite expression shall not be assigned to an object with wider essential type (--misra_2012=R10.6)** — Enforced. Note that validation of this rule is only performed on operations with operands of the same essential type category (width does not translate to different essential type categories). Ensuring that operands are of the same essential type category is a MISRA requirement for other guidelines, such as rule 10.4.

Associated diagnostics include 2092.

- **10.7 [R] If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type (--misra_2012=R10.7)** — Enforced. Note that validation of this rule is only performed on operations with operands of the same essential type category (width does not translate to different essential type categories). Ensuring that operands are of the same essential type category is a MISRA requirement for other guidelines, such as rule 10.4.

Associated diagnostics include 2105.

- **10.8 [R] The value of a composite expression shall not be cast to a different essential type category or a wider essential type (--misra_2012=R10.8)** — Enforced.

Associated diagnostics include 2086 and 2108.

MISRA C 2012 Guidelines - 11 Pointer Type Conversions

Permitted settings for this option are:

- **11.1 [R] Conversions shall not be performed between a pointer to a function and any other type (--misra_2012=R11.1)** — Enforced.

Associated diagnostics include 1832.

- **11.2 [R] Conversions shall not be performed between a pointer to an incomplete type and any other type (--misra_2012=R11.2)** — Enforced.

Associated diagnostics include 2077.

- **11.3 [R] A cast shall not be performed between a pointer to object type and a pointer to a different object type (--misra_2012=R11.3)** — Enforced.

Associated diagnostics include 1833.

- **11.4 [A] A conversion should not be performed between a pointer to object and an integer type (--misra_2012=R11.4) — Enforced.**

Associated diagnostics include 1834.

- **11.5 [A] A conversion should not be performed from pointer to void into pointer to object (--misra_2012=R11.5) — Enforced.**

Associated diagnostics include 1833.

- **11.6 [R] A cast shall not be performed between pointer to void and an arithmetic type (--misra_2012=R11.6) — Enforced.**

Associated diagnostics include 2041.

- **11.7 [R] A cast shall not be performed between pointer to object and a non-integer arithmetic type (--misra_2012=R11.7) — Enforced.**

Associated diagnostics include 1833.

- **11.8 [R] A cast shall not remove any const or volatile qualification from the type pointed to by a pointer (--misra_2012=R11.8) — Enforced.**

Associated diagnostics include 1836.



Note

To reflect the spirit of the rule, this option will also check for casts that remove qualifiers for C extensions (such as `__bayerversed` and `__packed`).

- **11.9 [R] The macro NULL shall be the only permitted form of integer null pointer constant (--misra_2012=R11.9) — Enforced.**

Associated diagnostics include 2084.

MISRA C 2012 Guidelines - 12 Expressions

Permitted settings for this option are:

- **12.1 [A] The precedence of operators within expressions should be made explicit (--misra_2012=R12.1) — Enforced.**

Associated diagnostics include 1737.

- **12.2 [R] The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand (--misra_2012=R12.2)** — Enforced. Run-time error checking is performed to confirm that run-time shifts lie in the range required by rule 12.2. Some versions of the MULTI debugger may output a diagnostic with the following string: `Unknown error on line X of function Y in file.` Passing `--no_misra_runtime` will disable this option's run-time validation.

Associated diagnostics include 1986, 62, and 63.

- **12.3 [A] The comma operator should not be used (--misra_2012=R12.3)** — Enforced.

Associated diagnostics include 1825.

- **12.4 [A] Evaluation of constant expressions should not lead to unsigned integer wrap-around (--misra_2012=R12.4)** — Enforced. This option will also check for overflows on non-evaluated constant expressions.

Associated diagnostics include 1735.

- **12.5 [M] The sizeof operator shall not have an operand which is a function parameter declared as 'array of type' (--misra_2012=R12.5)** — Enforced.

Associated diagnostics include 1904.

MISRA C 2012 Guidelines - 13 Side Effects

Permitted settings for this option are:

- **13.1 [R] Initializer lists shall not contain persistent side effects (--misra_2012=R13.1)** — Enforced. Any function call that is not a call to a safe routine is considered to have a persistent side effect. Consider the use of `__attribute__((const))` and `__attribute__((pure))`.

Associated diagnostics include 2087.

- **13.2 [R] The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders (--misra_2012=R13.2)** — Enforced. Note that this option does not check for side-effects across function

boundaries. Additionally, checking for this rule may not catch all reads or writes of volatile objects.

Associated diagnostics include 2017, 2018, 2106, and 2107 .

- **13.3 [A] A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator (--misra_2012=R13.3)** — Enforced. This option treats the initialization of a variable as a side-effect within its initializer expression. Thus, using the increment or decrement operator in an initializer will trigger a violation. This is done to reflect the presumed intent of the MISRA standard.

Associated diagnostics include 2080.

- **13.4 [A] The result of an assignment operator should not be used (--misra_2012=R13.4)** — Enforced.

Associated diagnostics include 2076.

- **13.5 [R] The right hand operand of a logical && or || operator shall not contain persistent side effects (--misra_2012=R13.5)** — Enforced.

Associated diagnostics include 1724.

- **13.6 [M] The operand of the sizeof operator shall not contain any expression which has potential side effects (--misra_2012=R13.6)** — Enforced. Note that this option will not diagnose apparent side effects from the dimension expression of function parameters declared with array type. Such expressions are never evaluated (regardless of context), and are therefore never real side effects.

Associated diagnostics include 1725.

MISRA C 2012 Guidelines - 14 Control Statement Expressions

Permitted settings for this option are:

- **14.1 [R] A loop counter shall not have essentially floating type (--misra_2012=R14.1)** — Enforced. Note that this option will flag any use of floating-point variables in loop conditions. You may need to suppress the error in cases where they are not being used as counters.

Associated diagnostics include 1750.

- **14.2 [R] A for loop shall be well-formed** — Not enforced.
- **14.3 [R] Controlling expressions shall not be invariant** (`--misra_2012=R14.3`) — Enforced. This option's rule enforcement is limited: only trivially invariant controlling expressions are caught. Further, certain constant expressions that create invariant statements (such as `if (x = 1)`) will not be identified. The presence of side-effects in the controlling expression also hinders the checking for this rule. This option will diagnose invariantly true loop conditions even though the MISRA standard treats infinite loops as an explicit exception to this rule.

Associated diagnostics include 236 and 237.

- **14.4 [R] The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type** (`--misra_2012=R14.4`) — Enforced. This option additionally enforces that the first operand of a conditional operator (`?:`) is essential Boolean. This is not strictly required by the MISRA standard, but is assumed to reflect its intent. Note also that the validation of rule 14.4 will allow integer literals with a value of 0 or 1 in contexts that require an essentially Boolean type.

Associated diagnostics include 1855 and 1881.

MISRA C 2012 Guidelines - 15 Control Flow

Permitted settings for this option are:

- **15.1 [A] The goto statement should not be used** (`--misra_2012=R15.1`) — Enforced.

Associated diagnostics include 1705.

- **15.2 [R] The goto statement shall jump to a label declared later in the same function** (`--misra_2012=R15.2`) — Enforced.

Associated diagnostics include 2073.

- **15.3 [R] Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement** (`--misra_2012=R15.3`) — Enforced.

Associated diagnostics include 2088.

- **15.4 [A] There should be no more than one break or goto statement used to terminate any iteration statement (--misra_2012=R15.4) — Enforced.**

Associated diagnostics include 2097.

- **15.5 [A] A function should have a single point of exit at the end (--misra_2012=R15.5) — Enforced.**

Associated diagnostics include 1734.

- **15.6 [R] The body of an iteration-statement or a selection-statement shall be a compound-statement (--misra_2012=R15.6) — Enforced.**

Associated diagnostics include 1709 and 1826.

- **15.7 [R] All if ... else if constructs shall be terminated with an else statement (--misra_2012=R15.7) — Enforced.** This option will validate that an `else` statement has a statement in it. If it does not, the MISRA 2012 standard allows for comments "to ensure that a positive indication is given of the desired behaviour." To avoid triggering a diagnostic in this instance, add a statement such as `(void)0;`.

Associated diagnostics include 1710.

MISRA C 2012 Guidelines - 16 Switch Statements

Permitted settings for this option are:

- **16.1 [R] All switch statements shall be well-formed (--misra_2012=R16.1) — Enforced.** Although the grammar of rule 16.1 requires that a switch-clause be terminated with a `break` statement, this option's validation will also allow a switch-clause terminated with a `return`, `goto`, or `continue` statement. Note that this extension does not apply to the validation of rule 16.3, which will still require the use of a `break` statement.

Associated diagnostics include 1709, 1711, and 1838.

- **16.2 [R] A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement (--misra_2012=R16.2) — Enforced.**

Associated diagnostics include 1838.

- **16.3 [R] An unconditional break statement shall terminate every switch-clause (`--misra_2012=R16.3`)** — Enforced.

Associated diagnostics include 1711 and 1884.

- **16.4 [R] Every switch statement shall have a default label (`--misra_2012=R16.4`)** — Enforced. This option will validate that a default case has a statement in it. If it does not, the MISRA 2012 standard notes that comments "can be used to explain why no specific action has been taken." To avoid triggering a diagnostic in this instance, add a statement such as `(void)0;`.

Associated diagnostics include 1712.

- **16.5 [R] A default label shall appear as either the first or the last switch label of a switch statement (`--misra_2012=R16.5`)** — Enforced.

Associated diagnostics include 2040.

- **16.6 [R] Every switch statement shall have at least two switch-clauses (`--misra_2012=R16.6`)** — Enforced.

Associated diagnostics include 1944.

- **16.7 [R] A switch-expression shall not have essentially Boolean type (`--misra_2012=R16.7`)** — Enforced.

Associated diagnostics include 1733.

MISRA C 2012 Guidelines - 17 Functions

Permitted settings for this option are:

- **17.1 [R] The features of `<stdarg.h>` shall not be used (`--misra_2012=R17.1`)** — Enforced.

Associated diagnostics include 2055 and 2068.

- **17.2 [R] Functions shall not call themselves, either directly or indirectly (`--misra_2012=R17.2`)** — Enforced. Although this option enables some checks for recursion, consider compiling your program with debugging symbols and using **gstack** for a more thorough test. For example:

```
gstack -entry_point main a.out
```

If the output provided by **gstack** lists any function names in the "Recursive Clusters" section, your program has a MISRA 2012 rule 17.2 violation.

Associated diagnostics include 580.

- **17.3 [M] A function shall not be declared implicitly (--misra_2012=R17.3)** — Enforced. Note that this option cannot be used with **--prototype_silent** or **--prototype_warnings**.

Associated diagnostics include 223.

- **17.4 [M] All exit paths from a function with non-void return type shall have an explicit return statement with an expression (--misra_2012=R17.4)** — Enforced.

Associated diagnostics include 117 and 940.

- **17.5 [A] The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements (--misra_2012=R17.5)** — Enforced. This option will check for straightforward cases where an array (including the decayed array corresponding to a parameter) is passed along as an argument corresponding to an array parameter. This option will not check for cases where a pointer is passed along as an argument to an array parameter, nor will it diagnose code like the following:

```
void f(int arr[15]);  
void f2(int arr[]) { f(arr); }
```

Associated diagnostics include 2094 and 2095.

- **17.6 [M] The declaration of an array parameter shall not contain the static keyword between the [] (--misra_2012=R17.6)** — Enforced.

Associated diagnostics include 2070.

- **17.7 [R] The value returned by a function having non-void return type shall be used (--misra_2012=R17.7)** — Enforced.

Associated diagnostics include 2071.

- **17.8 [A] A function parameter should not be modified (--misra_2012=R17.8)** — Enforced. This option acts by detecting direct writes to a function parameter.

It will also diagnose cases where the address of a function parameter is taken and not used as a pointer to `const`. For example:

```
void foo(int i)
{
    const int *p1 = &i; /* OK */
    int *p2      = &i; /* Not OK */
    ...
}
```

Associated diagnostics include 2078 and 2079.

MISRA C 2012 Guidelines - 18 Pointers and Arrays

Permitted settings for this option are:

- **18.1 [R] A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand** — Not enforced.
- **18.2 [R] Subtraction between pointers shall only be applied to pointers that address elements of the same array** — Not enforced.
- **18.3 [R] The relational operators `>`, `>=`, `<` and `<=` shall not be applied to objects of pointer type except where they point into the same object** — Not enforced.
- **18.4 [A] The `+`, `-`, `+=` and `-=` operators should not be applied to an expression of pointer type (`--misra_2012=R18.4`)** — Enforced.

Associated diagnostics include 1752.

- **18.5 [A] Declarations should contain no more than two levels of pointer nesting (`--misra_2012=R18.5`)** — Enforced.

Associated diagnostics include 1741.

- **18.6 [R] The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist (`--misra_2012=R18.6`)** — Enforced. This option will not check across function boundaries.

Associated diagnostics include 1056 and 1780.

- **18.7 [R] Flexible array members shall not be declared (`--misra_2012=R18.7`)**
— Enforced.

Associated diagnostics include 1029.

- **18.8 [R] Variable-length array types shall not be used (`--misra_2012=R18.8`)**
— Enforced.

Associated diagnostics include 890 and 975.

MISRA C 2012 Guidelines - 19 Overlapping Storage

Permitted settings for this option are:

- **19.1 [M] An object shall not be assigned or copied to an overlapping object**
— Not enforced.
- **19.2 [A] The union keyword should not be used (`--misra_2012=R19.2`)** — Enforced.

Associated diagnostics include 1827.

MISRA C 2012 Guidelines - 20 Preprocessing Directives

Permitted settings for this option are:

- **20.1 [A] `#include` directives should only be preceded by preprocessor directives or comments (`--misra_2012=R20.1`)** — Enforced. Note that, for files preceded by anything other than preprocessor directives or comments, diagnostics will only be issued for the initial `#include` directive.

Associated diagnostics include 1748.

- **20.2 [R] The apostrophe, comma, double quotation, or backslash characters and the `/*` or `//` character sequences shall not occur in a header file name (`--misra_2012=R20.2`)** — Enforced.

Associated diagnostics include 1747.

- **20.3 [R] The `#include` directive shall be followed by either a `<filename>` or `"filename"` sequence (`--misra_2012=R20.3`)** — Enforced.

Associated diagnostics include 14.

- **20.4 [R] A macro shall not be defined with the same name as a keyword** (`--misra_2012=R20.4`) — Enforced. Enabling this option will catch any use of `#define` or `#undef` that shares the name of a keyword.

Associated diagnostics include 1888.

- **20.5 [A] #undef should not be used** (`--misra_2012=R20.5`) — Enforced.

Associated diagnostics include 1715.

- **20.6 [R] Tokens that look like a preprocessing directive shall not occur within a macro argument** (`--misra_2012=R20.6`) — Enforced.

Associated diagnostics include 10.

- **20.7 [R] Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses** (`--misra_2012=R20.7`) — Enforced. Note that the toolchain's enforcement of this rule is stricter than required by MISRA: the toolchain mandates that all macro parameters are enclosed in parentheses (this is checked during the definition of a macro). Specific invocations of macros are not taken into account, thus the toolchain does not honor MISRA's exceptions for non-expression arguments and for arguments with embedded parentheses.

Associated diagnostics include 1829.

- **20.8 [R] The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1** (`--misra_2012=R20.8`) — Enforced.

Associated diagnostics include 2065.

- **20.9 [R] All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation** (`--misra_2012=R20.9`) — Enforced.

Associated diagnostics include 193.

- **20.10 [A] The # and ## preprocessor operators should not be used** (`--misra_2012=R20.10`) — Enforced.

Associated diagnostics include 1830.

- **20.11 [R] A macro parameter immediately following a # operator shall not immediately be followed by a ## operator (--misra_2012=R20.11) — Enforced.**

Associated diagnostics include 2054.

- **20.12 [R] A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators (--misra_2012=R20.12) — Enforced.**

Associated diagnostics include 2093.

- **20.13 [R] A line whose first token is # shall be a valid preprocessing directive (--misra_2012=R20.13) — Enforced.**

Associated diagnostics include 11 and 14.

- **20.14 [R] All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related (--misra_2012=R20.14) — Enforced.**

Associated diagnostics include 36 and 37.

MISRA C 2012 Guidelines - 21 Standard Libraries

Permitted settings for this option are:

- **21.1 [R] #define and #undef shall not be used on a reserved identifier or reserved macro name (--misra_2012=R21.1) — Enforced.** Note that this option will also enforce the exclusion of keywords from #define and #undef.

Associated diagnostics include 45, 46, 1885, 1861, and 1888.

- **21.2 [R] A reserved identifier or macro name shall not be declared — Not enforced.**
- **21.3 [R] The memory allocation and deallocation functions of <stdlib.h> shall not be used (--misra_2012=R21.3) — Enforced.**

Associated diagnostics include 2055 and 2056.

- **21.4 [R] The standard header file <setjmp.h> shall not be used (--misra_2012=R21.4) — Enforced.**

Associated diagnostics include 1906, 2042, and 2043.

- **21.5 [R] The standard header file <signal.h> shall not be used** (`--misra_2012=R21.5`) — Enforced.

Associated diagnostics include 2042.

- **21.6 [R] The Standard Library input/output functions shall not be used** (`--misra_2012=R21.6`) — Enforced. Note that this option is stricter than required by MISRA, as it forbids the use of **stdio.h** entirely.

Associated diagnostics include 2042, 2055, and 2056.

- **21.7 [R] The atof, atoi, atol and atoll functions of <stdlib.h> shall not be used** (`--misra_2012=R21.7`) — Enforced.

Associated diagnostics include 1907 and 2055.

- **21.8 [R] The library functions abort, exit, and system of <stdlib.h> shall not be used** (`--misra_2012=R21.8`) — Enforced.

Associated diagnostics include 2055 and 2056.

- **21.9 [R] The library functions bsearch and qsort of <stdlib.h> shall not be used** (`--misra_2012=R21.9`) — Enforced.

Associated diagnostics include 2055 and 2056.

- **21.10 [R] The Standard Library time and date functions shall not be used** (`--misra_2012=R21.10`) — Enforced. Note that this option is stricter than required by MISRA, as it forbids the use of **time.h** entirely.

Associated diagnostics include 2042, 2055, and 2056.

- **21.11 [R] The standard header file <tgmath.h> shall not be used** (`--misra_2012=R21.11`) — Enforced.

Associated diagnostics include 2042.

- **21.12 [A] The exception handling features of <fenv.h> should not be used** (`--misra_2012=R21.12`) — Enforced.

Associated diagnostics include 561 and 2055.

- **21.13 [M] Any value passed to a function in <ctype.h> shall be representable as an unsigned char or be the value EOF** — Not enforced. However, when using the Green Hills standard library, this behavior is defined.

The standard library understands a **ctype.h** function to be one that takes an `int` and checks if it is in legal range. If not, it returns the same output as its input value.

- **21.14 [R] The Standard Library function `memcmp` shall not be used to compare null terminated strings (`--misra_2012=R21.14`)** — Enforced. This option will only check for rule violations involving string literals.

Associated diagnostics include 2081.

- **21.15 [R] The pointer arguments to the Standard Library functions `memcpy`, `memmove` and `memcmp` shall be pointers to qualified or unqualified versions of compatible types (`--misra_2012=R21.15`)** — Enforced.

Associated diagnostics include 2082.

- **21.16 [R] The pointer arguments to the Standard Library function `memcmp` shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type (`--misra_2012=R21.16`)** — Enforced.

Associated diagnostics include 2083.

- **21.17 [M] Use of the string handling functions from `<string.h>` shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters** — Not enforced.
- **21.18 [M] The `size_t` argument passed to any function in `<string.h>` shall have an appropriate value** — Not enforced.
- **21.19 [M] The pointers returned by the Standard Library functions `localeconv`, `getenv`, `setlocale` or `strerror` shall only be used as if they have pointer to const-qualified type** — Not enforced.
- **21.20 [M] The pointer returned by the Standard Library functions `asctime`, `ctime`, `gmtime`, `localtime`, `localeconv`, `getenv`, `setlocale` or `strerror` shall not be used following a subsequent call to the same function** — Not enforced.

MISRA C 2012 Guidelines - 22 Resources

Permitted settings for this option are:

- **22.1 [R] All resources obtained dynamically by means of Standard Library functions shall be explicitly released (`--misra_2012=R22.1`)** — Enforced. This option provides a mechanism that can be used to find dynamic memory leaks. It does not allow checking for leaks of file handles allocated via `fopen`. This option implies `-check=alloc` and thus is not supported in environments where `-check=alloc` is also not supported. Note that detecting memory leaks via `-check=alloc` requires additional steps. For more information, see “Enabling General Memory Allocation Checking” on page 52. Passing `--no_misra_runtime` will disable this option.
- **22.2 [M] A block of memory shall only be freed if it was allocated by means of a Standard Library function (`--misra_2012=R22.2`)** — Enforced. Provides a run-time check for attempts to free memory that was not allocated by a Standard Library function. This option implies `-check=alloc` and thus is not supported in environments where `-check=alloc` is also not supported. Passing `--no_misra_runtime` will disable this option.



Note

Diagnostics associated with this option will not identify rule 22.2. For example:

```
Attempt to free something already free adr=0x4008390
Attempt to free something not allocated adr=0x4008390
```

- **22.3 [R] The same file shall not be open for read and write access at the same time on different streams** — Not enforced.
- **22.4 [M] There shall be no attempt to write to a stream which has been opened as read-only** — Not enforced. However, when using the Green Hills standard library, this behavior is defined. If the standard library detects an attempt to write to a read-only `FILE` stream, the function will return without writing to the stream. In such an instance, `errno` will be set to `EBADF`.
- **22.5 [M] A pointer to a `FILE` object shall not be dereferenced (`--misra_2012=R22.5`)** — Enforced. This option enables rule validation at compile time: the compiler will diagnose uses of the arrow (`->`) and star (`*`) operators applied to a `FILE*` value. The validation of this rule does not include

checks for indirect dereferences of `FILE` objects, such as those done via `memcpy` or `memcmp`.

Associated diagnostics include 2085.

- **22.6 [M] The value of a pointer to a `FILE` shall not be used after the associated stream has been closed** — Not enforced. However, when using the Green Hills standard library, this behavior is defined. If the standard library detects an attempt to pass a closed `FILE` stream to a `stdio.h` function, the function will return without accessing the stream. In this instance, `errno` will be set to `EBADF`.
- **22.7 [R] The macro `EOF` shall only be compared with the unmodified return value from any Standard Library function capable of returning `EOF`** — Not enforced.
- **22.8 [R] The value of `errno` shall be set to zero prior to a call to an `errno`-setting-function** — Not enforced.
- **22.9 [R] The value of `errno` shall be tested against zero after calling an `errno`-setting-function** — Not enforced.
- **22.10 [R] The value of `errno` shall only be tested when the last function to be called was an `errno`-setting-function** — Not enforced.

MISRA C 2012 Guidelines - Directives

Permitted settings for this option are:

- **D1.1 [R] Any implementation-defined behavior on which the output of the program depends shall be documented and understood** — Not enforced.
- **D2.1 [R] All source files shall compile without any compilation errors** — Not enforced.
- **D3.1 [R] All code shall be traceable to documented requirements** — Not enforced.
- **D4.1 [R] Run-time failures shall be minimized (`--misra_2012=D4.1`)** — Enforced. Directive 4.1 is implemented by enabling run-time error checking. These diagnostics are given during run-time and will not include the MISRA 2012 directive 4.1 header. Note that this option cannot be used if any of the following run-time checks are explicitly disabled: `assignbound`, `bounds`, `nilderef`, `switch`, or `zerodivide`.

Associated diagnostics include 68 and 69.

- **D4.2 [A] All usage of assembly language should be documented** — Not enforced.
- **D4.3 [R] Assembly language shall be encapsulated and isolated** (`--misra_2012=D4.3`) — Enforced.

Associated diagnostics include 1749 and 1839.

- **D4.4 [A] Sections of code should not be 'commented out'** (`--misra_2012=D4.4`) — Enforced.

Associated diagnostics include 1771.

- **D4.5 [A] Identifiers in the same name space with overlapping visibility should be typographically unambiguous** — Not enforced.
- **D4.6 [A] typedefs that indicate size and signedness should be used in place of the basic numerical types** (`--misra_2012=D4.6`) — Enforced. MISRA allows the use of `int` in the declaration of the `main` function. To avoid triggering a diagnostic on `main` declarations, use both of the following `#pragmas`:

```
#pragma ghs nowarning 1698
int main(int argc, char *argv[])
#pragma ghs endnowarning 1698
{
    ...
}
```

Associated diagnostics include 1698.

- **D4.7 [R] If a function returns error information, then that error information shall be tested** — Not enforced.
- **D4.8 [A] If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden** (`--misra_2012=D4.8`) — Enforced.

Associated diagnostics include 2090.

- **D4.9 [A] A function should be used in preference to a function-like macro where they are interchangeable** (`--misra_2012=D4.9`) — Enforced.

Associated diagnostics include 1862.

- **D4.10 [R] Precautions shall be taken in order to prevent the contents of a header file being included more than once (`--misra_2012=D4.10`)** — Enforced. Note that because `#pragma once` will not function as a `#define` guard for this directive, it should not be used.

Associated diagnostics include 1882.
- **D4.11 [R] The validity of values passed to library functions shall be checked** — Not enforced.
- **D4.12 [R] Dynamic memory allocation shall not be used (`--misra_2012=D4.12`)** — Enforced. This option validates directive 4.12 by validating rule 21.3, as the latter is a subset of the former. Because enforcing rule 21.3 is required to enforce this directive, all error messages pertaining to directive 4.12 will be issued as rule 21.3 violations. Note also that this option implies `--misra_2012=R21.3` and cannot be used when rule 21.3 is explicitly turned off.
- **D4.13 [A] Functions which are designed to provide operations on a resource should be called in an appropriate sequence** — Not enforced.
- **D4.14 [R] The validity of values received from external sources shall be checked** — Not enforced.

MISRA C 2012 Guidelines - Stricter R8.7

Controls whether to use a stricter rule validation process for MISRA C 2012 rule 8.7 (`--misra_2012=R8.7`). This option is not enabled by default as it disallows some optimization processes. For more information about rule 8.7, see “MISRA C 2012 Guidelines - 8 Declarations and Definitions” on page 182. Permitted settings for this option are:

- **On (`--misra_2012_stricter_R8_7`)**
- **Off (`--no_misra_2012_stricter_R8_7`)** — [default]

MISRA C 2012 Guidelines - Stricter R8.9

Controls whether to use a stricter rule validation process for MISRA C 2012 rule 8.9 (`--misra_2012=R8.9`). This option is not enabled by default as it disallows some optimization processes. For more information about rule 8.9, see “MISRA C 2012

Guidelines - 8 Declarations and Definitions” on page 182. Permitted settings for this option are:

- **On** (`--misra_2012_stricter_R8_9`)
- **Off** (`--no_misra_2012_stricter_R8_9`) — [default]

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`)
- **Unsigned** (`--unsigned_chars`) — [default]

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.



Note

To function correctly, this option's setting must be uniformly applied to all parts of a program.

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`) — 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`)** — [default] 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 601.

Memory Access

Specifies memory access behaviors. Permitted settings for this option are:

- **Language default memory accesses (`-memory_access=default`)** — Directs the compiler to act as if `-memory_access=efficient` were passed. When using C11/C++11 or later, this setting directs the compiler to act as if `-memory_access=safe` were passed.
- **Efficient memory accesses (`-memory_access=efficient`)** — Overaligns reads and writes of bitfields in structs to generate efficient loads and stores. Adjacent struct fields may be non-atomically read and written.
- **Thread-safe memory accesses (`-memory_access=safe`)** — Performs struct bitfield accesses in a thread-safe manner. This means avoiding the overalignment of bitfield reads and writes that can introduce race conditions when adjacent struct fields are non-atomically read and written.

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.



Note

To function correctly, this option's setting must be uniformly applied to all parts of a program.

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.

The `--no_long_long` setting is supported in ANSI C, GNU C, and Standard C++ modes. It is not supported in C99, C++11, or later language modes. It is not supported on any CPU with 64-bit registers, nor is it supported in INTEGRITY or u-velOSity. In addition, certain target-specific header files may require the `long long` type to exist.



Warning

The same libraries are used for `--long_long` and `--no_long_long`. As such, passing in the 32-bit `intmax_t` or `uintmax_t` types after using `--no_long_long` will create a mismatch with the 64-bit expectations of library functions. This is an ABI incompatibility that can lead to library functions effectively reading uninitialized values. This includes the `%j` format length modifier for `printf`. (Other format specifiers will be unaffected, as long as the type being formatted is the same in `--no_long_long` mode as in `--long_long` mode.)

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).

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 823.



Note

To function correctly, this option's setting must be uniformly applied to all parts of a program.

Alignment and Packing

This option category is contained within **C/C++ Compiler**.

Packing (Maximum Structure Alignment)



Note

Support for structure packing in C++ is limited. See the release notes for more information.

Controls the default maximum alignment of all objects of types `struct` and `class`. Permitted settings for this option are:

- **None (-pack=none)** — [default]
- **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 748).

If you specify **None**, no packing alignment is specified to the compiler; structures, classes, and their members are aligned to their alignment requirements.



Note

To function correctly, this option's setting must be uniformly applied to all parts of a program.

Misaligned Memory Access

Specifies whether the compiler may generate code that performs misaligned memory accesses. Permitted settings for this option are:

- **On (-misalign_access)** — [default] Enables the compiler to generate code that performs misaligned accesses on CPUs that support them.
- **Off (-no_misalign_access)** — Prevents the compiler from generating misaligned memory accesses outside of NEON instructions, even when they are supported by the CPU. This may cause accesses to misaligned volatile memory regions to be split into multiple instructions. Disabling misaligned accesses also disables vectorization optimizations. NEON intrinsics may still produce misaligned accesses. Note that this setting implies (and requires) **-mem_funcs=safe** (see “Memory Functions” on page 79).

C/C++ Data Allocation

This option category is contained within **C/C++ Compiler**.

These options control the treatment of data in C and C++.

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 variables 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

Controls support for variable length arrays (VLAs) when using the ISO C99 dialect (or later) as well as GNU C. Permitted settings for this option are:

- **On** (`--vla`) — [default]
- **Off** (`--no_vla`)

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 (the two-character Alternatives in the table below) in C and C++. In C++, this option also controls the recognition of operator keywords such as `and`, `or`, `bitand`, `bitor`, `and_eq`, and `or_eq`. In C, these operators are never recognized as built-in keywords, but they are provided by `#define` directives

when the **iso646.h** header file is included, regardless of whether this option is enabled. Permitted settings for this option are:

- **On** (**--alternative_tokens**) — [default] Enables support for the digraphs listed below. In C++, this also enables support for the lexical keywords listed below.
- **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 standard used for multibyte and wide characters (for example, UTF-8, Shift-JIS, EUC-JP). Multibyte and wide characters are used for encoding text outside the basic ASCII character set.

- Multibyte character encodings, such as UTF-8, are variable-width encodings. In a variable-width encoding, some characters are represented as a single byte

while other characters require multiple bytes. For example, in UTF-8, the character `A` is represented as a single byte: decimal 65 or hexadecimal `0x41`. In contrast, the division sign (`÷`) is represented as a sequence of two bytes: `0xc3, 0xb7`.

- Wide character encodings use fixed-lengths to represent characters. For example, UTF-32 uses 4-byte integers to represent Unicode characters (`A` is encoded as `0x00000041` and `÷` is encoded as `0x000000f7`). UCS-2 is a fixed length encoding that uses two bytes to represent any character in the Unicode Basic Multilingual Plane (BMP). (The BMP is the character space that describes most modern languages.) Like UTF-32, UCS-2 maps directly to the Unicode code point (the list of codes that correspond to characters). UTF-16 is an extension of UCS-2 where characters outside the BMP are encoded using a sequence of two surrogate pairs.

When using UTF-8 as a multibyte character format, wide-character encoding will be handled either by UCS-2 (when passing **`-wchar_u16`**) or UTF-32 (when passing **`-wchar_s32`**). Unlike UTF-8, Shift-JIS and EUC both serve as multibyte and wide-character encodings.



Note

This option also affects the `mbtowc()` and `wctomb()` library functions defined in **`libind.a`**, **`libutf8.a`**, and **`lib8bit.a`**.

Permitted settings for this option are:

- **UTF-8 on Host and Target (`-kanji=utf8`)** — [default for C11/C++11 or later] Specifies UTF-8 as the multibyte character encoding.
- **Shift-JIS on Host and Target (`-kanji=shiftjis`)** — [default for language modes other than C11/C++11 or later] Specifies Kanji character interpretation using the Shift-JIS format on the host and target.
- **EUC on Host and Target (`-kanji=euc`)** — Specifies Kanji character interpretation using the EUC 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 multibyte 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. Wide characters are represented as zero-extended versions of 8-bit characters.

Currently, **-kanji=utf8** is the default when using C11/C++11 or later. This is true even if the command line contains no C or C++ source files. Otherwise, **-kanji=shiftjis** is the default on all other configurations.



Note

Future versions of the Green Hills Compiler will default to **-kanji=utf8** in all configurations. If you use modes that rely on a default setting of **-kanji=shiftjis** or **-kanji=euc**, you should now explicitly pass these settings to avoid a future change in behavior.



Note

To function correctly, this option's setting must be uniformly applied to all parts of a program.

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++]

The `--no_namespaces` setting is not supported in language modes for C++11 or later.

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 in C++. 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`)



Note

The ability to disable `bool` type support is deprecated. When this option is removed in the future, `bool` type support will be obligatory.

restrict Keyword Support

Allows you to force language dialects to support the `restrict` keyword if it is not already accepted. (The `restrict` keyword is a C99 feature used to indicate non-aliased pointers.) Permitted settings for this option are:

- **On (`--restrict`)** — Acceptance of the `restrict` keyword will be universal across dialects.
- **Off (`--no_restrict`)** — [default] Acceptance of the `restrict` keyword will be limited to those dialects that already support it.

Support `__noinline` Keyword for Templates

Controls whether the `__noinline` keyword has an effect on function templates and member functions of class templates. Permitted settings for this option are:

- **On (`--enable_noinline`)**
- **Off (`--disable_noinline`)** — [default] The compiler ignores the `__noinline` keyword for member function templates and member functions of class templates.

For more information, see “Additional C++ Inlining Information and Caveats” on page 334.



Note

`__noinline` does not affect function templates and member functions of class templates that are declared as `constexpr`. These functions are implicitly inline as a result of the `constexpr` keyword and `__noinline` will not change that.

Instantiate Extern Inline

Controls the instantiation of inline functions with external linkage and inline member functions with external linkage. Permitted settings for this option are:

- **On (`--instantiate_extern_inline`)** — [default] Produces exactly one out-of-line copy in the program for each inline function that requires one. This option might increase compile time.

- **Off (`--no_instantiate_extern_inline`)** — Produces an out-of-line copy of each inline function with external linkage and each inline member function of a class with external linkage (in each module requiring such a copy). These out-of-line copies share static data but are otherwise distinct from each other. 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 “Manually Indicating Functions for Single-Pass Inlining” on page 329.

The `--no_instantiate_extern_inline` setting is not supported in language modes for C++17 or later.

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]

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()`. The `_main()` function is responsible for invoking constructors for global objects, initializing the destruction mechanism for global objects, and initializing various threading-related features. 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.

Note that destructors for the constructed objects are registered with the run-time library as each object is constructed. These destructors are called when the program exits and at the same time as functions registered with `atexit()`.

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++]



Note

When `--no_rtti` is specified, the following C++ functions are not supported:

- `std::dynamic_pointer_cast`
- `std::get_deleter`
- `std::function::target_type`
- `std::function::target`
- `locale(const locale&other, Facet*f);`

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`)

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 685.

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. The **--guiding_decls** setting is not supported in language modes for C++11 or later.

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]

The **--implicit_typename** setting is not supported in language modes for C++11 or later.

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]

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 **--STD**, **--g++**, and **--c++11/--C++11** or later]
Also implies **C/C++ Compiler**→**C++**→**Templates**→**Parse Templates in Generic Form** (**--parse_templates**). This setting implements what is sometimes called "two-stage" name lookup.
- **Off** (**--no_dep_name**) — [default for all other dialects]

The **--no_dep_name** setting is not supported in language modes for C++11 or later.

Parse Templates in Generic Form

Controls whether nonclass templates are parsed in a generic form (that is, without definite template arguments). With the exception of GNU dialects, this generic form parsing happens at the point of template definition and happens regardless of whether the template is instantiated in the program. Note that dependent name lookup (**--dep_name**)—also known as "two phase name lookup"—requires generic form parsing. Permitted settings for this option are:

- **On (--parse_templates)** — [default for modes **--STD**, **--g++**, **--c++11/--C++11** or later]
- **Off (--no_parse_templates)** — [default for all other modes]

The **--no_parse_templates** setting is not supported in language modes for C++11 or later.

This option is implied by **Dependent Name Processing** (see “Dependent Name Processing” on page 219).

Deferral of Function Template Parsing

If function templates are being parsed in their generic form (that is, **--parse_templates** is on), this option controls whether the parsing of the generic form of a template happens immediately upon the template's definition or is deferred until the end of the file. Furthermore, unused function templates will not be parsed at all (not even in their generic form). If **--no_parse_templates** is selected or defaulted to, this option has no effect. Note that this option is primarily for GNU compatibility.

Permitted settings for this option are:

- **On (--defer_parse_function_templates)** — Referenced function templates are parsed in their generic form at the end of the file. Unreferenced function templates will not be parsed at all. This option is enabled by default in the GNU C++ dialect.
- **Off (--no_defer_parse_function_templates)** — All function templates will be parsed in their generic form when their definition is seen. This option is enabled by default in all non-GNU dialects.

Max Depth for Recursive Template Instantiations

Change the maximum allowed depth for recursive template instantiations. This limit serves to detect and prevent infinitely recursing template instantiations. The equivalent driver option is:

- **--pending_instantiations=*n***

Note that, for function templates, the compiler may allow more instantiations than indicated by *n*. Additionally, if *n* is 0, no limit will be enforced, causing infinite recursions to fail when some resource is exhausted. The default value (64) should be sufficient for nearly all programs. Do not increase *n* unless your program contains a template that recursively instantiates itself (directly or indirectly) more than 64 times.

C++11

This option category is contained within **C/C++ Compiler**→**C++**.

Range-Based For Loop Support

Controls support for C++11-style range-based for loops. Permitted settings for this option are:

- **On (--range_based_for)** — [default for C++11 or later]
- **Off (--no_range_based_for)** — [default for all other dialects]

Range-based for loops are not supported when namespaces are disabled. Range-based for loops whose range expressions involve other C++11 features such as initializer lists and `rvalue` references are not supported except in language modes for C++11 or later. Additionally, the **--no_range_based_for** setting is not supported in language modes for C+11 or later.

auto Type Keyword Support

Controls the use of the `auto` keyword as a type specifier for C++11-style type deduction. Permitted settings for this option are:

- **On (`--auto_type`)** [default for C++11 or later] — A variable's type may be omitted and the `auto` keyword used instead to have the type automatically deduced from the variable's initializer.
- **Off (`--no_auto_type`)** — [default for all other dialects] The `auto` keyword is treated as a storage class specifier.

The `--no_auto_type` setting is not supported in language modes for C++11 or later.

User-Defined Literals

Controls support for C++11-style user-defined literals. Permitted settings for this option are:

- **On (`--user_defined_literals`)** — [default for C++11 or later]
- **Off (`--no_user_defined_literals`)** — [default for all other dialects]

Thread Safe Initializations

Controls whether initializations are thread-safe. Permitted settings for this option are:

- **On (`--thread_safe_initializations`)** — If the `_mtx_*` functions are defined, adds thread safety to `std::call_once`, the initializations of function-local static variables, and various run-time library mechanisms. (The library and run time require the `_mtx_*` functions in order to make these actions thread-safe. For more information, see the file *install_dir/src/libsys/ind_thrd11.c* included in your compiler installation). This option is enabled by default in language modes for C++11 or later.
- **Off (`--no_thread_safe_initializations`)** — Removes thread-safety from `std::call_once`, the initializations of function-local static variables, and various run-time library mechanisms. This setting should not be used unless your application is single threaded.



Note

This option is deprecated and will be subsumed by the **-single_threaded** option in future compiler versions.



Note

Thread-safe initialization uses `mutex` to ensure thread safety. For this reason, any constraints with respect to `mutex` accesses across signal and interrupt boundaries also apply to thread-safe initialization. Variables that require thread-safe initialization should not be used in signal or interrupt contexts.

Allow String Literals to be Assigned to non-const char Pointers

Controls whether the deprecated conversion from string literal to `non-const char *` is allowed in C++11 mode. Permitted settings for this option are:

- **Allowed** (**--deprecated_nonconst_string_literal_conv**) — [default for **--c++11** and non-C++11 dialects]
- **Disallowed** (**--no_deprecated_nonconst_string_literal_conv**) — [default for **--C++11** or later]

This conversion results in a warning in language modes for C++11 or later, even when it is explicitly allowed via the **--deprecated_nonconst_string_literal_conv** option.

Allow Narrowing Conversions in C++11 Dialects

Controls whether narrowing conversions (as defined in the C++11 standard) are allowed in language modes for C++11 or later. Permitted settings for this option are:

- **Allowed** (**--narrowing_conversions**)
- **Disallowed** (**--no_narrowing_conversions**) — [default]

This option is ignored in language modes other than C++11 or later.

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 313.

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 and directory of the object file with the **.lst** extension.
- **Generate User-Specified Listing (-list=*filename*)** — Creates a listing with the specified name and directory. When using project files, this option should only be applied to individual source files.
- **Suppress Listing (-no_list)** — [default]

Source Listing Generation Output Directory

Outputs assembly listing files in the specified directory, unless the **-list** option is used with a directory, such as **-list=*directory/name***. 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)** — Files with suffix **.s** and **.asm** will be preprocessed and all preprocessed assembly files will be given a **.si** suffix.
- **Off (-no_preprocess_assembly_files)** — [default] Only assembly files with special extensions are preprocessed.

Preprocess Special Assembly Files

Controls whether assembly files with a **.arm64**, **.a64**, **.sx**, or **.cppasm** extension are preprocessed. Permitted settings for this option are:

- **On** (**-preprocess_special_assembly_files**) — [default]
- **Off** (**-no_preprocess_special_assembly_files**) — Files with a special 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 **asarm64** 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 that can be passed in this manner, see Chapter 6, “The asarm64 Assembler” on page 389.

Assembler Command File

Passes the options specified in *filename* directly to the assembler. The equivalent driver option is:

- **-asmcmd**=*filename*

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 asarm64 Assembler” on page 389.

Support for C Type Information in Assembly Files

Controls assembler support for the `.inspect` and `.struct` directives, as well as 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 files. This option is not supported with **-kanji=shiftjis/utf8**. For more information, see Chapter 7, “Assembler Directives” on page 405, and “C Type Information Operators” on page 398. 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 314).

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. This setting implies **-nostdlib** (meaning the linker will not link in any startup files or libraries) and **-undefined** (meaning the linker will not give errors for undefined symbols). Note also that this setting will not allocate common variables.

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 to retain relocations and avoid locating the program.

- **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 both **-a** and **-r** to the linker to locate the program and retain relocations.

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 513.
- **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. If both **-srec** and **-hex** are on the driver command line, **-hex** will take effect even if it is followed by **-srec**. For more information about **gsrec**, see “The gsrec Utility Program” on page 532.
- **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 532.
- **None (--no_additional_output)** — [default] You can use this setting 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 561.

Start Address Symbol

Specifies the program entry point or start address. The argument must be the name of a symbol in an executable section or an address in an executable section. (See “Specifying the Program Entry Point” on page 429.) Permitted settings for this option are:

- **Start Address** (**-entry** *symbol*|-**e** *symbol*) — The default symbol is `_start`.
- **No Entry Symbol** (**-noentry**) — The linker will choose a symbol based on the symbols defined in the program (see “Specifying the Program Entry Point” on page 429).

Append Comment Section with Link-Time Information

Controls whether to embed linker version information in the comment section of linker output. This is similar to using `#pragma ident` (see “General Pragma Directives” on page 748). 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** *filename*

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`, pass `-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. To access the data, adapt the following example:

```
extern char __ghsbegin_myrawdata[], __ghsend_myrawdata[];

int main() {
    const char *p=__ghsbegin_myrawdata;
    /* print contents of raw section, one byte at a time */
    while( p < __ghsend_myrawdata ){
        printf("0x%02x ", (unsigned char)*p);
        p++;
    }
    return 0;
}
```

For more information, see “Using Section Attributes” on page 443 and “Defining a Section Map with the SECTIONS Directive” on page 434.

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 424.

For additional information about passing linker options directly, see “Additional Linker Options (before start file)” on page 232.

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 424.

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 424.

For additional information about passing linker options directly, see “Additional Linker Options (before start file)” on page 232.

Linker Command File

Passes the options specified in *filename* directly to the linker. The equivalent driver option is:

- **-lnkcmd=filename**

The file must contain only one linker option per line. For example:

```
-multiple  
-undefined  
-e _mystart
```

For more information about linker-specific options, see “Linker-Specific Options” on page 424.

Extract Whole Archive

Permitted settings for this option are:

- **On** (**--whole-archive**) — Cause all files to be extracted from any library that appears after the option **--whole-archive** on the command line until **--no-whole-archive**.
- **Off** (**--no-whole-archive**) — [default] Undo the effect of any **--whole-archive** option for any libraries after the option **--no-whole-archive** until **--whole-archive**.

These options only apply to libraries specified on the **ccarm64** command line and not to any libraries selected implicitly by the **ccarm64** command. These options can be used when creating a merged archive if the option **-merge_archive** is also used.

If the same library is specified on the command line twice, all of its contents will be extracted twice even if only one instance of that library on the command line is after **--whole-archive**.

Report an Error When Multiple Libraries on a Link Line Contain a Definition of the Same Symbol

Controls whether an error is reported when a symbol is defined in multiple libraries on the link line. Permitted settings for this option are:

- **On** (**-link_reject_duplicate_lib_syms**)
- **Off** (**-no_link_reject_duplicate_lib_syms**) — [default]

When Rejecting Duplicate Symbols, Allow Certain Symbols Added by the compiler

When rejecting duplicate symbols, controls whether certain symbols known to be generated by the compiler (as opposed to those present in source) are allowed. Note that not all compiler-generated symbols will be allowed by this option. Permitted settings for this option are:

- **On** (**-link_allow_duplicate_ghs_syms**)

- **Off** (`-no_link_allow_duplicate_ghs_syms`) — [default]

When Rejecting Duplicate Symbols, Allow Symbols in .ghs.linkonce sections

When rejecting duplicate symbols, controls whether those in `.ghs.linkonce` sections are allowed. Permitted settings for this option are:

- **On** (`-link_allow_duplicate_linkonce_syms`)
- **Off** (`-no_link_allow_duplicate_linkonce_syms`) — [default]

Duplicate Whitelist

When rejecting duplicate symbols, allows those present in the specified whitelist. A whitelist is a text file with one symbol per newline. To allow all symbols from a library, use `@libfoo.a` where `libfoo.a` is the base name of the library. Path information should be omitted, and symbols will be allowed from any library with that name. The equivalent driver option is:

- **-duplicate_whitelist=whitelist**

Linker Optimizations

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

These options control individual optimizations that can be implemented at link-time.

Deletion of Unused Functions

Controls the removal of unused and unreferenced functions from an executable. 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 459.



Note

The setting **-delete** may cause DWARF debugging information to be out of sync with the program in memory.



Note

This option is not supported with INTEGRITY versions prior to 10.0, nor with INTEGRITY IoT.

Deletion of Unused Data

Controls whether to remove unused and unreferenced data from an executable. This option may improve program size if your code declares many variables that are ultimately unused. However, it may also impact other optimizations and has important caveats discussed below. Permitted settings for this option are:

- **On (-data_delete)**
- **Off (-no_data_delete)** — [default]

Keep in mind that modules not built with **-data_delete** may be ineligible for this optimization, and certain options that create references to data may inhibit data deletion. Additionally, **-data_delete** may disable some optimizations that improve address calculations using a shared base offset between variables.



Note

Data deletion does not affect uninitialized global variables in C that are **COMMON** (see “Allocation of Uninitialized Global Variables” on page 91).



Note

This option is not supported with INTEGRITY versions prior to 10.0.

Deletion of Unused C++ Virtual Functions

Enables the deletion of C++ virtual functions. This option has no effect unless you pass **-delete**. It also requires that all object files in the program be compiled with Green Hills compilers. This option is enabled implicitly by **-Olink**. Permitted settings are:

- **On (-uvfd)**
- **Off (-no_uvfd)** [default]

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 460.



Note

The setting **-codefactor** may cause DWARF debugging information to be out of sync with the program in memory

Link-Time Constant Propagation

Controls the link-time constant propagation optimization. Permitted settings for this option are:

- **On (--link_constprop)** — Instructs the linker to optimize the program further when new constants are resolved at link time, for example, when weak symbols are defined/left undefined during the final link (i.e. the constant 0 will be propagated for weak undefined symbols).
- **Off (--no_link_constprop)** — [default]



Note

The setting **--link_constprop** may cause DWARF debugging information to be out of sync with the program in memory.

Link-Time Trivial Function Inlining

Allows inlining of calls to trivial functions without deleting callees. Examples of trivial functions include those whose whole body is a return or tail-call instruction.

- **On (-inline_trivial)**
- **Off (-no_inline_trivial)** — [default]

This option is enabled implicitly by **-Olink** (see “Linker Optimizations” in “Optimization Options” on page 119). Note that we recommend that you use **-Olink** rather than individually enabling its implied options.

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)** — [default] 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. Start files are the first object files passed on the command line to the linker. The default is to use the Green Hills start files. Permitted settings for this option are:

- **Use Specified Start Files** (**-startfiles**=*filename*[*filename*]...)
- **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**=*filename*[*filename*]...

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)**

In general, you should not use **-multiple**, as this option suppresses a fatal link error. It is primarily useful in bypassing build failures in order to gain further visibility into the build process. 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] The linker gives a fatal error for each undefined symbol.
- **Silent (-undefined)** — The linker gives each undefined symbol an address of zero, or a zero offset if you are using PID.

In general, you should not use **-undefined**, as this option suppresses a fatal link error. It is primarily useful in bypassing build failures in order to gain further visibility into the build process. 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:

- **-Cname=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
```



Note

The builder may expand relative paths for any libraries specified as files, requiring you to use the full path as the argument to **-extractall**. To specify the full path in a portable manner, see the `%expand_path` macro function in “Macro Functions” on page 922.

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
```

Record Pathname (Not linked)

This option instructs the linker to create a special symbol that notifies **intex** of the location of INTEGRITY shared libraries. It allows you to link an image against a shared library without pulling the library into the image. This is used for libraries that are already built into the kernel. The equivalent driver option is:

- **-UA=string**

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] Generates a listing of symbols sorted alphabetically by name.
- **Numeric (-Mn)** — Generates a listing of symbols sorted numerically by address.
- **Both Alphabetic and Numeric (-Man|-Mna)** — Generates both a listing of symbols sorted alphabetically by name and a listing of symbols sorted numerically by address.

Map File Cross-Referencing

Controls the generation of cross-reference information in the map file. Permitted settings for this option are:

- **On (-Mx)**
- **Off (-no_Mx)** — [default] You can use this setting in a MULTI Project (.gpj) file, but there is no equivalent driver option.

Display Section Layout in Map File

Adds information about the correlation between output sections and section inclusion rules in the linker directives file. Permitted settings for this option are:

- **On (-Ms)**
- **Off (-no_Ms)** — [default] You can use this setting in a MULTI Project (.gpj) file, but there is no equivalent driver option.

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 459.
- **Off (-no_Mu)** — [default] You can use this setting in a MULTI Project (.gpj) file, but there is no equivalent driver option.

Display Local Symbols in Map File

Permitted settings for this option are:

- **On (-MI)** — Adds list of locals to the linker-generated map file. This list does not contain zero-sized compiler-generated symbols.
- **Off (-no_MI)** — [default] You can use this setting in a MULTI Project (.gpj) file, but there is no equivalent driver option.

Display End-of-Function Symbols in Map File

Controls whether symbols of the form `__ghs_eofn_name` appear in the map file, where *name* is the name of a function. These symbols may be generated by the compiler under certain circumstances, such as when DWARF is being output. By default, end-of-function symbols appear in the map files unless the program is built with `-mapfile_type=2`. For more information about linker-specific options, see “Linker-Specific Options” on page 424. Permitted settings for this option are:

- **Off (-no_map_eofn_symbols)**
- **On (-map_eofn_symbols)**

Output File Size Analysis

Controls use of the **gsize** utility to determine output file size. The **gsize** utility creates a file with the same root name as the output file and a `.siz` suffix. The file contains a list of section sizes. Permitted settings for this option are:

- **On (-gsize)**
- **Off (-no_gsize)** — [default]

For more information, see “The gsize Utility Program” on page 531.

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 non-compressed 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` with an initial value of `0` (by default), or `0x04C11DB7` with an initial value of `0xffffffff` (for INTEGRITY). For information about how to use these checksums, see “Verifying Program Integrity” on page 76. Sample source code for verifying checksums is included in `src/libstartup/cksum.c` of your installation.

Link-Time Argument Type Checking

Controls the level of link-time argument type checking. When enabled, the linker will issue a warning when an externally visible function is called with a type that

does not match its definition (argument type or return type). Pass this option when compiling and linking. Permitted settings are:

- **Full (-argcheck=full)** — Generates warnings for type inconsistencies and when type-checking information is unavailable. (Files built with the `none` setting will not trigger warnings for missing type-checking information.)
- **Normal (-argcheck=normal)** — Generates warnings for type inconsistencies (if type-checking information is available).
- **None (-argcheck=none)** — [default] Disables argument type checking.

Type inconsistencies typically occur when a function is not prototyped or is called in one file but externally defined with a different prototype. The **-argcheck** option guards against these inconsistencies by issuing warnings. However, not all type differences will be noted by **-argcheck**. For example:

- Qualifiers such as `const` and `volatile` are ignored.
- The signedness of integer arguments is ignored.
- The `int` type and all smaller integer types are treated as equivalent. (This is because smaller integers are promoted to `ints` when passed to a function.)
- Pointers to all types are treated as equivalent.
- The `struct` and `union` types are distinguished by size only. If a function expecting one type of `struct` or `union` is passed a `struct` or `union` with an identical size but different field types, no warning will be issued. This is true even if the function expects a `struct` but is passed a `union` or vice versa.

Important notes on link-time argument checking:

- **-argcheck** cannot detect mismatches between a function definition and a pointer to a function in different files if the latter is within a structure.
- In some cases, a mismatch for a function pointer passed as an argument may not be detected.
- If a function's address is assigned to a function pointer variable of prototyped type, **-argcheck** will treat it as a call to that function.
- Some issues that **-argcheck** can identify can also be caught at compile time. To ensure that all functions are called with the correct argument types, check for the following compiler diagnostics: 223 (`function declared`

implicitly) and 1547 (call is made to a non-prototyped function).

- If you are building an INTEGRITY project, set this option to `none`. Link-time argument checking is not supported with INTEGRITY.
- Do not use utilities to modify the symbol table (such as **gstrip** and **ghide**). These utilities may interfere with the symbols **-argcheck** uses for type checking.

Adding Argument Type-Checking Information to Assembly Functions

Argument type-checking information is usually generated automatically by the compiler. However, the compiler cannot generate this information for functions written in assembly code. If a project is linked with **-argcheck=full**, any functions defined in assembly code must be manually annotated with type-checking information. Otherwise, you may see errors like the following:

```
[elxr] (warning #215) function: function called in module doesn't have an  
argument type checking declaration
```

To add these annotations, define a global symbol corresponding to each defined assembly function and, optionally, another symbol corresponding to each function called from assembly code. The format for these symbols is given below.

For definitions:

```
__callee.name.return_type.parameter_types
```

For calls:

```
__caller.name.return_type.parameter_types
```

Where:

- *name* is the name of the function.
- *return_type* is a base type that encodes the return type (see below).
- *parameter_types* is one or more concatenated base types that encode the parameter types.

Supported base types:

- *i* — Integer type, 32 bits or smaller, signed or unsigned.

- `l` — Integer type, 64 bits, signed or unsigned.
- `p` — Any pointer type.
- `f` — Single-precision float type (32-bit).
- `d` — Double-precision float type (64-bit).
- `snum` — struct or union type, where `num` is the number of registers needed to store the type.
- `e` — The start of a variable argument list (“...” in C). If `e` is used, it must be the last parameter.
- `v` — void type. If void type is used in the parameter list, it must be the only parameter.

For example, a function implemented in assembly and prototyped in C as:

```
void myfunc(int x, char *y, ...);
```

should be given the **-argcheck** symbol:

```
__callee.myfunc.v.ipe
```

Any difference between the types specified in the `callee` and `caller` symbol will result in a link-time warning.

Link-Time Global Variable Type Checking

Controls the level of link-time global variable type checking. Permitted settings for this option are:

- **Full (-globalcheck=full)** — Generates warnings in the event of inconsistent types for the definition and uses of global variables, or where type-checking information is unavailable.
- **Normal (-globalcheck=normal)** — Generates warnings in the event of inconsistent types for the definition and uses of global variables only where type-checking information is available.
- **None (-globalcheck=none)** — [default] Does not generate or use global variable type-checking information.

The link-time global variable type checking verifies that the definition and all uses of a global variable agree on the data type of the variable (size and signedness). Qualifiers such as `const` and `volatile` can optionally also be checked by using **-globalcheck_qualifiers** (see “Link-Time Global Variable Type Qualifiers” on page 250)

This option should be passed both when compiling and when linking.



Note

Full type checking (**-globalcheck=full**) requires that all files in the project be built with type checking. The standard libraries are not built with this option enabled, so the linker may generate warnings about missing type declarations for any library variables that are used in code where **-globalcheck=full** is passed. Some options, such as run-time error checking, may generate implicit references to variables in the libraries.

In most cases, it is recommended that you use **-globalcheck=normal** for type checking.

Link-Time Global Variable Type Qualifiers

Controls whether the linker verifies qualifiers such as `const` and `volatile` when performing link-time global variable type checking. Permitted settings for this option are:

- **On (-globalcheck_qualifiers)** — The type checking, if enabled, verifies qualifiers on the type in addition to size and signedness.
- **Off (-no_globalcheck_qualifiers)** — [default] The type checking, if enabled, verifies only size and signedness.

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=filename**

Quit Building if Warnings are Generated

Controls whether the build will terminate with an error if any compiler warnings are generated, providing the ability to treat all compiler warnings as errors. This option does not affect warnings issued by the Builder, driver, linker, assembler, or **dblink**. To treat Builder warnings as errors, pass the **-strict** option to the Builder (for more information, see “The gbuild Utility Program” on page 488). 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]

Write Version Info to .comment Section

Creates or appends version and build information to the `.comment` section of any object file generated by the assembler or compiler. (These are strings identifying the version of the compiler and any driver options passed when the object was built.) Permitted settings for this option are:

- **On** (**-version_info**)
- **Off** (**-no_version_info**) — [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 260). 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 900.
- **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 13, “Coding Standards” on page 717. The equivalent driver option is:

- **--coding_standard=filename**

Where *filename* is the name of a coding standard. If *filename* does not end in a file extension, the coding standard will be searched for as a **.csp** file in the compiler directory **install_dir/defaults/coding_standards**. Otherwise, the argument will be processed as the full filename of a coding standard file (including the extension) 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 Diagnostic 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`)

Exclude Error Message Paths

Controls whether full pathnames are excluded from files mentioned in diagnostic messages. Permitted settings for this option are:

- **On** (`-error_basename`) — Display only file base names in diagnostic messages.
- **Off** (`-no_error_basename`) — [default] — Display filenames and path information in diagnostic messages.

Show Header Trace for Diagnostics

Controls whether diagnostic messages emitted from header files are preceded by a trace of the directives to `#include` said header files. Permitted settings for this option are:

- **On** (`--diagnostic_header_trace`)

- **Off** (`--no_diagnostic_header_trace`) — [default]

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 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`)



Note

In C89—but not C99—dialects, diagnostic 223 is also controlled by this option. (Diagnostic 223 is issued for implicitly declared functions.)

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 ISO C++]
- **Silent** (`--asm_silent`) — [default for all other C and C++ modes]

In strict ANSI C and strict ISO C99, 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 C dialects, ISO C99, and in C++, the two spellings (`asm` and `__asm`) are identical. This option controls whether a diagnostic indicating that `__asm` is non-standard is given for a correct use of `__asm` (or in non-strict C and in C++, a correct use of `asm`).

Any correct use of `__asm` may still give a diagnostic indicating that `__asm` is non-standard.



Note

This one option controls both C and C++ languages. Using either strict ANSI C or strict ISO C++ will cause this option to default to **`--asm_warnings`** for both C and C++. Using strict ISO C99 and any non-strict dialect will give a remark by default.

Unknown Pragma Directives

Controls the treatment of `#pragma` directives that are not recognized by the compiler. Green Hills extension `#pragma` directives (those prefixed with `#pragma ghs`) are always recognized by the compiler, even if the extension directives do not correspond to known or supported behaviors. Incorrect uses of Green Hills extension `#pragma` directives are diagnosed according to the "Incorrect Pragma Directives" option. See “Incorrect Pragma Directives” on page 256 and “Green Hills Extension Pragma Directives” on page 752. 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]

64-bit Conversion Warnings

This option is designed to assist with converting 32-bit code to 64-bit. The **-Wconvert64** flag will enable a set of warnings that are otherwise typically disabled. Permitted settings for this option are:

- **On (-Wconvert64)** — Turns on the following warnings:

- 68: integer sign change, for example:

```
unsigned int ui = 0;
// ...
ui = -10;                // causes underflow
```

- 767: caused when converting a pointer to a smaller integer type, for example:

```
int          x = 0;
unsigned short us = 0;
// ...
us = (unsigned short)&x; // loses upper bits of the pointer value
```

- 1373 and 1374: implicit or explicit narrowing of integer type from 64-bit integer to a smaller integer type, for example:

```
long long x64 = 0LL, y64 = 0LL;
int       x32 = 0;
// ...
x32 = x64 + y64;           // loses upper bits of the integer value
```

- 1375: conversion of pointer to an integer of the same size (which could be different when ported to 64-bits), for example:

```
int   x32 = 0;
void *ptr = NULL;
// ...
x32 = (int)ptr;           // correct when 'void *' and 'int' are
                          // the same size, but causes problems when
                          // ported to 64-bits
```

- 2033: possibly unintended loss of precision, for example:

```
long long x64 = 0LL;
int       x32 = 0, y32 = 0;
// ...
x64 = x32 * y32;          // x32 * y32 is a 32-bit value, possibly
                          // truncated. However, it is assigned to
                          // 64-bit, and the user likely intended to
                          // keep all 64-bits of the computation
```

- 1703: converting pointer-to-integer of one size to pointer-to-integer of another size, for example:

```
long long *p64 = NULL;
int       *p32 = NULL;
// ...
p32 = (int *)p64;         // accessing through *p32 will incorrectly
                          // access the original object (only partially)
p64 = (long long*)p32;    // accessing through *p64 will access more
                          // than the original object, possibly
                          // accessing illegal memory
```

- **Off (-Wno-convert64)** — [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 *discretionary* C and C++ compiler diagnostic messages. A diagnostic is discretionary if it is issued as

a remark, a warning, or an error with a diagnostic number followed by a "-D" suffix. As arguments, these options take a list of either diagnostic numbers or tags. Each diagnostic has both an associated number and textual tag. For example, both `--diag_error=9` and `--diag_error=nested_comment` are equivalent. Values for the diagnostic numbers can be obtained by enabling the **Compiler Diagnostics→C/C++ Messages→Varying Message Severity→Display Diagnostic Numbers (--display_diag_number)** option (see below). The complete set of diagnostic numbers is also listed in the *MULTI: Toolchain Error Messages and Other Diagnostics* book (not available in print).

Set Message to Error

Sets the specified compiler diagnostic messages to the level of *error*:

- `--diag_error=n[,n2]...`

Set Message to Warning

Sets the specified compiler diagnostic messages to the level of *warning*:

- `--diag_warning=n[,n2]...`

Set Message to Remark

Sets the specified compiler diagnostic messages to the level of *remark*:

- `--diag_remark=n[,n2]...`

Set Message to Silent

Sets the specified compiler diagnostic messages to the level of *silent*:

- `--diag_suppress=n[,n2]...`

Display Diagnostic Numbers

Controls the display of diagnostic message numbers in the compiler and in assemblers that support this feature. Permitted settings for this option are:

- **On** (`--display_diag_number`) — [default]
- **Off** (`--no_display_diag_number`)

When a generated diagnostic message number has a **-D** suffix, it is discretionary. For example:

```
"hello.c", line 24: warning #549-D:  
variable "i" is used before its value is set
```

The severity level of discretionary diagnostics may be changed using the preceding options.

Toolchain Messages

This option category is contained within **Compiler Diagnostics**.

Set Linker Message to Error

Sets the specified `elxr` diagnostic messages to the level of error:

- `-elxr_diag_error=n[,n2]...`

Set Linker Message to Warning

Sets the specified `elxr` diagnostic messages to the level of warning:

- `-elxr_diag_warning=n[,n2]...`

Set Linker Message to Silent

Sets the specified `elxr` diagnostic messages to the level of silent:

- `-elxr_diag_suppress=n[,n2]...`

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 357.

DoubleCheck Report File

Instructs DoubleCheck to produce a report file with the given name. If this option is not used, no report is generated. The equivalent driver option is:

- **-double_check.report=filename**

For more information about **-double_check.report**, see “Specifying a DoubleCheck Report File” on page 360.

DoubleCheck Config File

Specifies the configuration file (**.dcc**) containing function properties. The equivalent driver option is:

- **-double_check.config=filename**

For more information about **-double_check.config**, see “Specifying Function Properties in a DoubleCheck Configuration File” on page 361.

DoubleCheck Errors to Ignore

Disables *all* or a specific DoubleCheck error/warning message (*number*) for a file or part of a build hierarchy. Note that this option assumes that **-double_check.level** has been set properly for the diagnostics in question (see “DoubleCheck Level” on page 262). The equivalent driver option is:

- **-double_check.ignore=all | number**

For more information about **-double_check.ignore**, see “Ignoring and Enabling Source Analysis Errors and Warnings” on page 368.

DoubleCheck Errors to Enable

Enables *all* or a specific DoubleCheck error/warning message (*number*) for a file or part of a build hierarchy. Note that this option assumes that **-double_check.level** has been set properly for the diagnostics in question (see “DoubleCheck Level” on page 262). The equivalent driver option is:

- **-double_check.enable=all | number**

For more information, see “Ignoring and Enabling Source Analysis Errors and Warnings” on page 368.

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 367.

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 79.

Target Processor

Specifies code generation for a particular target *processor* (see “ARM64 Processor Variants” on page 37).

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 60. 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 429.

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).



Note

To function correctly, this option's setting must be uniformly applied to all parts of a program.

Support for Very Large Switch Statements

Controls the size of switch statement offset entries. Permitted settings for this option are:

- **On (-bigswitch)** — Use a 32-bit offset. This option is necessary when compiling a file with a very large switch statement. The assembler or linker will give a specific error if this option is needed for a particular file. Otherwise the option is not necessary.
- **Off (-nobigswitch)** — [default] Use a 16-bit offset, which results in smaller code.

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, except for **tgmath.h**. No error is given when a single precision floating-point expression is passed to a function that is not declared with a prototype or to a function that takes variable parameters, even though the floating-point expression will be promoted to double precision before being passed.
- **Off (-fnonodouble)** — [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 83; it completely disables floating point in the compiler and may cause different libraries to be selected. If **-fnone** is used together with either **-fnonodouble** or **-fnodouble**, **-fnone** always takes precedence.



Note

Because the relevant header files have not been modified for **-fnodouble**, some C++, **tgmath.h**, or OS (including INTEGRITY) headers may use doubles and give an error accordingly.

Advanced Operating System

This option category is contained within **Advanced**→**Target Options**.

Include libbsp.a

This option is deprecated and provided for backwards compatibility only.

Controls whether your INTEGRITY kernel application will link against **libbsp.a**. Permitted settings for this option are:

- **On (-libbsp)** — [default]
- **Off (-no_libbsp)**

This option has no effect if you are not building an INTEGRITY kernel project.

Relocatable Module

This option is deprecated in INTEGRITY 5, and does not exist in INTEGRITY 10.

Creates a relocatable module, `mod`, that can be dynamically loaded to an already running *velOSity* system. Permitted settings for this option are:

- **On (-irel)**
- **Off (-no_irel)**

Relocatable Module Base Image

This option is deprecated in INTEGRITY 5, and does not exist in INTEGRITY 10.

Specifies an alternate kernel image if the default kernel is not used (see “Building Relocatable Modules” in the *velOSity User's Guide*). The equivalent driver option is:

- **-dyload**

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 113.

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) 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 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. Note that if an error occurs during processing, some files may be invalid or incomplete.
- **Delete (-nokeeptempfiles)** — [default]

Temporary Assembly File Directory

Specifies a directory for retaining assembly files with **-keepasmfiles**. This option implies **-keepasmfiles**. If **-asmfile_dir** is placed on the command line before **-nokeepasmfiles**, no assembly files will be retained and this option will have no effect. This option is not supported in the builder. The driver option is:

- **-asmfile_dir=directory**



Note

If multiple object files are created with identical base names in different locations, and **-asmfile_dir** is set to the same directory for these files, one assembly file will be used for all of the object files. The assembly file will be overwritten when any of the object files are built. In a parallel build, a wrong version of the assembly file may be used or the assembly file may be corrupted, resulting in build errors or an incorrect program.

Temporary Assembly Files

Permitted settings for this option are:

- **Retain (-keepasmfiles)** — Retains the temporary **.s** file produced when the compiler is invoked on a C or C++ file to produce an object file, and retains the temporary **.si** file produced when the preprocessor is invoked on an assembly file to produce an object file. The assembly file has the same base name as the

object file. Unless **-asmfile_dir** is used, the assembly file resides in the same location as the object file.

- **Delete (-nokeepasmfiles)** — [default] Places temporary **.s** or **.si** file in the temporary directory and deletes it after use.

If **-asmfile_dir** and **-nokeepasmfiles** are on the same command line, the latter option will take effect.

Temporary Preprocessor Output Directory

Specifies a directory for the preprocessor output produced with **-keepcppfiles**. This option implies **-keepcppfiles**. If **-cppfile_dir** is placed on the command line before **-nokeepcppfiles**, no preprocessor output will be retained and this option will have no effect. This option is not supported in the builder. The driver option is:

- **-cppfile_dir=directory**



Note

If multiple object files are created with identical base names in different locations, and **-cppfile_dir** is set to the same directory for these files, one preprocessor file will be used for all the object files. The preprocessor output file will be overwritten when any of the object files are built. In a parallel build, the preprocessor output file may even be corrupted.

Temporary Preprocessor Output

Permitted settings for this option are:

- **Retain (-keepcppfiles)** — Produces a **.i** file (preprocessed output) when the compiler is invoked on a C or C++ file to produce an assembly or object file. Produces a **.si** file when the preprocessor is invoked on an assembly file to produce an object file when building with **-preprocess_assembly_files** (see “Preprocess Assembly Files” on page 224). The preprocessor output file has the same base name as the final assembly or object file. Unless **-cppfile_dir** is used, the preprocessor output resides in the same location as the final assembly or object file.

- **Delete (-nokeepcppfiles)** — [default] Does not produce **.i** files when the compiler is invoked on a C or C++ file. Places preprocessor output produced when the preprocessor is invoked on an assembly file in the temporary directory and deletes it after use.

If **-cppfile_dir** and **-nokeepcppfiles** are on the same command line, the latter option will take effect.



Note

If an error is encountered during preprocessing, preprocessed output files will be deleted even if **-keepcppfiles** is used. To retain preprocessed output files even on error, pass **-keepcppfiles -keeptempfiles**.

Line Numbers in Temporary Preprocessor Output

Controls whether the **.i** files produced by **-keepcppfiles** for C and C++ source files include **#line** directives (like **-E**) or omit them (like **-P**). This option implies **-keepcppfiles**. Permitted settings for this option are:

- **Include (-keepcppfileswithlines)**
- **Omit (-nokeepcppfileswithlines)** — [default]



Note

This option has no effect on preprocessed assembly files, which always include **#line** directives.

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 on a driver command with exclusively source-language input files.

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 **cxarm64**, **-language=cxx** is implied.

Non-Standard Output Suffix

Controls the acceptance of non-standard output suffixes (see “Output File Types” on page 14). 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) 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**

Note that because **:appendExtension** is not inherited, it should be set on each object that requires it, rather than on a parent project.

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 source files in the Program, but before linking it.
- If you set **:preexec** on a C/C++ 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 277.

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 969).

If you only want to execute the command when processing a file in certain contexts, use a conditional control statement (see “Conditional Control Statements” on page 918). For example, to run **preProc.bat** on a file only when building on a 32-bit 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 276.

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 276.

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 278.

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 — */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 — `/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
Library	Archiving
C/C++ Header	None
Project	None

For example:

- If you set **:postexec** on an assembly source file, the commands are run after assembling the file, but before linking it.
- If you set **:postexec** on a C/C++ 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 280.

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 969).

If you only want to execute the command when processing a file in certain contexts, use a conditional control statement (see “Conditional Control Statements” on page 918). For example, to run **postProc.bat** on a program's output file when building on a 32-bit 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 279.

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 279.

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 281.

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 — `/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 — `/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 `c, asm`, the Builder searches your **Select One** project for a `.c` file first, then a `.asm` 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, 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 use 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.



Note

There are also predefined options files available, designed to aid with secure coding. For more information, see “Green Hills Secure Coding Options Files” on page 944.

Build Macro to Define

Sets a macro for use in a project file. Because **:macro** is treated as an option rather than a header directive, it can be set in an options file. Additionally, setting this option in the project options section will override any setting of the same macro in the project header. While **:macro** can be set in the children section (see “The Children Section” on page 912), it is only supported if the child it is set on is a project file. The syntax for this option is:

- **:macro=KEY=VALUE**

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 **-Llibrary_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 119.

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 “Using Inlining Optimizations” on page 328.

Inline Constant Math Functions

At compile time, this optimization evaluates certain calls to math library functions that have compile-time constant arguments. For example, `sqrt()` and `sqrtf()` are evaluated for all finite, non-negative values, and `exp2()` and `exp2f()` are evaluated for all integer values that do not cause underflow or overflow. In addition, various special cases for the following functions are recognized for both the single- and double-precision versions: `exp()`, `exp2()`, `log()`, `log2()`, `log10()`, `sin()`, `asin()`, `cos()`, `acos()`, `tan()`, `atan()`.

This optimization is automatically enabled if **-frigor** is set to `fast` or `default` and one of the following optimization strategies is used: **-Ogeneral**, **-Osize**, or **-Ospeed**. If **-frigor** is set to `accurate` when using the above strategies, this optimization is disabled but can be manually enabled. In all other circumstances it is not performed, even if explicitly turned on. Permitted settings for this option are:

- **On (-Oinline_constant_math)** — Allows inlining of math functions with constant parameters.
- **Off (-Ono_inline_constant_math)** — Disables inlining of math functions with constant parameters if this has been enabled by an optimization strategy.



Note

The values computed for these functions at compile time may differ slightly from what would have been computed at run time.

Inline C Memory Functions

Allows the compiler to inline calls to certain C memory functions. This option is always off with **-Onone**, **-Odebug**, **-Omoredebug**, and **-Omaxdebug**. It is on by default with **-Ogeneral** or **-Ospeed** (except with **-ANSI**) and it is otherwise off by default. Permitted settings for this option are:

- **On (-Omemfuncs)**
- **Off (-Onomemfuncs)** — [default]



Note

Most inlining of memory functions requires the inclusion of **string.h**. **-Omemfuncs** may produce multiple copies of memory functions; do not use the option if these functions require unique addresses. Additionally, this option is not strictly standards compliant.



Note

This option is deprecated and will be removed in a future release.

Convert C Memset to Memclr

Allows you to disable an optimization that replaces `memset` function calls with a constant zero parameter to `__gh_memclr32` or `__gh_memclr`, depending on the optimization strategy and whether the memory is known to be aligned. Permitted settings for this option are:

- **Off (-Onomemclr)** — Disables this optimization if it is enabled by another optimization option (such as an optimization strategy). You may want to use this option if you define your own `memset` function.
- **On (-Omemclr)** — [default] Cancels the effect of **-Onomemclr**.

Inline C String Functions

Allows the compiler to inline calls to certain C string functions. These may include `strlen()`, `strcpy()`, `strcmp()`, `strchr()`, `strrchr()`, `strncmp()`, `strncpy()`, `strcat()`, `strncat()`, and `strstr()`. This option is always off with **-Onone**, **-Odebug**, **-Omoredebug**, and **-Omaxdebug**. The option is on by default with **-Ogeneral** or **-Ospeed** (except with **-ANSI**), and off by default otherwise. Permitted settings for this option are:

- **On (-Ostrfuncs)**
- **Off (-Onostrfuncs)** — [default]



Note

Most inlining of string functions requires the inclusion of **string.h**. **-Ostrfuncs** may produce multiple copies of string functions; do not use the option if these functions require unique addresses. Additionally, this option is not strictly standards compliant.

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**.

Explode Jumps

Allows you to disable the explode jump optimization. This optimization will replace jump instructions with the code they would otherwise jump to, duplicating code to remove the jump instruction for speed. Permitted settings for this option are:

- **On (-Oexplodejumps)** — Cancels the effect of **-Onoexplodejumps**.
- **Off (-Onoexplodejumps)** — Disables this optimization if it is enabled by another optimization option (such as an optimization strategy).

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 344.

Register Allocation by Coloring

Controls the dynamic storage of variables in registers.

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. (This is because 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 it is left as-is to simplify the example:

Initial C source code	Optimized C source code
<pre>int subr(int x) { int a,b; a=x; b=x*2; return b; }</pre>	<pre>int subr(int x) { int a; a=x; a=x*2; return a; }</pre>

Scalar variables are generally considered for register allocation unless their values are accessed with the address operator (&). 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**.

Automatic Register Allocation

Controls the automatic allocation of local variables to registers. For example:

Initial C source code	Optimized C source code
<pre>extern int global; int foo(int e) { int local = e; if (global) local = local * 2; return local; }</pre>	<pre>extern int global; int foo(int e) { register int local = e; if (global) local = local * 2; return local; }</pre>

Permitted settings for this option are:

- **On (-autoregister)** — [default]
- **Off (-noautoregister)**

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 352.

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 353.

Constant Propagation

Controls the *Constant Propagation* optimization. This option is deprecated and will be removed in a future release. 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 353.



Note

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 354.

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

For more information, see “Memory Optimization” on page 354.



Note

This option is deprecated and will be removed in a future release.

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 (Including SIMD Vectorization)” on page 340.

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 351.

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 351.

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 119). Permitted settings for this option are:

- **-O0** — Use **-Onone**.
- **-O1** — Use **-Ogeneral**.
- **-O2** — Use **-Ospeed** and **-Onounroll**.
- **-O3** — Use **-Ospeed**, **-OI**, and **-OB**.

Align Standalone Objects

Controls the use of optimizations that increase the alignment of objects beyond their natural alignment. For example, an array containing 4 members of type `short` may have a natural alignment of 2 bytes, but the array may be aligned on a higher

alignment boundary for improved performance. Note that even after passing **-no_align_standalone_objects**, variable alignment may increase in some cases. Permitted settings for this option are:

- **On** (**-align_standalone_objects**) — [default]
- **Off** (**-no_align_standalone_objects**)

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 132.

Generate Function Entry/Exit Instrumentation

Controls whether the compiler generates function entry/exit instrumentation. Permitted settings for this option are:

- **Off** (**-no_gen_entry_exit_history**)
- **On** (**-gen_entry_exit_history**)

For more information, see “Function Entry/Exit Logging” on page 133. This feature is licensed separately from the Green Hills toolchain.



Note

The setting **-gen_entry_exit_history** may cause DWARF debugging information to be out of sync with the program in memory, unless the option **-mute_entry_exit_history** is also enabled.

EAGLE Record Signal

Indicated EAGLE signals are enabled by the linker. Signals can be disabled at run-time with the **eagleconfig** MULTI command. For information about this

command, see the MULTI: Debugging Command Reference. The equivalent driver option is:

- **-record_eagle_signal**=*signal1*[,*signal2*]...

This feature is licensed separately from the Green Hills toolchain.

EAGLE Mute Signal

Indicated EAGLE signals are disabled by the linker. Signals can be enabled at run-time with the **eagleconfig** MULTI command. For information about this command, see the MULTI: Debugging Command Reference. The equivalent driver option is:

- **-mute_eagle_signal**=*signal1*[,*signal2*]...

This feature is licensed separately from the Green Hills toolchain.

EAGLE Strip Signal

Indicated EAGLE signals are stripped from the code by the linker. (Stripped code is similar though not identical to code without EAGLE signals.) Stripped signals cannot be enabled at run-time with the **eagleconfig** MULTI command. For information about this command, see the MULTI: Debugging Command Reference. The equivalent driver option is:

- **-strip_eagle_signal**=*signal1*[,*signal2*]...

This feature is licensed separately from the Green Hills toolchain.

Generate Target-Walkable Stack

When this option is set to **On**, routines always use a frame pointer, stored in register `r29`. You can use this option to allow the program running on the target to perform stack traces on itself. This capability enables the target code to retrieve information about its function call stack without using the Debugger, which is necessary if the Debugger is not available or if obtaining information from the Debugger is prohibitively expensive. Some debugging features, such as **-check=alloc**, depend

on this option being set to **On**. If you have enabled optimizations, there might be gaps in the stack trace. This option might also produce unexpected complications, such as disabling certain optimizations.

When this option is set to **Off**, some routines do not create a stack frame. This may reduce code size and improve performance.

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 the same code with the **--no_debug** option as it does with either **-g** or **-G**. Do not use this option with **-g**, **-gs**, or **-G**. Permitted settings for this option are:

- **On** (**-consistentcode**)
- **Off** (**-noconsistentcode**) — [default]

INTEGRITY TimeMachine Enabled

Specifies whether the compiler generates INTEGRITY TimeMachine-instrumented code. Please refer to your INTEGRITY documentation for configuration information. Permitted settings for this option are:

- **Off** (**-no_gen_timemachine**)
- **On** (**-gen_timemachine**)

INTEGRITY TimeMachine instrumentation may be stripped away by the linker unless you also pass **-record_timemachine** (see “INTEGRITY TimeMachine” on page 132). This feature is licensed separately from the Green Hills toolchain.

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, consider the following three commands:

1. `ccarm64 file1.c file2.c -G -relobj -o libfile12.o`
2. `rm -rf file1.o file2.o file1.dbo file2.dbo`
3. `ccarm64 file3.c libfile12.o -G`

Performing the above commands in the order listed will result in warnings about debug info not being found. Adding **-search_for_dba** to commands 1 and 3 will resolve this issue: It will cause command 1 to create **libfoo12.dba** and command 3 to make use of it.

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.
- **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** 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 42 and “Source Scanning” on page 42.

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.

Multi-Core Configuration File

Specifies the path to a multi-core configuration file (***.ghsmc**) that the MULTI Debugger uses to set up your multi-core target. The equivalent driver option is:

- **-ghsmc_file=*path***

For more information, see the documentation about using a multi-core configuration file in *MULTI: Debugging*.

Multi-Core Core Count

Applies the project executable to any SMP hardware target whose core count is within the specified range (boundaries are included). The equivalent driver option is:

- **-ghsmc_core_count=range**

Where *range* can be specified in four ways:

-ghsmc_core_count=CoreCount	Enable SMP debugging when target hardware has exactly the number of cores as <i>CoreCount</i> .
-ghsmc_core_count=MinCoreCount-	Enable SMP debugging when target hardware has at least as many cores as <i>MinCoreCount</i> .
-ghsmc_core_count=-MaxCoreCount	Enable SMP debugging when target hardware has up to <i>MaxCoreCount</i> cores.
-ghsmc_core_count=MinCoreCount-MaxCoreCount	Enable SMP debugging when target hardware has a number of cores that falls within <i>[MinCoreCount, MaxCoreCount]</i> , inclusive.

Debug Info Linker Extra Arguments

Directs the compiler driver to append *argument* to commands used to invoke **dblink**. The equivalent driver option is:

- **-dblink.args=argument**

Commonly appended arguments include the following:

- **-dloVersion=number** — Directs **dblink** to generate **.dla** information of the given version number.
- **-dnmVersion=number** — Directs **dblink** to generate **.dnm** information of the given version number.

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 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.



Note

When using **--xref=none** or **--xref=global**, the MULTI Debugger may not correctly display a variable if it is shadowed by an identically named variable. For example:

```
void foo(int x, int z)
{
    {
        int y = x;    // 'x' refers to the parameter variable
        int x = z;    // new 'x' is declared, and is shadowing
                     // the parameter 'x'
        // ...
    }
    // ...
}
```

In the code above, MULTI may not pick the version of `x` the user wants to see. This inaccuracy will not occur when using **--xref=full** or **--xref=declare**.

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.

Alternate Debugging Information

This option category is contained within **Advanced**→**Debugging Options**.

Generate MULTI and DWARF 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. Generated DWARF information conforms to the DWARF 2.0 standard. Permitted settings for this option are:

- **On (-dual_debug-dwarf2)**
- **Off (-no_dual_debug)** — [default]

To have any effect, this option requires that **Debugging**→**Debugging Level** be set to Plain or MULTI (see “Debugging Level” on page 132).



Note

The Green Hills toolchain does not generate DWARF Call Frame information that is suitable for unwinding the call stack. For third-party debuggers incapable of unwinding the call stack through an alternate means (disassembling code), this may nullify the ability to walk the call stack or view variables saved in previous frames.

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** (**-dwarf2dbo**)
- **Off** (**-no_dwarf2dbo**) — [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 140.

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 setting 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]

Message Pragma

Controls the acceptance of `#pragma message` directives in C and C++ sources. Permitted settings for this option are:

- **Accept pragma message and emit message (--pragma_message)** — [default] Accept `#pragma message` directives and print the string to standard error output while compiling. The `-w` option does not suppress messages produced

by the directive. When **-quit_after_warnings** is used, compilation does not stop if a message is printed.

- **Accept pragma message and suppress message** (**--pragma_message_silent**) — Accept and ignore `#pragma message` directives in any format.
- **Do not accept pragma message** (**--pragma_message_unknown**) — Treat `#pragma message` directives as unknown `#pragma` directives. See “Unknown Pragma Directives” on page 256.

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 Standard Include Directories

Changes the default location in which the compiler searches for the C header files provided with the C library. The equivalent driver option is:

- **-c_include_directory** *directory*

Headers found in directories specified with this option are treated specially in the following ways:

- Warnings, remarks and a few discretionary errors are suppressed even if the severity is raised by the user.
- The headers are omitted from dependency files created with **-MMD**.

C++ Standard Include Directories

Changes the default location in which the compiler searches for the C++ header files provided with the C++ library. The equivalent driver option is:

- **-std_cxx_include_directory** *directory*

Headers found in directories specified with this option are treated specially in the following ways:

- Warnings, remarks and a few discretionary errors are suppressed even if the severity is raised by the user.
- The headers are omitted from dependency files created with **-MMD**.

System Include Directories

Changes the default location in which the compiler searches for system header files. This option may be specified multiple times to cause the compiler to search in multiple directories. Any directory that is not specified with an absolute path is located relative to the current working directory. The equivalent driver option is:

- **-sys_include_directory** *directory*

Headers found in directories specified with this option are treated specially in the following ways:

- Warnings, remarks and a few discretionary errors are suppressed even if the severity is raised by the user.
- The headers are omitted from dependency files created with **-MMD**.

RTOS Include Directories

Specifies directories in which the compiler searches for system header files after searching any directories specified by **-sys_include_directory**. This option may be specified multiple times. Any directory specified without an absolute path will

be located relative to the directory specified by **-os_dir**. The equivalent driver option is:

- **-rtos_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 142.

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. When using C++, 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 and Caveats” on page 334.

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 how the `-gnu99` C dialect will determine the linkage for functions with the `inline` keyword. Permitted settings for this option are:

- **ISO C99 Inline Semantics** (`-fno-gnu89-inline`) — [default] The `inline` keyword will be handled in accordance with the ISO C99 standard.
- **GNU C89 Inline Semantics** (`-fgnu89-inline`) — The `inline` keyword will be handled in a manner similar to that of earlier GNU compilers.

Prepend String to Every Section Name

Specifies a *string* to prepend to the name of each standard and user-defined section. The names of special sections, such as those used to record profiling data with `-coverage`, are not affected. The equivalent driver option is:

- `--section_prefix string`



Note

Using this feature may produce unexpected behavior if the names of sections with special meanings are changed (such as those beginning with `.ghs.linkonce`).

Append String to Every Section Name

Specifies a *string* to append to the name of each standard and user-defined section. The names of special sections, such as those used to record profiling data with **-coverage**, are not affected. The equivalent driver option is:

- **--section_suffix** *string*



Note

Using this feature may produce unexpected behavior if the names of sections with special meanings are changed (such as those beginning with `.ghs.linkonce`).

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. Support for changing the type of `wchar_t` is only available for stand-alone, u-Velocity, ThreadX, or custom operating system targets. Note that the representation of wide characters is also dependent on the **-kanji** option (see “Host and Target Character Encoding” on page 210). Permitted settings for this option are:

- **-wchar_u16** — `wchar_t` is unsigned short.
- **-wchar_s32** — [default] `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 823) and causes the linker to associate calls to some standard C library functions to alternate names (see “Libraries” on page 776).



Note

To function correctly, this option's setting must be uniformly applied to all parts of a program.

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, but may be enabled or disabled independently in non-GNU modes. Permitted settings for this option are:

- **On (--gnu_asm)** — [default in 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)** — [default in non-GNU modes] 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 748). 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`)

**Note**

The **--munch** setting is not supported on INTEGRITY versions after 11, including INTEGRITY IoT.

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 313. 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 142.

Assembler Warnings

Controls the display of assembler warning messages. Permitted settings for this option are:

- **Display (--assembler_warnings)** — [default]

- **Suppress (`--no_assembler_warnings`)**

Linker Options

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

These options control advanced linker settings. For more common options, see “Linker Options” on page 227. 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**

Address Space Layout Randomization Seed

Specifies a random seed for address space layout randomization (ASLR). During link time, ASLR randomly reorders code or data in output sections marked with the `RANDOMIZE` attribute.

Note that, given the same source code and seed value, affected sections will retain the same ordering. If a seed value is not supplied with **--aslr_seed** but the **RANDOMIZE** attribute is used, the linker will pick an arbitrary seed based on the system clock. For the sake of reproducible builds, the seed chosen will be output to a file with the suffix **.aslr_seed**. The seed value can then be fed to **--aslr_seed** to rebuild the same executable. For more information, see “Address Space Layout Randomization” on page 464. The equivalent driver option is:

- **--aslr_seed=number**

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 238).

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.



Note

This option may cause the compiler to avoid introducing a call to a routine in the Green Hills libraries if the same result could be achieved less efficiently without calling said routine. For example, this option may result in the use of repeated store instructions instead of a call to `__gh_memclr32`.

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]



Note

This option may cause the compiler to avoid introducing a call to a routine in the Green Hills libraries if the same result could be achieved less efficiently without calling said routine. For example, this option may result in the use of repeated store instructions instead of a call to `__gh_memclr32`.

Force Deletion of Unused C++ Virtual Function Symbol

Specifies C++ virtual function symbols that should be deleted even if they appear to be used. This option has no effect unless you pass **-delete**. It also requires that all object files in the program be compiled with Green Hills compilers. The equivalent driver option is:

- **-force_uvfd=symbol**

To specify more than one symbol, use this option multiple times.

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` — [default]

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 `.ROMy`, `.CROMy`, or `.ROM_NOCOPYy` is inserted following `z` as well.

For more information about these attributes and linker directives file syntax, see Chapter 8, “The elxr Linker” on page 423.

Monotonic Block Order

Controls whether sections are treated as `MONOTONIC`. Permitted settings for this option are:

- **On** (`-monotonic_block_order`)
- **Off** (`-nomonotonic_block_order`) — [default]

For more information, see the `MONOTONIC` attribute in “Using Section Attributes” on page 443.

Archiver Options

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

Output Archives in Large Format

Specifies when the archiver should output archives in the large format and when to print related warning messages. Permitted settings for this option are:

- **When Needed (With Warnings)** (`-auto_large_archive`) — [default] If the input archive exists and is in large format, large format will be used. Otherwise, large format will not be used except in `r` mode where the files specified on the command line require it. In the latter instance, a warning will be output. To get this behavior without a warning, use `-quiet_auto_large_archive`. If the large format is ultimately needed but not selected by the archiver, the archive will be unchanged and the archiver will output a fatal error recommending the use of `-large_archive`. When working with large files, we recommend manually specifying `-large_archive` rather than relying on the archiver's automatic format detection.

- **When Needed (Suppress Warnings) (-quiet_auto_large_archive)** — Same as **-auto_large_archive**, except the archiver does not issue warnings related to the selection of the large format.
- **Always (-large_archive)** — The archiver always uses the large format without warnings.
- **Never (-no_large_archive)** — The archiver never uses large archives. This will fail and the archive will be unchanged if the large format is ultimately needed.

**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.

Error on Duplicate Symbols in an Archive

When creating or modifying an archive, controls whether an error is generated if the archive would have more than one global definition of a symbol with the same name in its table of contents. If an error is generated, the archive is removed.

Permitted settings for this option are:

- **Generate Error (-reject_duplicates)** — Output an error and remove the archive if the table of contents would have more than one global definition of a symbol.
- **Ignore (-no_reject_duplicates)** — [default] Do not detect if the table of contents would have more than one global definition of a symbol.

When Rejecting Duplicate Symbols in An Archive, Certain Symbols Generated by the C++ Front End in COMMON Sections Are Ignored

When rejecting duplicate symbols in an archive, controls whether those symbols known to be generated automatically by the compiler in order to support C++ language features are allowed, as opposed to those present in the original source. Permitted settings for this option are:

- **Ignore Duplicates (-allow_cxx_duplicates)**
- **Generate Error (-no_allow_cxx_duplicates)** — [default]

When Rejecting Duplicate Symbols in An Archive, Symbols in .ghs.linkonce Sections Are Ignored

When rejecting duplicate symbols in an archive, controls whether those symbols in sections with names beginning with `.ghs.linkonce` are allowed. These symbols are automatically generated by the compiler to support template instantiation. Permitted settings for this option are:

- **Ignore Duplicates** (`-allow_linkonce_duplicates`)
- **Generate Error** (`-no_allow_linkonce_duplicates`) — [default]

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=filename`

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:

```
ccarm64 file1.c -stderr_overwrite=file1.err -c
ccarm64 file2.c -stderr_overwrite=file2.err -c
```

as opposed to how you might normally use `-stderr`:

```
ccarm64 file1.c -stderr=err.out -c
ccarm64 file2.c -stderr=err.out -c
```

Symbol Blacklist

Specifies the path to a file containing a newline-separated list of symbols. The assembler and linker will produce a diagnostic if the output ELF file would reference or define a non-static symbol contained in the blacklist. Symbol names for C++

functions must be mangled. In the case of a build that separately performs compilation, assembly, or linkage, this option must be passed to each driver invocation to catch all references and definitions of blacklisted symbols. The equivalent driver option is:

- **-symbol_blacklist_file**=*filename*

Identifier Blacklist

Specifies the path to a file containing a newline-separated list of identifiers. The compiler will produce a diagnostic if the source file contains any identifiers in the blacklist. The equivalent driver option is:

- **-identifier_blacklist_file**=*filename*

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:

- **-Xswitch**

EDG Front End Options

Specifies one or more internal EDG front end options. The equivalent driver option is:

- **--option**=*option*

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)**
- **Off (-no_dbo_trace)** — [default] You can use this setting in a MULTI Project (.gpj) file, but there is no equivalent driver option.

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

Inlining Optimizations	326
Interprocedural Optimizations	335
Loop Optimizations (Including SIMD Vectorization)	340
General Optimizations	350

Optimizing a program using the Green Hills toolchain begins with selecting an appropriate over-arching optimization strategy (see “Optimization Strategy” on page 119). Each strategy collects a variety of optimizations into a single setting, striking one possible balance between execution speed, program size, and debugging ability. The available optimization strategies are designed to cover the needs of most project types and development cycles.

However, there may be additional improvements available from optimizations that are not enabled by optimization strategies. Further, there may be circumstances in which it is advantageous to selectively disable particular optimizations. The remaining sections in this chapter provide detailed information that will be useful in making these judgments.

To provide a sense of how optimizations transform the compiler's internal representation of code, many optimization descriptions also include a pre and post-optimization code comparison. These should be regarded as insights into compiler behavior, rather than explicit instructions on how best to manually optimize code.

Inlining Optimizations

The term *inlining* refers to the process of substituting a call to a function with the actual content of that function. This eliminates overhead and may result in faster code at the cost of an increase in program size.

For example, the main program in the case below 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
<pre>inline void sub(int x) { printf("x=%d\n", x); } int main(void) { for (int i=1; i<10; i++) sub(i); return 0; }</pre>	<pre>int main(void) { for (int i=1; i<10; i++) printf("x=%d\n", i); return 0; }</pre>

When You Should Consider Inlining Optimizations

Inlining works best when there are small, frequently executed functions called in only a few locations. In these circumstances, inlining can increase program speed in high usage areas without significantly increasing program size.

If program size is at a premium or debuggability is a high priority, inlining may not be advisable. If the latter is true, note that source line number information related to inlined routines will be deleted by the compiler, and no inlined source code will be available to a debugger. Single stepping by source line will cause any inlined call to be executed as a single statement. (You can, however, debug any inlined call by stepping through the sequence of inlined assembly instructions at the point of the source-level call.)

Finally, be aware that inlining can have counterintuitive impacts on a program's size and speed. See details below.

Advantages of Inlining

As explained above, inlining is an optimization that typically 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 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 that 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 the compiler performs 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 that 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 (specifically, those that must live across the call). If all function calls can be eliminated by inlining, the number of registers available for variables and temporaries increases.

Inlining may reduce other negative effects that a function call typically incurs. 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.

Using Inlining Optimizations

Unless you have selected a debugging optimization strategy (or passed **-Onone**, **-g**, or **-G**) the compiler will attempt to automatically inline a subset of simple functions. These functions are judged by the compiler to be the safe targets of straightforward inlining optimizations. There are two types of such functions:

- Static functions that are referenced only once.
- Very small functions, such as single simple assignments.

But there are far more functions where the ultimate impact of inlining is complex enough that it will not be done automatically. This is where user-initiated inlining methods come into play. There are three such methods, none of which are mutually exclusive:

1. Manually inserting keywords in your code to suggest functions for single-pass inlining. Single-pass inlining is the simpler and more straightforward form of inlining. An example is given in the introduction to this section (see “Inlining Optimizations” on page 326).
2. Using a driver command (**-OI**, with arguments) to indicate specific functions for two-pass inlining. Two-pass inlining is a more aggressive and thorough form of inlining (see “Indicating Functions for Two-Pass Inlining” on page 331).
3. Using a driver command (**-OI**, without arguments), to direct the compiler to automatically use two-pass inlining on as many smaller functions as possible. This is different from the automatic inlining discussed above in that the compiler will select larger, more complex functions to be automatically inlined, and will also attempt the more involved and aggressive means of inlining (see “Indicating Functions for Two-Pass Inlining” on page 331).

Note that all of the above methods only *suggest* functions for inlining. The compiler must still determine whether a given request is sensible, and will examine the

function to be inlined, the calling function, and other factors. If you are using C++, you should review some additional details before implementing these methods (see “Additional C++ Inlining Information and Caveats” on page 334). Finally, keep in mind that enabling **-Omaxdebug** will cause the compiler to ignore all inlining requests.

Manually Indicating Functions for Single-Pass Inlining

The sections below describe the manual inlining process for supported programming languages. Note that manual inlining will only be effective when the function to be inlined is defined in the same module as the caller.

Manual Inlining In C

To manually indicate a function for single-pass inlining, identify the `inline` keyword associated with your C language mode (see the table at the end of this section), and place it immediately before the target function. If the compiler decides that a function declared with an `inline` keyword cannot be inlined, it will either create an out-of-line copy in the module or externally link to another module's out-of-line copy. The toolchain will use three different linking methods, depending on how the function was declared:

- *Exported Inline* — The compiler will generate 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 in two or more modules, the linker will report that you have multiply defined symbols.
- *Imported Inline* — The compiler will not generate an out-of-line copy of the function. It will instead assume 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 will generate a normal static function only if 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 a static inline function, there are some important consequences:
 - If you compare two static inline functions in different modules, they will not equate due to differing addresses.

- Your code size might increase due to multiple out-of-line copies of the function.

Static inlining should be used 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
<code>inline</code>	exported	imported	imported
<code>__inline</code>	exported	imported	static
<code>__inline__</code>	exported	imported	static
<code>extern inline</code>	imported	exported	exported
<code>extern __inline</code>	imported	exported	static
<code>extern __inline__</code>	imported	exported	static
<code>static inline</code>	static	static	static

Manual Inlining in C++

To manually indicate a C++ function for single-pass inlining, place the `inline` keyword immediately before the function declaration. 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.

Indicating Functions for Two-Pass Inlining

Unlike single-pass inlining, two-pass inlining involves a precompilation pass over all source files. This pass generates intermediate code for all target functions and stores it in **.inf** files. On the second pass, each source file is compiled normally, but the compiler is given all of the **.inf** files as additional input. Using this information, the compiler is able to inline functions across source files and different programming languages. Note that some of the inlining performed will look identical to methods used in single-pass inlining. Two-pass inlining simply allows for intermodule inlining if doing so makes sense for optimization. When implementing two-pass inlining, keep the following in mind:

- Compiler warnings will be suppressed on the first pass to prevent issuing duplicate warnings.
- If you have different projects that share the same source file, you will need to set a different **:outputDir** for each of the projects to prevent linker errors.

To suggest particular functions for two-pass inlining:



Use **Optimization**→**Individual Functions**→**Inline Specific Functions**
(**-OI=fn1[,fn2]...**).

To direct the compiler to consider as many smaller functions as possible for two-pass inlining:



Use **Optimization**→**Intermodule Inlining** (**-OI/-Onoinline**).

You can increase the size threshold for functions automatically considered for two-pass inlining by also passing **-OB**. If you need even larger functions to be inlined, they will need to be manually indicated as arguments for **-OI**.

Combining the Inlining Option Variants

There are a number of ways to combine the available inlining option variants. Below is a series of optimization-related command lines followed by explanations of their effects. Consider the contents of two files:

file_a.c

```
int a_very_big_function(void)
{
    /* lots of code in here, not shown for brevity... */
    return 0;
}
int b(void)
{
    return 2;
}
```

file_b.c

```
extern int a_very_big_function(void);
extern int b(void);
static inline int c(void)
{
    return 3;
}
int main(void)
{
    return a_very_big_function() + b() + c();
}
```

Given the command line:

```
ccarm64 -OI file_a.c file_b.c:
```

- The function `a_very_big_function()` will not be inlined, even though **-OI** is enabled, because the compiler deems it to be too large.
- The function `b()` will be inlined because **-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:

```
ccarm64 -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 **-OI=a_very_big_function**.
- The function `b()` will not be inlined because **-OI** (without arguments) 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:

```
ccarm64 file_a.c file_b.c
```

- The functions `a_very_big_function()` and `b()` will not be inlined. In the former case, this is because the function was not given as an argument to **-OI**. In the latter case, this is because **-OI** was not passed without arguments nor was the function given as an argument to **-OI**.
- 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.

Inlining Options You Should Generally Leave Alone

Depending on your optimization strategy, the compiler will commonly inline certain math functions (**-Oinline_constant_math**), C string functions (**-Ostrfuncs**), and C memory functions (**-Omemfuncs**). All of these inlining optimizations are designed to function as part of various optimization strategies, with little to be gained when used outside of this context. In general, you should not manually enable or disable these options unless you are attempting to work around an intractable bug in your code. For more information about these options, see “Inline Constant Math Functions” on page 286, “Inline C String Functions” on page 288, and “Inline C Memory Functions” on page 287.

Additional C++ Inlining Information and Caveats

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 will not inline very large functions.

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 308. 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 such functions and the `#pragma ghs endnoinline` directive after such functions. The `__noinline` keyword can also be used to suppress the inlining of non-template functions, but legacy reasons cause the keyword to behave differently when applied to a function template or a member function of a class template.

If you pass `--disable_noinline`, the `__noinline` keyword will have no effect on function templates and member functions of class templates. However, if you pass `--enable_noinline`, the `__noinline` keyword will control whether such a function is considered implicitly declared inline. The primary purpose of this interaction is to control how such member functions are treated by the `--instantiate_extern_inline` option. The `__noinline` keyword does not prevent the compiler from inlining function templates or member functions of class templates, although it may still affect the inlining outcome due to treating the member function as no longer declared inline.



Note

`__noinline` does not affect function templates and member functions of class templates that are declared as `constexpr`. These functions are implicitly inline as a result of the `constexpr` keyword and `__noinline` will not change that.

Interprocedural Optimizations

Interprocedural optimizations are a collection of methods designed to perform analyses of function use within and between source files. These methods determine points at which functions can be inlined, reduced, or otherwise efficiently transformed. The overall effect tends to increase program speed and reduce program size. The cost of these optimizations is slower compile times, reduced debuggability, and greater RAM requirements on host machines. The compiler can perform many different interprocedural optimizations. To get a sense of these optimizations, consider two representative examples:

Example 4.1. Alias Analysis

By default, the compiler assumes that all memory may potentially be read or written by function calls. This is a safe assumption, but it leads to conservative decisions that are not optimal for code size or speed. When this optimization is enabled, the compiler looks for additional information that will allow it to eliminate some regions of memory as possible read/write targets for functions.

Initial C Source Code	Optimized C Source Code
<pre>int X = 0; int Y; void may_write_X() { Y = 10; // Does not write X, // but the compiler must // assume it is possible. } void use_global() { X = 1; may_write_X(); if (X != 1) do_large_computation(); }</pre>	<pre>int X = 0; int Y; void may_write_X() { Y = 10; // Does not write X, // but the compiler can // determine this fact. // Now do_large_computation() // can be discarded. } void use_global() { X = 1; may_write_X(); }</pre>

Example 4.2. Constant Returns

By default, the compiler assumes that the return value of a function call can be any in the domain of its return type. As with the default behavior discussed in the prior example, this is a safe assumption but it is not optimal for code size and speed. The

constant returns optimization identifies an instance where this assumption can be discarded: when a function consistently returns a particular value at compile time.

Initial C Source Code	Optimized C Source Code
<pre>int x = 0; int foo() { x++; return 0; } void bar() { if (foo()) // The compiler must assume // that foo() can return any // integer, even though foo() // will always return 0. x *= 100; }</pre>	<pre>int x = 0; int foo() { x++; return 0; } void bar() { foo(); // The compiler can determine // that foo() will always // return 0. Now the statement // "x *=100;" can be discarded. }</pre>

The two methods shown above are good illustrations of interprocedural optimizations, but they are not designed to be enabled on an individual basis. Instead, interprocedural optimizations are enabled collectively, in one of two sets: a smaller set called **Standard Interprocedural Optimizations (-Ointerproc)**, and a larger, more thorough set called **Wholeprogram Interprocedural Optimizations (-Owholeprogram)**. (The latter set adds methods that require the compiler to have total program visibility.) Note that these two sets of optimizations cannot be mixed within your program: you must choose one or the other.

When You Should Consider Interprocedural Optimizations

As long as you are not concerned about increasing compile times and reducing debuggability, the standard form of interprocedural optimizations (**-Ointerproc**) will likely be advantageous. Wholeprogram optimizations are yet more powerful,

but there are additional considerations when employing them. For example, consider the wholeprogram optimization *IP Constant Propagation*:

Example 4.3. IP Constant Propagation

By default, the compiler does not notice if a function parameter is always set to the same constant value by callers. With this optimization, the compiler will transform any use of such a parameter into a constant instead of setting it in callers.

Initial C Source Code	Optimized C Source Code
<pre>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); }</pre>	<pre>int can_be_simple(int y) { return y; } int is_simple() { return can_be_simple(1) + can_be_simple(2); }</pre>

Optimizations like **IP Constant Propagation** are useful, but they introduce problems if the compiler does not have complete program visibility. For example, if a module is calling `can_be_simple(10, 10)` in the program above, and this module's interactions with the program are unknown to the compiler, then the optimization will be incorrectly applied. That is, the call to `do_large_computation()` will be discarded in error, and `x+y` will be incorrectly transformed into `y`. Wholeprogram optimizations thus create additional demands on your build strategies. For example, if you plan to reuse object files across projects, and you additionally plan to do so without rebuilding these object files, wholeprogram optimizations should generally be avoided. If you still want to employ wholeprogram optimizations in these circumstances, you will need to perform a rigorous and comprehensive accounting of the interactions between the non-rebuilt object files and the rest of your program (see “Using Wholeprogram Interprocedural Optimizations” on page 338). Providing the compiler with an incomplete list of referenced functions and variables will lead to a variety of problems.

Be aware that if you intend to use interprocedural optimizations on different projects that share the same source file, you will need to set a different **:outputDir** for each project to prevent linker errors.

Disabling Specific Interprocedural Optimizations

The compiler comes equipped with many options to disable specific interprocedural optimizations (see “Interprocedural Optimization Scope” on page 126). These options are primarily designed to work around issues derived from detrimental interactions between interprocedural optimizations and your code. They should only be used when indicated by Green Hills Support.

Using Standard Interprocedural Optimizations

To enable the standard collection of interprocedural optimizations:



Set the **Optimization→Interprocedural Optimizations** option to **Standard Interprocedural Optimizations (-Ointerproc)**.

For an example of a method included with **Standard Interprocedural Optimizations**, see “Interprocedural Optimizations” on page 335.

Using Wholeprogram Interprocedural Optimizations

To enable the wholeprogram collection of interprocedural optimizations:



Set the **Optimization→Interprocedural Optimizations** option to **Wholeprogram Interprocedural Optimizations (-Owholeprogram)**.

For an example of a method included with **Wholeprogram Interprocedural Optimizations**, see “When You Should Consider Interprocedural Optimizations” on page 336.

As noted earlier, for wholeprogram optimizations to work correctly, all function call sites must be made available to the compiler. If there are functions or variables located in files that you do not want modified by this option, you must comprehensively indicate them to the compiler. You will also need to indicate any

such symbols used in assembly code. To provide the necessary information, use one or more of the following:

- The options **-external** or **-external_file** (see “Symbols Referenced Externally” on page 124).
- The pragma `ghs_start_externally_visible` (see “Green Hills Extension Pragma Directives” on page 752).
- The GNU attribute `externally_visible`.

You will also need to properly exclude the program parts in question (see the next section).



Note

If you plan to use **-Owholeprogram** and makefiles, see Example 1.5. Using Wholeprogram Interprocedural Optimizations With Makefiles on page 21.

Selectively Excluding Compilation Units

Interprocedural optimizations can require code changes across a variety of different compilation units. Some of these changes can be quite extensive, such as with wholeprogram optimizations. But regardless of whether you employ **-Ointerproc** or **-Owholeprogram**, you may find circumstances in which it is necessary to exclude compilation units from optimization. For example, you may have a well-tested module that is certified by a third party. If so, keeping the certification may require that the module's code remain unchanged.

To selectively exclude specific compilation units while using **Standard Interprocedural Optimizations**, choose one of two options:

- Set **Optimization→Interprocedural Optimizations** to **Off (-Onoipa)** for each program part. Doing so will cause the compiler to wholly exclude the indicated units from analysis and optimization.
- Set **Optimization→Interprocedural Optimizations** to **Analysis Without Optimizations (-Oip_analysis_only)** for each program part. This will cause the compiler to analyze all such units but refrain from changing their code.

These two options present a tradeoff. You can exclude program parts completely using **-Onoipa**, as doing so may result in reduced compilation time. Alternately, you can direct the compiler to analyze excluded program parts without changing their code using **-Oip_anaysis_only**. This will likely increase compilation time relative to **-Onoipa**, but will allow the compiler to make more intelligent optimization decisions in non-excluded compilation units.

Selectively excluding specific compilation units while using **Wholeprogram Interprocedural Optimizations** is slightly different. In this case, you must use **-Oip_analysis_only** for each such program part (**-Onoipa** cannot be used). This restriction is necessary to fulfill the baseline requirement of accurate wholeprogram optimization: full program visibility.

Libraries and Interprocedural Optimizations

Interprocedural optimizations will operate differently on a program that contains libraries in a subproject (rather than simply linking to external libraries). The modified behavior will depend on the chosen set of interprocedural optimizations:

- If **-Ointerproc** is used, the library subproject will be optimized and compiled separately from the rest of the program.
- If **-Owholeprogram** is used, the library subproject will be compiled separately from the rest of the program. The compiler will analyze the library project for the information required by **-Owholeprogram**, but will not alter its code.

Loop Optimizations (Including SIMD Vectorization)

Loop optimizations are designed to take a common code structure—loops—and transform them for faster execution. Like inlining optimizations, loop optimizations frequently require more code than the original code segment, and thus can result in a larger program.

When You Should Consider Loop Optimizations

The efficacy of loop optimizations will depend on the specific form of the loops used in your code and the kind of loop optimization applied. Note that loop optimizations are not intended to be global optimizations. They should be applied

selectively, ideally after analyzing program performance and determining whether considerable execution time is consumed by one or more loops.

Although loop optimizations can increase program size, their highly localized application will tend to minimize this tradeoff, making their potential benefits worth exploring.

Finally, keep in mind that many loop optimizations are inhibited when using the weaker optimization strategies: **-Onone**, **-Omoredebug**, **-Odebug**, and **-Omaxdebug**.

Automatic Loop Optimization

This setting allows the compiler to transform loops in any instance where the performance benefit is judged to be significant relative to the increase in code size. Consider this option if loop-related performance issues are common throughout one or more source files. Automatic loop optimization is generally controlled by your optimization strategy (see “Optimization Strategy” on page 119). To control it individually:



Use the **Advanced**→**Optimization Options**→**Optimize All Appropriate Loops** option (**-OL** /**-Onoloop**).

Automatic loop optimization consists of common loop transformations such as strength reduction, loop invariant analysis, and loop unrolling (see “Loop Optimization Processes” on page 342).

Manual Loop Optimization

If performance problems are spread out among a small number of functions in many different source files, automatic loop optimization could cause an undesirable

increase in code size. In this instance, consider instructing the compiler to optimize only particular functions:



Use the **Optimization→Individual Functions→Loop Optimize Specific Functions** option (**-OL=fn1[,fn2]...**).

Note that **-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 **-Onoloop** is specified, loops in all functions will be optimized and **-OL=fn1[,fn2]...** is redundant.

For example, the following command line directs the compiler to consider `sub` and `sub2` for loop optimizations:

```
ccarm64 -OL=sub,sub2 main.c prog1.c prog2.c
```

As with automatic loop optimization, any functions targeted may be subjected to strength reduction, loop invariant analysis, and loop unrolling.

Loop Optimization Processes

Strength Reduction

Strength reduction allows the compiler to store the address of an array in a register and add the array's size to the register on each iteration. This is in contrast to a non-optimized scenario, where the compiler will access an array element by multiplying its size by the loop index. For example:

Initial C Source Code	Optimized C Source Code
<pre>subr() { int i; int q[4]; for (i=0;i<4;i++) q[i]=i; }</pre>	<pre>subr() { int i; int q[4]; int *_ptr; for (i=0, _ptr=q; i<4; i++) *_ptr++ = i; }</pre>

Strength reduction is included in automatic (**-OL**) and manual (**-OL=**) loop optimization.

Loop Invariant Analysis

Loop invariant analysis examines loops for unchanging expressions or address calculations across iterations. When identified, these computations are placed outside the loop and their values are stored in registers. For example:

Initial C Source Code	Optimized C Source Code
<pre> 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]; } } </pre>	<pre> 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; } } </pre>

Loop invariant analysis is included in automatic (-OL) and manual (-OL=) loop optimization.

Loop Motion

Loop motion searches for invariant statements (or inner-loops) that only need to be executed once, and then extracts them. For example:

Initial C Source Code	Optimized C Source Code
<pre> void lm1(void) { for (int i = 0; i < 128; i++) { arr1[i] = i; for (int j = 0; j < 128; j++) { arr2[j] = j; } } } </pre>	<pre> void lm1(void) { for (int j = 0; j < 128; j++) { arr2[j] = j; } for (int i = 0; i < 128; i++) { arr1[i] = i; } } </pre>

Loop motion is included in automatic (-OL) and manual (-OL=) loop optimization.

Loop Merging

Loop merging searches for opportunities to merge sets of similar loops into single loops. For example:

Initial C Source Code	Optimized C Source Code
<pre>void lm3(void) { for (size_t i = 0; i < N; i++) { arr1[i] = i; } for (size_t i = 0; i < N; i++) { arr2[i] = i; } }</pre>	<pre>void lm3(void) { for (size_t i = 0; i < N; i++) { arr1[i] = i; arr2[i] = i; } }</pre>

Loop merging is enabled when **-Omax** is paired with **-OL** or **-Ospeed**.

Loop Unrolling

Loop unrolling duplicates the code in innermost loops to produce more straightline code. This removes much of the loop overhead caused by branching and testing for a stop condition. It also allows for more instruction pipelining and better use of the register allocator.

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
<pre>subr(a) int a[]; { int i; for (i=0;i<100;i++) a[i]=i; }</pre>	<pre>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; } }</pre>

The initial code has 4 instructions per iteration:

- 1 comparison
- 1 memory store
- 1 increment
- 1 branch

Assuming each instruction takes exactly one cycle to complete, and assuming time outside of the loop is negligible, this code will take $100 * 4 = 400$ cycles to complete.

The optimized code has 10 instructions per iteration:

- 1 comparison
- 3 additions
- 4 memory stores
- 1 increment
- 1 branch

Using the same assumptions, this code will take $100 / 4 * 10 = 250$ cycles to complete, for a 37.5% improvement.

Loop unrolling is included in automatic (**-OL**) and manual (**-OL=**) loop optimization.

Note that the compiler may decline to unroll some loops because it anticipates a burdensome increase in code size. To increase the code size threshold and unroll larger loops, pass **-Ounrollmore**.

Finally, although loop unrolling is a powerful optimization, it has two notable tradeoffs. First, loop unrolling can significantly increase code size. Although code size increases are associated with all loop optimizations, they are particularly large for loop unrolling. Second, unrolled loops are treated as single units by the MULTI Debugger, removing the ability to debug specific loop iterations. Because of these tradeoffs, you may want to disable loop unrolling while keeping other loop optimizations intact:



Set the **Advanced→Optimization Options→Loop Unrolling** option to **Off** (**-Ounroll**).

SIMD Vectorization

Loop vectorization is an optimization process similar to loop unrolling. However, vectorization uses single instruction multiple data (SIMD) operations to allow loop iterations to happen in parallel. Of all loop optimizations, vectorization can generate the largest performance benefits, but its requirements are more exacting (see the sections below).

As with standard loop optimizations, you can manually specify loops for vectorization or apply automatic vectorization to an entire file. The choice of automatic or manual vectorization should follow the same considerations discussed for other loop optimizations (see “When You Should Consider Loop Optimizations” on page 340). If the compiler decides it would be disadvantageous to vectorize a particular loop, it will attempt other loop optimization methods (see “Loop Optimization Processes” on page 342).

To generate vectorized loops, the compiler requires the use of ARM64 processors.

Automatic Vector Optimization

As with automatic loop optimization, automatic vector optimization considers all loops for potential optimization. The compiler analyzes each loop to determine if vectorization is both possible and safe (see “Automatic Vectorization Validity Checking” on page 348). To enable this optimization:



Set the **Optimization→Automatic Vector Optimization** option to **On (-OV)**.

Manual Vectorization

Manual vectorization is enabled by passing **-OL** and is controlled through the following `#pragma` directive:

```
#pragma ghs vectorize[=n]
```

This `#pragma` must be placed directly before the loop to be vectorized. An optional vectorization factor, *n*, specifies the number of iterations that can be performed in

parallel: 2, 4, 8, 16, or 32. If no factor is specified, the compiler will analyze the loop to determine a factor. In the following code fragment, the compiler is instructed to vectorize a loop and select a valid vectorization factor:

```
#pragma ghs vectorize
for (i=0; i < 128; i++)
    a[i]=b[i]+c[i];
```

Unlike automatic vectorization, manual vectorization entails only very basic tests for suitability prior to optimization. As such, it is your responsibility to ensure that all relevant iterations can be performed independently (see “Vectorization Limitations” on page 347). Non-independence can lead to the generation of invalid code.

Vectorization Limitations

The vectorization optimization is subject to certain limitations, detailed below.

- Loops containing the following kinds of constructs cannot be vectorized:
 - Function calls (except for some specific intrinsic functions).
 - `if` statements (except for simple constructs that can be replaced by predicated code).
 - Switch statements.

Automatic vectorization will ignore loops that contain the above constructs. Manual vectorization will decline to vectorize such loops and produce warnings.

- Dependencies within a vector are not permitted. For example, the following loop cannot be safely vectorized, because the previous element of `a` is used when calculating `a[i]`:

```
#pragma ghs vectorize=4
for (i=0; i < 128; i++){
    a[i] = a[i-1];
}
```

After vectorization, `a[i-1]` and `a[i]` could be calculated at the same time, causing the use of incorrect values.

**Note**

The vectorization optimization is not guaranteed to reproduce arithmetic operations in the same sequence that would be executed absent vectorization. Because of this, floating-point calculations will exhibit different rounding.

- `#pragma ghs vectorize` instructs the compiler that vectorizing the loop is a valid transformation, and requests that it be done. However, there are many possible reasons that this optimization may not occur. Certain compilation options may inhibit vectorization, in some cases even suppressing the warning regarding not vectorizing a loop marked by the `#pragma`. Optimizing for speed, including loop optimizations, will increase the likelihood that the compiler is able to honor vectorization requests.
- The vectorization of a reduction (such as summing the elements in an array) is done via an associative transformation. Because partial results are calculated and then combined, the process can result in different overflow or underflow conditions. This transformation will not respect the sign of zero, even when **`-precise_signed_zero`** is passed.
- Because vector instructions do not support all data types for all arithmetic operations, a sequence of vector instructions might be required to perform an operation that can be accomplished via a single non-vector instruction. Alternatively, the optimization may use multiply-add vector instructions to perform two arithmetic operations in a single instruction.

Furthermore, vectorization attempts to use vector instructions to operate on data types that might not be available in non-vector mode. For example, a vector addition instruction may operate directly on 8-bit or 16-bit values without needing to first widen the values to 32-bits, perform the operation, and then narrow the 32-bit value back to the original size.

Automatic Vectorization Validity Checking

Automatic vectorization uses several tests to determine if a loop can be safely vectorized. If any of the tests fail, the compiler will decline to vectorize the loop.

These tests include:

- Verifying that there is at least one vector length that supports all necessary vector operations within the loop.
- Verifying that there are no unsupported operations within the loop being vectorized (`if`, `switch`, `goto` statements, or function calls). For more information, see “Vectorization Limitations” on page 347.
- Verifying that there are no instances of scalar expansion that would need to be stored and then retrieved from an indexed reference that is independent of the iteration variable. For example, the following code would not be vectorized because of the need to have multiple instances of `x[i]` within the inner loop:

```
for (i=0; i < 128; i++){
    for (j=0; j < 128; j++){
        x[i] = a[i][j] * b[i][j];
        c[i][j] += x[i];
    }
}
```

- Verifying that all indexed references use indices that are either constant within the loop being vectorized or are dependent upon the iteration variables. For example, the following loop would not be vectorized due to the reference `a[p[i]]`:

```
for (i=0; i < 128; i++) {
    a[p[i]] = b[i];
}
```

- Using the vector length, verifying that there are no data dependencies between two loop iterations where the data value written in one loop iteration would be used in another loop iteration. For example, if the vector length is determined to be 4, elements of this first loop would not be vectorized:

```
for (i=4; i < 128; i++){
    a[i] = a[i-3];
}
```

In contrast, this second loop would be vectorized:

```
for (i=4; i < 128; i++){  
    a[i] = a[i-4];  
}
```

This test is very conservative and only considers whether the reference spaces overlap, not whether a conflict occurs. For instance, the following two references to `a` are viewed as a potential conflict even though no conflict will occur due to the loop stride. As a result, the following loop would not be vectorized:

```
for (i=0; i < 128; i+=2){  
    a[i] = a[i-1];  
}
```

- Verifying that there are no possible aliases between array or pointer references. If there are possible aliases, the dependency tests may miss potential conflicts, making it unsafe to vectorize the loop.

General Optimizations

The optimizations collected in this section cover a variety of different domains and methods. General optimizations share two core features:

1. They are only enabled when one of the following optimization strategies are enabled: **-Omoredebug**, **-Odebug**, **-Osize**, **-Ogeneral**, **-Ospeed**.
2. When enabled as part of the above strategies, they should almost never be selectively disabled. For more information, see below.

When You Should Consider General Optimizations

Because general optimizations can only be enabled by certain optimization strategies, their use should be determined by the appropriateness of the strategies that imply them.

Unless explicitly stated otherwise, general optimizations should only be selectively disabled if their operation reveals an intractable bug in your code, or idiosyncrasies in how your code was constructed. The latter case may arise if you are working with code that was engineered using a variety of non-standard coding techniques.

For information about options to disable or modify these and other optimizations, see “Optimization Options” on page 286.

Peephole Optimization

Peephole optimization identifies common code patterns and replaces them with more efficient code. This includes flow of control, algebraic simplifications, and the 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 Assembly Code	Optimized Assembly Code
<pre>mul x3, x0, x1 add x0, x3, x2</pre>	<pre>madd x0, x1, x0, x2</pre>

Pipeline Instruction Scheduling

Pipeline instruction scheduling reorders instructions to make more efficient use of the processor's pipeline. While the exact methods used will depend on your target's processor architecture, some techniques are commonly employed. One such technique attempts to eliminate pipeline stalls. A pipeline stall is created when the result of one instruction is unavailable to a later instruction that relies on it. In this instance, the pipeline can stall until the result of the first instruction becomes available. Another frequently used technique seeks to maximize the use of a processor's instruction dispatch and instruction execution units. In typical circumstances, a processor is only capable of dispatching one floating-point or one integer instruction per cycle. With this optimization, both can be applied simultaneously.

Both techniques can be seen in the example below.

	Before Instruction Scheduling	After Instruction Scheduling
1	<pre>sub x1, x1, x2</pre>	<pre>sub x1, x1, x2</pre>
2	<pre>add x0, x0, x1</pre>	<pre>fsub x5, x4, x5</pre>
3	<pre>sub x1, x12, x4</pre>	<pre>sub x1, x12, x4</pre>

	Before Instruction Scheduling	After Instruction Scheduling
4	<code>add x1, x3, x1</code>	<code>fsub d5, d1, d2</code>
5	<code>fsub d5, d4, d5</code>	<code>add r0, r0, r1</code>
6	<code>fadd d3, d3, d5</code>	<code>fadd d3, d3, d5</code>
7	<code>fsub d5, d1, d2</code>	<code>add r1, r3, r1</code>
8	<code>fadd d1, d0, d5</code>	<code>fadd d1, d0, d5</code>

In the original code, the instruction dispatch is limited since integer and floating point instructions are clumped together. Also, there are several pipeline stalls due to instructions requiring the result of a prior instruction. In the scheduled code, instruction dispatch is optimal since each integer instruction is followed immediately by a floating point instruction. Also, pipeline stalls are reduced by separating the instructions that compute a result from the instruction that use the results.

Common Subexpression Elimination

Common subexpression elimination 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
<pre>int subr(int x, int y, int a, int b) { int a, b; x += a+b; y += a+b; if (y < 0) return (y); return (x); }</pre>	<pre>int subr(int x, int y, int a, int b) { int a, b, _v6; x+=(_v6=a+b); y+=_v6; if (y<0) return (y); return (x); }</pre>

Tail Calls

Tail calls can assist in avoiding function frames that would otherwise be necessary. For example: a function `foo()` will be considered for tail call optimization if the last executed statement returns the value of a function call `bar()`. With this optimization, the compiler will branch from `foo()` to `bar()`, and `bar()` will return directly to `foo()`'s caller. Consider another example:

C Source Code	Initial Assembly Code	Optimized Assembly Code
<pre>int bar(void); int foo(void) { return bar(); }</pre>	<pre>str lr, [sp, #-16]! bl bar ldr lr, [sp], #16 ret lr</pre>	<pre>b bar</pre>

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
<pre>int sum(int n) { if (n <= 1) return 1; else return n + sum(n-1); }</pre>	<pre>int sum(int n) { int p = 0; top: if (n <= 1) return p + 1; else { p += n; n--; goto top; } }</pre>

Constant Propagation

Constant propagation replaces variables with constants in any instance where the compiler determines that the value of the variable does not change.

Initial C Source Code	Optimized C Source Code
<pre>int main() { int i, a, b; a = 3; b = 0; for (i = 0; i < 1000; i++) b += a; printf("%d\n", b); return 0; }</pre>	<pre>int main() { int i, b; b = 0; for (i = 0; i < 1000; i++) b += 3; printf("%d\n", b); return 0; }</pre>

C/C++ Minimum/Maximum Optimization

C/C++ minimum/maximum optimization generates special code for minimum, maximum, and absolute value expressions of the following form:

```
(i < j) ? i : j
(i > j) ? i : j
(i < 0) ? -i : i
```

Memory Optimization

Memory optimization increases the speed of 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 that 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 to explicitly declare any object that may change without the compiler's knowledge or control (see “Type Qualifiers” on page 597). If circumstances prevent the use of `volatile`, you can disable this optimization by passing **-Onomemory**.

Dead Code Elimination

Dead code elimination removes code that computes unused values and eliminates unreachable blocks. In the following example, the expression $0 * x$ is determined to have a constant value of 0, and is propagated into the `if` conditional. The compiler then determines that block 1 is unreachable and removes the call `foo()` and the statement `return c`. The compiler can finally determine that the value of variable `c` is no longer used, and thus eliminate the expression `c = 3 * x`.

Initial C Source Code	Optimized C Source Code
<pre>int subr(int x) { int a, b, c; a = 0 * x; b = 2 * x; c = 3 * x; if (a) { // block 1 foo(); return c; } else { // block 2 bar(); return b; } }</pre>	<pre>int subr(int x) { int b; b = 2 * x; bar(); return b; }</pre>

Static Address Elimination

Static address elimination assigns the address of frequently used global or static variables to registers within the scope of a given function. This reduces the overhead required to generate an instruction's address in the code. In the following example, the address of the variable `arr` is maintained in a register:

Initial C Source Code	Optimized C Source Code
<pre>int arr[20]; int subr1(int i) { arr[0] = i; return arr[i]; }</pre>	<pre>int arr[20]; int subr1(int i) { register int *tmpaddr = &arr[0]; tmpaddr[0] = i; return tmpaddr[i]; }</pre>

In another variation of this optimization, instead of maintaining a variable's address in a local register, the compiler caches the value instead. In the following example, the value of `x` is maintained in a register:

Initial C Source Code	Optimized C Source Code
<pre>int x = 0; int subr2(int q) { x++; q += x; return q; }</pre>	<pre>int x = 0; int subr(int q) { register int x_tmp = x; x_tmp++; q += x_tmp; x = x_tmp; return q; }</pre>

The above optimization is more challenging because the cached value is only guaranteed to be valid until another operation could access it through another means. As a result, this optimization is implemented as part of another optimization, such as common subexpression elimination (see “Common Subexpression Elimination” on page 352).

Chapter 5

The DoubleCheck Source Analysis Tool

Contents

Introduction	358
Enabling DoubleCheck	359
Using Custom Functions with DoubleCheck	361
Viewing DoubleCheck Reports	366
Controlling DoubleCheck Output	367
Source Analysis Error and Warning Messages	369

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 359), 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.



Note

Enabling DoubleCheck may cause minor differences in code generation.

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 configure DoubleCheck to generate a report file (**.gsr**), use the **-double_check.report=filename** option. 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 366 for additional report viewing methods.



Note

The report contains information about the most recent build of each individual source file. If you fix an error in a source file and rebuild it, 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.



Note

Cleaning a project does not cause DoubleCheck report files to be removed. Once the information for a particular source file is included in a report, it will persist until that source file is rebuilt. This is true even if the source file is removed from your project. This behavior is intended to minimize the risk that DoubleCheck diagnostics will be lost before the referenced issues are addressed in the source. To remove outdated information from a DoubleCheck report file, manually remove the file and recompile each source file that still contains relevant diagnostics.



Note

Specifying a DoubleCheck report file when building with **make** requires that the build be single-threaded. This is to avoid corruption, as each compilation unit appends data to the report file.

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” on page 361
- “Specifying Function Properties with Pragma Directives” on page 362
- “Property Types” on page 363

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 363 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 10, “Utility Programs” on page 473).

When naming a DoubleCheck configuration file, use the **.dcc** extension. Specify the configuration file to the compiler with the **-double_check.config=filename** option.

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 361.

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 361). 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=filename** 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 will not interrupt or terminate a build. If you build again without cleaning or modifying dependencies, compilation steps will be skipped, hence source analysis diagnostics will not be re-issued in the build output. These diagnostics will still be present in the report file, if one was requested via the **-double_check.report** build option. To get the most benefit out of DoubleCheck, we recommend that you fix all bugs found by DoubleCheck, 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 and Enabling Source Analysis Errors and Warnings

In certain cases it might be useful to ignore specific source analysis error and warning messages, or to enable only specific source analysis errors/warnings. For example, if certain parts of your project cannot be modified, you can ignore their associated source analysis errors and warnings with a build option. You can also ignore source analysis errors/warnings by inserting `#pragma` directives in your code.



Note

You might also want to disable source analysis errors or warnings 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 or Enable Source Analysis Errors and Warnings

To disable or enable a particular source analysis error or warning message for a file or part of a build hierarchy, use the **`-double_check.ignore=number`** option or the **`-double_check.enable=number`** option, where *number* is the number of the error or warning you want to ignore/enable. 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.

To disable all source analysis error and warning messages, use the **`-double_check.ignore=all`** option. To enable all source analysis error and warning messages, use the **`-double_check.enable=all`** option.

Later options will override earlier options. For example, passing `-double_check.ignore=all -double_check.enable=5` will disable all source analysis messages other than error #5. In a project hierarchy, this also allows the top level to specify some options that are overridden by lower levels. For example, the top level may specify `-double_check.ignore=1 -double_check.ignore=2`, while a lower level may undo this by passing `-double_check.enable=all`.

Since most messages are already enabled by default, note that passing them as an enable option on their own (with no ignore option beforehand) will have no visible effect.

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
```



Note

Using `#pragma` directives to enable source analysis errors/warnings is currently unsupported.

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 370 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 374.

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.

19: Potential signed integer overflow - check or cast operands



Note

This error is disabled by default. Enable with `-double_check.enable=19` or `-double_check.enable=all`.

DoubleCheck issues this error when an expression performs signed integer arithmetic, the operation is capable of overflow, and insufficient Boolean checks have been performed on the operands beforehand. A Boolean check on an operand is an expression of Boolean type that uses the operand, such as `x != INT_MIN` on the operand `x` in the case of the operation `-x`.

Boolean checks are usually required on both operands for binary arithmetic operations. (One exception is integer division, which only requires checking one operand.) Note that the check `x != 0` is not considered sufficient as it does not prevent overflow (which occurs in the case of `INT_MIN / -1`). Finally, be aware that this rule can trigger false positives when used with programs that `#include stdarg.h`.

Importance

Signed integer overflow is undefined behavior that may yield unexpected results. For example, the arithmetic may wrap or trigger an exception, or the compiler may generate object code that assumes overflow never occurs. The previous behaviors may also change unpredictably when code elsewhere in the project is changed.

Avoidance

Perform a proper check on the operand(s) prior to performing the operation:

```
int a;  
if (a != INT_MIN) {
```

```
my_func(-a);  
}
```

In a number of cases, if the operands are `ints`, casting them up to `long longs` also guarantees that overflow will not happen:

```
int a, b;  
long long c = (long long)a + b;
```

20: Potential unsigned integer overflow - check operands



Note

This error is disabled by default. Enable with **-double_check.enable=20** or **-double_check.enable=all**.

DoubleCheck issues this error when an expression performs unsigned integer arithmetic, the operation is capable of overflow, and either insufficient Boolean checks were performed on the operands beforehand or insufficient Boolean checks were performed afterward. A Boolean check on an operand is an expression of Boolean type that uses the operand. For example: `a >= b` in the case of the operation `a - b` (which counts as Boolean checks on both `a` and `b`).

When the checks come before the operation, they are required on both operands. When they come after the operation, the result of the operation must be stored and the checks must cover that variable and either of the two operands. However, if the operation is unsigned multiplication (`*`), both operands must still be covered. As an exception, the unsigned left shift operator (`<<`) is permitted to overflow due to common usage. Finally, be aware that this rule can trigger false positives when used with programs that `#include stdarg.h`.

Importance

Unsigned integer overflow causes wrap-around behavior, which can create serious logic errors later in a program. For example, a very large number may be unexpectedly used as an array index or size, or passed to a function like `malloc()`.

Avoidance

Perform a proper check on the operands prior to performing the operation:

```
unsigned int a, b;
if (a >= b) {
    my_func(a - b);
}
```

Or perform a proper check on one operand and the result after performing the operation:

```
unsigned int a, b, c;
c = a - b;
if (c > a) { /* Handle error */ }
```

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. Note the following caveats:

- A variable's initialization is not considered a write to that variable for the purposes of this warning.
- 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 asarm64 Assembler

Contents

Running the Assembler from the Builder or Driver	390
Running the Assembler Directly	391
Assembler Options	391
Assembler Syntax	392
Expressions	396
Labels	401
Reading the Compiler's Assembly Output	402

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 project tree, or to the driver's command line. If your assembly file ends with **.a64** 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 224). To pass the full range of assembler options listed in this chapter:



Enter the required assembler option (see “Assembler Options” on page 391) 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:

```
ccarm64 -asm=-l -asm=-w stack.s -c -o stack.o
ccarm64 -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=filename**).

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**:

```
ccarm64 -S main.c
```

See also the option **-keepasmfiles** in “Temporary Assembly Files” on page 270.

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:

```
asarm64 [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 390.

General Assembler Options

@filename
Passes commands listed in <i>filename</i> to the assembler.
-cpu=cpu
Specifies the target <i>cpu</i> .
-help
Prints a help message.
-I dir (an uppercase letter “i”)
Searches directory <i>dir</i> for files specified in <code>.include</code> directives.

-list=[filename]

Generates a source listing. If *filename* 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 *filename* is specified, then the listing is written to *filename*.

-o filename

Sets the name of the output object file to *filename*. 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. 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 397.

The following additional identifiers are built into ARM64 assembly language.

Identifier	Meaning
b0-b31	SIMD registers, 8-bit
d0-d31	Floating-point/SIMD registers, 64-bit
fp	Frame pointer (x29)
h0-h31	Floating-point/SIMD registers, 16-bit
ip0-ip1	Intra-procedure call scratch registers (x16-x17)
lr	Link register (x30)
q0-q31	SIMD registers, 128-bit
r0-r30	General purpose integer registers
s0-s31	Floating-point/SIMD registers, 32-bit
sp	Stack pointer
v0-v31	Floating-point/SIMD registers
w0-w30	General purpose integer registers, 32-bit
x0-x30	General purpose integer registers, 64-bit

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

Type	Prefix	Example
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 ASCII characters (other than newline) or character escape sequences enclosed in double quotation marks (`"`). 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 a single ASCII character (other than newline) or character escape sequence, enclosed within single quotation marks (`' '`).

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
<code>\0</code>	null	0 (0x0)

Escape Sequence	Character	ASCII value
\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 <i>nnn</i>	n/a
\xnn	hexadecimal value <i>nn</i>	n/a
\'	single quotation mark	39 (0x27)
\"	double quotation mark	34 (0x22)
\\	backslash	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 401.

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 “;” or “//”. The assembler ignores all characters to the right of the comment symbol until the end of the line.

Note that while the number symbol (#) can sometimes be used as a comment character, this usage may conflict with other syntax and is therefore deprecated. For # to be a comment character, it must be placed in the first column of a line. The use of # as a comment character does not work with preprocessed assembly files.

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 .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 400).

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).

Operator	Type	Meaning
!=	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 ().

C Type Information Operators

If you enable the **-asm3g** option (see “Support for C Type Information in Assembly Files” on page 226), you can import type information from C header files using the `.inspect` directive and use it in the following operators:

Expression	Meaning
<code>offsetof(<i>type</i>, <i>ident</i>)</code>	The offset of <i>ident</i> from the base type of <i>type</i> in bytes, where <i>type</i> is a structure tag or a typedef identifier.

Expression	Meaning
<code>sizeof(type)</code>	The size of <i>type</i> in bytes, where <i>type</i> 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])
```



Note

Unlike `#include`, the result of `.inspect` is not visible to subsequent `.inspect` directives. A file processed with `.inspect` must resolve its own dependencies (potentially using its own `#include` directives) without relying on the context provided by prior `.inspect` directives.



Note

This capability is available only for assembly files. It will not function with different forms of inlined assembly in C or C++ source files.

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 <code>absolute</code> , 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 <code>\n</code> for newline, are allowed. These sequences are described in “Character Escape Sequences” on page 394.
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, **ccarm64**:

```
;Driver Command: ccarm64 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.

```
;f      %d8      local
;c      %w6      local
;.L12   .L15     static

;a      %x4      param
;b      %x5      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

Alignment Directives	406
Conditional Assembly Directives	407
Data Initialization Directives	408
File Inclusion Directives	409
Macro Definition Directives	409
Repeat Block Directives	412
Section Control Directives	413
Source Listing Directives	415
Symbol Definition Directives	415
Symbolic Debugging Directives	419
Miscellaneous Directives	420

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 **asarm64** assembler.

Alignment Directives

```
.align const-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
.ascii "if directive is TRUE"
.elseif x-6
.ascii "elseif directive is TRUE"
.else
.ascii "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.

<code>.else</code>	Assembles the subsequent block of assembly statements if the previous <code>.if</code> directives evaluate to false.
<code>.elseif <i>integral-const</i></code>	Assembles the subsequent block of assembly statements if the previous <code>.if</code> directives evaluate to false and the <code>.elseif <i>integral-const</i></code> evaluates to true. The assembler must be able to resolve <i>integral-const</i> to a constant without referring to information within the conditional block of code that follows the <code>.elseif</code> directive.
<code>.endif</code>	Terminates the code block that began with the previous <code>.if</code> directive.
<code>.if <i>integral-const</i></code>	Starts a conditional block of assembly statements; the assembler generates object code for these statements only if <i>integral-const</i> evaluates to true. The assembler must be able to resolve <i>integral-const</i> to a constant without referring to information within the <code>.if / .endif</code> conditional block of code.

Data Initialization Directives

The following directives evaluate expressions and place successive values of the specified type in the assembly output.

<code>.ascii "string"</code>	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.
<code>.asciiz "string"</code>	Evaluates a C-style string enclosed in double quotation marks (" ") and places the characters in successive bytes. Unlike the <code>.ascii</code> directive, the terminating null is placed with the string. The delimiting double quotation mark characters are discarded. Identical to <code>.string</code> .
<code>.byte const-expr, const-expr, ...</code>	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.
<code>.double flt-expr, flt-expr, ...</code>	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.
<code>.dword const-expr, const-expr, ...</code>	Stores the values of constant expressions as successive 64-bit data. The value of each expression must be in the signed range (-2^{63}) to $(2^{63}) - 1$ or in the unsigned range 0 to $(2^{64}) - 1$.
<code>.float flt-expr, flt-expr, ...</code>	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.
<code>.half const-expr, const-expr, ...</code>	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.
<code>.space const-expr</code>	Generates <i>const-expr</i> bytes of zero data.
<code>.string "string"</code>	Evaluates a C-style string enclosed in double quotation marks (" ") and places the characters in successive bytes. Unlike the <code>.ascii</code> directive, the terminating null is placed with the string. The delimiting double quotation mark characters are discarded. Identical to <code>.asciiz</code> .

```
.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 **-Idirectory** 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*. Macros also have parameters and every parameter must be specified. (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.)

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.

Example 7.1. Basic Macro Usage

The following example defines two macros `trythis` and `log_and`:

```
.macro trythis parm1
ldrsh x1, [x4,parm1*2]
.endm
.macro log_and parm1 parm2
ldrsh parm1, [x4,parm2]
.endm
```

For example, to call 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 x1, 0
log_and x1, 2
```

The assembler performs macro expansion and generates the following:

```
.text
ldrsh x1, [x4, 32]
ldrsh x1, [x4, 0]
ldrsh x1, [x4, 2]
```

The following example concatenates the values of `val1` and `val2` to make a label:

```
.macro make_word_point_to_itself val1, val2
val1 >< val2:
.word val1 >< val2
.endm

.data
.space 20
make_word_point_to_itself label, 15
```

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 7.2. 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
b.lt label
mov x0, 1
b.gt label2
```

```
mov x0, 0
b label2
label:
movn x0, 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.

Example 7.3. Conditional Termination of Macro Expansion

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. For example:

```
.macro bits mask depth
.if (mask == 0)
.exitm

.elseif (depth >= 16)
.warning "mask exceeds 16 bits"
.exitm
.endif
.if (mask & 1)
.byte depth
.endif
bits (mask >> 1), (depth + 1)
.endm

bits 0x1234, 0
```

The above macro will output a byte of data for each bit set in a 16-bit mask, using `.exitm` to control recursion and range checking. In the example invocation, it will output 5 bytes of data: 2, 4, 5, 9, and 12.

Example 7.4. Using a Simplified Macro

A simplified macro is available via `.define`, which uses only one line and no parameters. A `.define` macro replaces each instance of a symbol with the text it is defined to. Using `.undef` for the same symbol will cancel this and evaluate it as the original symbol.

```
.define foo 1, 2, 3, 4
.word foo, foo, foo, foo
.undef foo
.word foo, foo, foo, foo
```

The above example results in twenty word-sized data elements. The first sixteen are the sequence 1, 2, 3, 4 repeated four times. The final four are relocations to the external symbol `foo`.

<code>.define name [contents]</code>	Replaces all instances of <i>name</i> with the <i>contents</i> it is defined to.
<code>.endm</code>	Terminates the body of a macro whose definition began with a preceding <code>.macro</code> directive. The <code>.endm</code> directive must be the first symbol on its line and cannot have a label.
<code>.exitm</code>	Causes the assembler to exit a macro without further expanding its body of assembly statements. Use within a <code>.if / .endif</code> block to specify a condition under which you do not want a macro expanded beyond that point.
<code>.macro name [param1, param2,...]</code>	Begins the definition of a macro, where <i>name</i> is a string representing the name of the macro and <i>param1</i> is the first parameter that the macro accepts. Parameters must be separated by a comma or whitespace.
<code>.macrolocal name</code>	Declares a temporary label. This directive must appear immediately following the <code>.macro</code> directive, before any non-macro header directives. The actual label may be defined anywhere in the body of the macro.
<code>.undef name</code>	Cancels a prior <code>.define</code> and parses <i>name</i> normally.

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.

<code>.endr</code>	Terminates a repeat block that began with a preceding <code>.rept <i>const-expr</i></code> directive. The <code>.endr</code> directive must be the first symbol on its line and cannot have a label.
<code>.rept <i>integral-const</i></code>	Begins a block of assembly statements that repeats <i>integral-const</i> times, where <i>integral-const</i> resolves to a constant number.

Section Control Directives

These directives place assembly output in the specified section.

<code>.data</code>	Changes the active section to the <code>.data</code> section. Assembly output that follows this directive is placed in the <code>.data</code> section.
<code>.note "string" [, "a b w x"]</code>	Changes the active section to an ELF note section. The new section is called <i>string</i> . The attribute letters a, b, w, and x function as they do with the <code>.section</code> directive.
<code>.org <i>const-expr</i></code>	Creates an absolutely located data section at address <i>const-expr</i> . The name of the section is the section's address, with a prefix of <code>.org</code> . The MULTI Debugger cannot debug code placed at an address with a <code>.org</code> directive. Therefore, we recommend that you use the <code>> <i>const-expr</i></code> of the <code>.section</code> directive described below.
<code>.previous</code>	Changes the active section back to the one in use before the most recent section control directive.
<code>.rodata</code>	Changes the active section to the <code>.rodata</code> section, which is used for read-only data. Assembly output that follows this directive is placed in the <code>.rodata</code> section.

```
.section "string"[,"a|b|w|x"][ > const-expr]
```

Places 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 output 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:

```
ccarm64 file.s -list
```

<code>.eject</code>	Puts a form feed (^L) into the source listing.
<code>.gen</code>	Causes macros following this directive to be included in the source listing.
<code>.list</code>	Cancels the .nolist directive (re-enables source listing).
<code>.nogen</code>	Prevents macros in the sections following this directive from being included in the source listing.
<code>.nolist</code>	Causes the assembler to exclude information from the source listing until a corresponding <code>.list</code> directive is encountered.
<code>.sbtbl "string"</code> <code>.subtitle "string"</code>	Inserts <i>string</i> as the subtitle at the top of each page in the source listing. The double quotation marks are discarded.
<code>.title "string"</code>	Inserts <i>string</i> as the title at the top of each page in the source listing. The double quotation marks are discarded.

Symbol Definition Directives

The following directives define the type and value of an identifier.

<code>.comm ident, const-expr[, const-expr]</code>
Assigns the specified identifier <i>ident</i> to a common area of <i>const-expr</i> bytes in length, and causes <i>ident</i> to be visible externally. If <i>ident</i> is not defined by another relocatable object file, the linker assigns space for the identifier in the <code>.bss</code> section. The optional <i>const-expr</i> specifies the variable alignment in bytes.
<code>.dsect</code>
Begins a block for defining constants. Within a <code>.dsect</code> / <code>.end</code> block, only labels and <code>.space</code> 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 <code>.dsect</code> block. No space is actually allocated; only the symbolic names in the label fields are defined.
<code>.end</code>
Terminates a block of constants that began with a preceding <code>.dsect</code> directive. Defining this directive outside of a <code>.dsect</code> directive may result in unexpected behavior.
<code>ident .equ const-expr</code>
Defines the constant value <i>const-expr</i> to the symbol <i>ident</i> . After a symbol is defined with a <code>.equ</code> directive, it cannot be redefined.
<code>.export ident</code>
Causes the identifier <i>ident</i> 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.
<code>.global ident</code> Same as <code>.export</code> .
<code>.import ident</code> Same as <code>.export</code> . You usually use this directive when you want to explicitly declare <i>ident</i> as defined in another module. The assembler does not require this usage, however.
<code>.inspect "file.h"</code>
Parses the C header file <i>file.h</i> to get type information. You can use this information in the <code>.struct</code> directive, and in the <code>offsetof()</code> and <code>sizeof()</code> operators (see “C Type Information Operators” on page 398). This directive is available only when the -asm3g option is enabled (see “Support for C Type Information in Assembly Files” on page 226) and only for assembly files. It will not function with different forms of inlined assembly in C or C++ source files.
<code>.lcomm ident, const-expr[, const-expr]</code>
Assigns the specified identifier <i>ident</i> to an area in <code>.bss</code> of <i>const-expr</i> bytes in length. The optional <i>const-expr</i> specifies the variable alignment in bytes. The <code>.lcomm</code> directive is similar to the <code>.comm</code> directive, except that <i>ident</i> is not exported.

```
.lssbss ident, const-expr [, const-expr]
```

Assigns the specified identifier *ident* to an area in `.lssbss` of *const-expr* bytes in length. The optional *const-expr* specifies the variable alignment in bytes. The `.lssbss` directive is similar to `.lssbss`, except that *ident* is not exported.

```
.lssbss 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 `.lssbss` 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.

```
.struct type ident
.struct type ident = {const-expr,...}
.struct type ident[const-expr]
.struct type ident[] = {const-expr,...}
```

Allocates a block of data based on the structure definition for *type*, and gives it the identifier *ident*. Use the `.inspect` directive to load structure definitions.

You can initialize values using field names, or using C-style initialization, where constant expressions are assigned to fields in the order you declare them. If you do not pass any initialization values, then the block is initialized to zero.

After defining an object with `.struct`, you can access the object's fields using C-style pointer references (`->`).

For example, if the C header file **t1.h** contains:

```
struct myStruct {
    int i;
    struct {
        char n_char;
        char n_chararray[5];
    } nested;
};
```

To allocate a block of data based on `myStruct` and initialize it to zero:

```
.struct myStruct empty
.struct myStruct empty_array[5]
```

To initialize the data to specific values:

```
;C-Style initialization
.struct myStruct s = {1, {'x', {'h','e','l','l','o'}}}
;Initialization using field names
.struct myStruct s2 = { i = 1,
    nested = { n_char = 'x',
        n_chararray = {'h','e','l','l','o'}
    }
}
```

You can also initialize data when using `.struct` to define an array:

```
;Allocates two blocks, and initializes the fields in the second one
.struct myStruct s_array[] = {{}, {1, {'x', {'h','e','l','l','o'}}}}
```

To use the object's fields in other instructions:

```
;Write the address of n_chararray in s to array_address
array_address:
    .word s->nested->n_chararray
```

```
.weak ident
```

Declares *ident* as an external weak symbol. If the *ident* cannot be located in any module at link time, the linker sets the symbol's value to zero.

Symbolic Debugging Directives

These directives are used for symbolic debugging.

```
.endf ident
```

Marks the end of a function that sets both the type and size of the function symbol. The `.endf` directive is equivalent to using the `.type` and `.size` directives individually. Specifically, `.endf ident` is equivalent to:

```
.type ident, $function  
.size ident, .-ident
```

```
.endo ident
```

Marks the end of a data object that sets both the type and size of the object symbol. The `.endo` directive is equivalent to using the `.type` and `.size` directives individually. Specifically, `.endo ident` is equivalent to:

```
.type ident, $object  
.size ident, .-ident
```

```
.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 function *func* is *const* bytes in size. This information maybe useful in certain situations when debugging call stacks involving assembly functions. You must place this directive within the scope of *func*, before you define the next function.

```
.maxinstances [func,] const
```

Specifies the maximum number of instances *const* of the function *func* that can exist on the stack at a given time. This information is used to provide **gstack** with an upper bound on possible recursions so it can accurately compute the maximum stack usage of the program. 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. For more information, see “Annotating Recursive Functions” on page 550.

```
.maxstack func, const
```

Specifies that *func* 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.

```
.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 to get the best debugging information. Your assembly code must properly annotate the types and sizes of symbols when link-time optimizations are enabled. See also the `.endf` and `.endo` directives which may be used in place of `.size` and `.type`.

```
.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. Your assembly code must properly annotate the types and sizes of symbols when link-time optimizations are enabled.

Miscellaneous Directives

```
.assert integral-const
```

Gives an error if *integral-const* evaluates to false.


```
.error "string"
```

Causes the error "string" to be emitted to standard error output.

```
.ident "string"
```

Adds *string* to the `.comment` section within the object file. The `.comment` section is not allocated; 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`.

```
.warning "string"
```

Causes the warning "string" to be emitted to standard error output.

Chapter 8

The elxr Linker

Contents

Linker-Specific Options	424
Specifying the Program Entry Point	429
Configuring the Linker with Linker Directives Files	429
Symbol Definitions	455
Advanced Linker Features	460

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 429).

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 227. 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 232 for more information about this option.

It is never necessary to invoke the linker directly; always specify linker-specific options through the driver.

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 241).
-allow_ambiguous_abs <i>symbol</i>
Specifies the name of a symbol to exclude from the error check enabled by -ambiguous_absolute_error . Symbols should be specified using their mangled names. Mangled names can be determined by using gnm or --no_link_filter option. For more information, see “The gnm Utility Program” on page 520 and “Demangling of Names in Linker Messages” on page 312.
-allow_bad_relocations
Emits only warnings when encountering bad or unknown relocations. The default behavior is to generate errors.
-ambiguous_absolute_error
When importing symbols via the -A driver option, gives an error for symbols that can be imported from multiple import files (see “Import Symbols” on page 241). Symbols with the same address in different files are given a warning instead. (Note that symbols imported from INTEGRITY POSIX DLLs are not included in these checks.)
-any_p_align
Does not set an upper bound on the value of the <code>p_align</code> field in ELF program headers.
-max_p_align = <i>n</i>
Limits the maximum value of the <code>p_align</code> field set in ELF program headers. Values of <i>n</i> must be positive, integral powers of 2, expressed in bytes. The <code>p_align</code> field is set based on the maximum alignment requirement of sections contained by ELF headers, as long as the requirement does not exceed <code>max_p_align</code> .
-min_p_align = <i>n</i>
Specifies the lower bound of the <code>p_align</code> field set in ELF program headers. Values of <i>n</i> must be positive, integral powers of 2, expressed in bytes. The <code>p_align</code> field may be larger, depending on the maximum alignment requirement of sections contained by headers.
-no_p_align
Always sets the <code>p_align</code> 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.

-Dsymbol=value
Defines a global <i>symbol</i> with an address of <i>value</i> in the ELF symbol table. See also “Define Linker Constant” on page 240, an option that defines a linker constant that does not appear in the symbol table.
-earlyabsolute
Imports absolute symbols before searching libraries. For use with the -A driver option (see “Import Symbols” on page 241).
-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 241).
-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 function <i>symbol</i> when you specify the -delete option..
-keep_from=name
Prevents the deletion of all functions defined in section <i>name</i> when you specify the -delete option..
-mapfile_type=n
Controls the output format of symbols in the map file. Valid options are: • -mapfile_type=1 — [default] Standard MULTI format. • -mapfile_type=2 — Extended format with additional details in global and local symbol tables. • -mapfile_type=3 — Like -mapfile_type=2, except <code>MEMORY</code> is omitted for symbols not in a <code>MEMORY</code> region.
-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 443.

-no_map_renaming

Does not include master section names for remapped sections in map file output.

-no_uncompressed_copy

By default, the linker creates a notes section called `.UNCOMPRESSEDsecname` 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.

-old_section_wildcard_matching

Reverts the wildcard matching of linker directives files for `SECTIONS` to the behavior of Compiler 2018.5 or earlier. When wildcard patterns for two different output sections both match the same input section, there is ambiguity regarding section placement. Generally, a more specific rule will take precedence over a less specific rule, with order of appearance in the file used as a tiebreaker. The newer behavior makes more subtle distinctions in wildcard pattern specificity, which can result in ambiguous linker directive rules being resolved differently between versions. The new behavior can also be significantly faster for linking large executables with complex directive files. Projects written using earlier versions that relied on prior wildcard rules may use this compatibility option while transitioning between versions.

-only_data_absolute

Gives an error if a symbol from an executable section is imported via the `-A` driver option (see “Import Symbols” on page 241).

-prog2

Produces exactly two segments: one to contain sections `.text` and `.rodata`, and the other to contain `.data` and `.bss`.

-retainlocals

By default, the linker strips most local module symbols from the ELF output, retaining only global symbols in the new symbol table. If you pass this option, local symbol names remain a part of the final image.

-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 <code>.debug</code> sections to have a 1-byte alignment.
-underscore
Adds an underscore to the beginning of all symbols created at link time.
-unweak <i>symbol</i>
If any undefined weak reference to <i>symbol</i> 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 <i>sym</i>
Resolves external references to function <i>sym</i> to <code>__wrap_sym</code> , and external references to <code>__real_sym</code> to <i>sym</i> . 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: <pre>void critical_section(void);</pre> you can check the calls to <code>critical_section()</code> by passing -wrap <code>critical_section</code> to the linker, and then define the following: <pre>void __wrap_critical_section(void) { assert(is_safe_to_call_critical_section()); __real_critical_section(); }</pre> Because the modifications performed by -wrap will not be visible to the compiler, you should not use this option if you are compiling your code with interprocedural optimizations (-Ointerproc or -Owholeprogram).

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 430
- “Setting Linker Options with the OPTION Directive” on page 430
- “Setting Defaults with the DEFAULTS Directive” on page 431
- “Defining a Memory Map with the MEMORY Directive” on page 432
- “Defining a Section Map with the SECTIONS Directive” on page 434
- “Modifying your Section Map for Speed or Size” on page 450
- “Customizing the Run-Time Environment Program Sections” on page 451

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 60.

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 451).



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 arithmetic 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 424 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 8.1. 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 identifiers 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 240).

Example 8.2. 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:

<i>memory_block</i>
Specifies the name of a memory block.
<i>origin_expression</i>
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 8.3. 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 `0xbfc00000`. 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,... | >> memory_block,... | >. | >>.]
  ...
}
```

where:

<i>secname</i>
Specifies name of the section.
<i>start_expression</i>
Specifies the starting address of the section. If you specify both <i>start_expression</i> and <i>memory_block</i> , <i>start_expression</i> takes precedence.
<i>attributes</i>
Specifies section attributes. You can use expressions in attribute parameters. For a list of section attributes, see “Using Section Attributes” on page 443.
{ <i>contents</i> }
Specifies the contents of the section. It contains <i>section inclusion commands</i> 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. If more than one memory block is specified, the section will be allocated to the first listed block that can accommodate the section in its entirety (see “Splitting a Section Across Multiple Memory Blocks” on page 437).

```
>> memory_block,... | >>.
```

Specifies multiple memory blocks for section allocation. If necessary, parts of the section may be allocated to discontinuous memory blocks. For more information, see “Splitting a Section Across Multiple Memory Blocks” on page 437.

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 442.

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 8.4. 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 using `>` syntax, and the section does not fit in that memory block, the linker will return an error. If you specify multiple memory blocks, the linker will place the section in the first specified memory block with enough remaining space to hold it (see Example 8.4. Writing a *SECTIONS* Directive on page 436).

Splitting a Section Across Multiple Memory Blocks

If you specify multiple memory blocks for a section using the `>>` syntax, the linker may split and allocate section contents into different memory blocks. If so, there will be multiple versions of the section in the output executable, one for each memory block used. Note, however, that there are many reasons why the linker may fail to split a section in this way, including interactions with other features that assume a single, contiguous memory region, such as ROM to RAM copy. If the linker is unable to split a section, it will behave as though the `>` syntax were used and allocate the entire section to one of the listed memory blocks. If this results in the section failing to fit in the memory map, the linker will output an error message with this information. If not splitting affects memory allocation in some way, it will output a warning with the reason.

Common reasons that a section may not be split include expressions in the section contents (other than section inclusion rules) and references to the section with the symbols `__ghsbegin`, `__ghsend`, or `__ghssize`. Additionally, the linker will not split up individual input sections originating from the same source file and section. Different input sections will be distributed across multiple versions of the output section in separate memory blocks. For finer-grained section splitting, consider compiling with **`-individual_function_sections`** or **`-individual_data_sections`** for smaller input sections.



Note

Split sections will not have a checksum with the **`-checksum`** feature.

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 full paths to libraries or modules; use only their (case-sensitive) filenames.

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 :
```

By default, the linker will place sections matched by wildcards in the order they are seen during linking. You can change this behavior by using the `SORT` or `SORT_BY_NAME` keyword on *section_names* with wildcards. When using the `SORT` or `SORT_BY_NAME` keyword, the linker will sort the sections into ascending order by section name prior to placement.

```
// SORT all sections matching ".bss.*" before placing into .mybss
.mybss :{ "*(SORT(.bss.*))" }
```

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) }
```

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 8.5. 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; *(<code>.text</code>) beta = .; *(<code>.text2</code>) 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.

<code>absolute(expr)</code>
Given a section-relative offset value, <code>absolute</code> returns the absolute address by adding the address of the containing section to <code>expr</code> . It is an error to use <code>absolute</code> outside of the braces that specify a section's contents.
<code>addr(sectname)</code>
Returns the memory address of the section or memory block <code>sectname</code> .
<code>align(value)</code>
Returns the current position (.) aligned to a <code>value</code> -byte boundary. <code>value</code> must be a power of two. This is equivalent to: <code>(. + value - 1) & ~(value - 1)</code>
<code>alignof(sectname)</code>
Returns the alignment of the section <code>sectname</code> .
<code>endaddr(sectname)</code>
Returns the memory address of the current end of the section <code>sectname</code> , or the end of the memory block <code>sectname</code> .
<code>error("string")</code>
Generates a linker error, displaying <code>string</code> , the current section's name and address, and the current section offset.
<code>final(finalexpression [,earlyexpression=0])</code>
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.
<code>isdefined(symbol)</code>
Returns 1 if <code>symbol</code> exists and is defined, 0 otherwise.
<code>isweak(symbol)</code>
Returns 1 if <code>symbol</code> is a weak global symbol, 0 otherwise.
<code>memaddr(sectname)</code>
See <code>addr</code> .
<code>memendaddr(sectname)</code>
See <code>endaddr</code> .

```
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 8.6. 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(<i>n</i>)
Aligns the start address of the section to the byte boundary given by <i>n</i> .
AT(<i>expr</i>) LOAD(<i>expr</i>)
These attributes specify an expression <i>expr</i> 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 463). CLEAR is implicitly specified on the .bss section. If you direct the linker to include COMMON data in any other section, it will also default to CLEAR. All other sections default to NOCLEAR. For suggested uses, see Example 8.8. Using CLEAR and NOCLEAR on page 447.
FILL(<i>value</i>)
Initializes the section to byte pattern <i>value</i> at run-time by making an entry in the run-time clear table (see “Run-Time Clear and Copy Tables” on page 463). 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. Only the least significant byte of <i>value</i> is used.
MAX_ENDADDRESS(<i>address</i>)
Causes the linker to generate an error if the section extends beyond <i>address</i> during final layout. The end address of a section is the start address plus the section size. In other words, 1 byte beyond the last legal address.
MAX_SIZE(<i>size</i>)
Causes the linker to generate an error if the section exceeds <i>size</i> bytes in length.
MIN_ENDADDRESS(<i>address</i>)
Instructs the linker to pad with zeros, if necessary, to ensure that the section extends to at least <i>address</i> .

<code>MIN_SIZE(<i>size</i>)</code> Instructs the linker to pad with zeros, if necessary, to ensure that the section is at least <i>size</i> bytes in length.
<code>MONOTONIC</code> <code>NOMONOTONIC</code> Sets or removes the <code>MONOTONIC</code> attribute of a section. If a <code>MONOTONIC</code> section has multiple memory blocks specified, the linker will not assign the section to a block in the section's block list if any sections are already assigned to a block that appears later in the list. The linker also will not assign subsequent sections to a block that appears in the <code>MONOTONIC</code> section's block list before the block that the <code>MONOTONIC</code> section was assigned to. This attribute provides a mechanism to prevent reordering sections that may be assigned to the same memory blocks from the way that they appear in the <code>.ld</code> file. You may use <code>-monotonic_block_order</code> or <code>-nomonotonic_block_order</code> to set a default behavior for sections without either attribute. For more information, see “Monotonic Block Order” on page 318. Note that neither attribute has an affect on sections for which fewer than two memory blocks are specified.
<code>NOCHECKSUM</code> Instructs the linker to output the section without a checksum. This overrides the effect of the <code>-checksum</code> option for an individual section.
<code>NOLOAD</code> Instructs the linker to output the section as <code>SHT_NOBITS</code>. The linker allocates the section into the target image, but discards its contents.
<code>PAD(<i>value</i>)</code> Places <i>value</i> bytes of padding (in zeros) 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: <pre>.stack pad(0x10000) : {} .stack : { . += 0x10000; }</pre>
<code>RANDOMIZE</code> Specifies that the section should be randomly reordered. Certain special sections (such as DWARF debug sections) cannot be marked <code>RANDOMIZE</code> because their function relies on maintaining an order consistent with other sections. For more information, see “Address Space Layout Randomization” on page 464.

```
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 449.

`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 884.

Example 8.7. 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 8.8. 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 463.

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 :
```

`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`. `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 463.

When defining a new section with the `ROM` or `CROM` attributes, the `ROM` or `CROM` attribute must specify the name of the existing section.

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 424). Sections marked as CROM are still copied to ROM but are not compressed.

**Note**

Compressed ROM sections do not have a checksum generated with the **-checksum** option.

Example 8.9. 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 449).

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 451 and “Run-Time Clear and Copy Tables” on page 463.

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 must generally copy `.data` to RAM, the Green Hills compiler will use the `.rodata` section for variables that do not change at run time. The `.rodata` section should be placed in the link map adjacent to the program `.text` section. This will cause the data to be put into ROM along with the program text.

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 825. 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 sections are not required for targets that do not support position independent code (PIC) and position independent data (PID). However, Green Hills recommends including **.fixaddr** and **.fixtype** in the linker directives file for completeness.

.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.

ROM Sections

When the linker encounters a section directive that has the `ROM` attribute, it creates a read-only copy of the section. The Green Hills startup code, `ind crt0.c` (see “`ind crt0.c`” on page 828), copies the data from the `.ROM.sect` section to the `.sect` section when copying initialized data from ROM to RAM (see also “Copying a Section from ROM to RAM at Startup” on page 449).

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 the target supports the SDA optimization.

.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.a64** 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.a64** to remove its dependency on the `.syscall` section (see “`ind_call.a64`, `ind_dots.a64`” on page 830).

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 `__ghsbeginsection`, `__ghsendsection`, and `__ghssizesection`, 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:

```
#include <stdio.h>  
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]) ();
    }
}
```

The linker also recognizes `__ghstableauto_tablename` and `__ghstablealways_tablename`, which behave similarly to `__ghstable_tablename`, except with regard to whether a `__ghsentry_tablename` symbol from a library will be pulled into a program that contains a corresponding `__ghstable`.

A `__ghsentry_tablename` symbol in a library will not be pulled in solely because the program contains a corresponding `__ghstable_tablename` symbol. It will be pulled in from a library if that table is `__ghstableauto_tablename` and an entry of the same name has not already been defined elsewhere. It will be pulled in from a library if the table is `__ghstablealways_tablename` regardless of whether an identical entry has been defined (possibly resulting in a multiply defined symbol error message).

Forcing the Linker to Pull in a Module From a Library

Normally, the linker only pulls in an object file from a library when that file provides a global definition for an unresolved, non-weak symbol. You can use the **-extractall** and **-extractweak** options to change this behavior. However, you may want the linker to only pull in particular object files from the library. For this reason, the linker searches each library for global symbols beginning with certain prefixes, which may modify the default behavior for an individual library module.

If it finds an object file that defines a symbol beginning with `__ghsautoimport`, the linker will pull in the file unless the symbol is already defined. In contrast, if the linker finds an object file that defines a symbol beginning with `__ghsalwaysimport`, the linker will pull in the file regardless of whether it is already defined.

Note that bringing in a symbol that is already defined will result in an error unless the symbol is common. As such, it is advisable to use different names in each file for global symbols beginning with `__ghsalwaysimport` or `__ghsautoimport`. For more information about common symbols, see “C/C++ Data Allocation” on page 208.

For unresolved, weak references, the linker recognizes the prefixes `__ghs_extract_weak_` and `__ghs_override_weak_`. If an object file defines a symbol of the form `__ghs_extract_weak_symname`, the linker will pull in the object file from the library if there are unresolved, weak references to `symname`.

`__ghs_override_weak_symname` triggers the same behavior, while also pulling in the object file if there is an extant weak definition of `symname`. The linker will then override `symname` with a strong definition.



Note

When using `__ghs_extract_weak_` and `__ghs_override_weak_` prefixes in a C++ module, the mangled form of `symname` must be specified with the prefix for the linker to resolve references to the definition. The mangled name can be found in the assembly, list, and object files.

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.

The linker supports **-delete** for object files produced by the Green Hills toolchain. Other toolchains may produce object files that do not preserve symbol types, sizes, or relocations required for **-delete** to function properly.



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.



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 linker never deletes symbols preserved by these methods, even if the symbols appear unused. It is useful to preserve symbols such as interrupt entry points, non-standard entry-points, or other portions of code not reached by normal control transfer within the program.

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.

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 312).

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 238 for more information.

In addition to the **-delete** option, **-uvfd** enables the deletion of C++ virtual functions. **-uvfd** requires that all modules being linked are compiled with Green Hills compilers. We recommend that you enable this option with the **-Olink** flag, rather than specify **-uvfd** explicitly (see Linker Optimizations in “Optimization Options” on page 119).

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 ARM64 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 of 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 8.10. Basic Code Factoring

To factor code from source files **foo.c** and **bar.c**, use:

```
ccarm64 -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 8.11. 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 8.12. 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:

```
ccarm64 -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:

```
ccarm64 -codefactor mod.o -o hello
```

This command produces the code-factored executable, **hello**.

Example 8.13. 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:

```
ccarm64 -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:

```
ccarm64 -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:

```
ccarm64 -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

- 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 752.
- 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. If you plan to include the assembly file in your project, with modified section contents or build options, you should remove the directives entirely.

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.

Address Space Layout Randomization

The linker supports address space layout randomization (ASLR) for improved security. During link time, ASLR will randomly reorder input sections assigned to an output section that has been marked with the `RANDOMIZE` attribute. This differs from the default behavior, wherein functions or variables from a given source file are maintained in the same text or data section in the same relative order.

To use ASLR, assign the attribute `RANDOMIZE` to indicate that a given section's contents should be reordered, then pass **--aslr_seed=number** to seed the randomization process. For more information, see “Address Space Layout Randomization Seed” on page 314 and the `RANDOMIZE` attribute in “Using Section Attributes” on page 443.

To increase the effectiveness of ASLR, consider using it in conjunction with **-individual_function_sections** or **-individual_data_sections** (see “Individual Variable Sections for Default Data Sections” on page 98 and “Individual Function Sections for Default Text Section” on page 100). Doing so will ensure that every function or variable from the same source file has its own section that can be reordered by ASLR.

Chapter 9

The ax Librarian

Contents

Creating a Library from the Compiler Driver	466
Modifying Libraries	467
Creating and Updating a Table of Contents	471
The Green Hills Large Archive Format	471

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:

```
ccarm64 filename ... -archive -o libname
```

where each *filename* is a source file or object file that you want to include in the library, and *libname* is the name of the library (**.a** file) 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* 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:

```
ccarm64 filename ... input_library ... -merge_archive -o output_library
```

-merge_archive is similar to **-archive**, with the following differences:

- If *output_library* exists, it is replaced.
- The entire contents of each specified *input_library* is added to the library (libraries specified with **-library** are ignored unless the option **--whole-archive** is enabled). The contents of each input debug library (**.dba**) are added to the output debug library.

-merge_archive is implemented by passing the **C** and **M** command modifiers to the archiver (see “Librarian Command Modifiers” on page 468).

If you need a 64-bit format archive, pass the **-large_archive** option.

Example 9.1. How to Create a Library

To create a library **libfoo.a** from the source file **foo.c**, enter:

```
ccarm64 foo.c -archive -o libfoo.a
```

To create a library **libhello.a** from the object files **hello.o** and **world.o**, enter:

```
ccarm64 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 468. Enter the command and modifiers as one string without spaces.

-d [<i>e</i>] [<i>S</i>] [<i>v</i>] <i>libraryfile memberfile</i> ...
Deletes each specified <i>memberfile</i> from the specified <i>libraryfile</i> .

-p [e] <i>libraryfile memberfile ...</i>
Prints each specified <i>memberfile</i> from the specified <i>libraryfile</i> to standard output (if the members are object files, then the output might not be human-readable).
-r [c] [C] [D] [e] [S] [n] [m] [M] [v] [u] <i>libraryfile memberfile ...</i>
Replaces each specified <i>memberfile</i> in the specified <i>libraryfile</i> . The new version of <i>memberfile</i> overwrites the existing version.
If <i>memberfile</i> does not already exist in <i>libraryfile</i> , the librarian adds it.
If <i>libraryfile</i> 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] <i>libraryfile ...</i>
Lists the names of the files contained in each specified <i>libraryfile</i> . Use the <i>v</i> option to list names, file sizes, and the dates that the members were last modified.
-x [e] [o] [v] <i>libraryfile memberfile ...</i>
Extracts each specified <i>memberfile</i> from the specified <i>libraryfile</i> into the current directory. The librarian does not alter <i>libraryfile</i> .
If you only specify <i>libraryfile</i> , 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 467 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.
D
Sets the UID, GID, and date to 0 and the file mode to a consistent value. (Use with -r .) This is useful for reproducible builds, as differences in file metadata across builds can create small differences in object files.
e
Prefixes warnings with <code>WARNING</code> and prefixes errors with <code>ERROR</code> . Equivalent to the option -prefixed_msgs recognized by some utilities.

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
See “Output Archives in Large Format” on page 318.
-quiet_auto_large_archive
See “Output Archives in Large Format” on page 318.
-large_archive
See “Output Archives in Large Format” on page 318.
-no_large_archive
See “Output Archives in Large Format” on page 318.
-reject_duplicates
This option can be used with options d or r without the S modifier, or it can be used with option t with the s modifier. See “Error on Duplicate Symbols in an Archive” on page 319.
-no_reject_duplicates
See “Error on Duplicate Symbols in an Archive” on page 319.

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 list of files in 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 Large Archive Format

The **ax** archiver supports a large archive format that allows archives and the individual files they contain to exceed 2 GB. For information about controlling the large archive format selection and warnings, see “Output Archives in Large Format” on page 318.

Chapter 10

Utility Programs

Contents

The gaddr2line Utility Program	475
The gasmlist Utility Program	476
The gbin2c Utility Program	478
The gbincmp Utility Program	486
The gbuild Utility Program	488
The gcolor Utility Program	499
The gcompare Utility Program	502
The gdump Utility Program	505
The gfile Utility Program	511
The gfunsize Utility Program	511
The ghide Utility Program	512
The gmemfile Utility Program	513
The gnm Utility Program	520
The gpatch Utility Program	526
The gpjmodify Utility Program	527
The gsize Utility Program	531
The gsrec Utility Program	532
The gstack Utility Program	539
The gstrip Utility Program	561
The gversion Utility Program	563
The gwhat Utility Program	564
The protrans Utility	567

The Green Hills toolchain includes many utilities, some of which serve as replacements for standard Linux utilities like **dump**, **file**, **hide**, **nm**, **size**, and **strip**.

- **gaddr2line**: Similar to the Linux **addr2line** program. Maps addresses to filenames and line numbers.
- **gasmlist**: Generates interlaced assembly and source output .
- **gbin2c**: Converts binary files into C array definitions.
- **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 the code sizes of matching functions in two files.
- **gdump**: Similar to the Linux **dump** program. Dumps or disassembles a file.
- **gfile**: Similar to the Linux **file** program. Describes the file type.
- **gfunsize**: Returns the code size of a function.
- **ghide**: Similar to the Linux **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 **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 **size** program. Displays section sizes.
- **gsrec**: Converts executable files 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 **strip** program. Removes symbol or debugging information from an executable.
- **gversion**: (for executable files) Returns version date and time information.
- **gwhat**: Reports or updates version information.
- **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 function 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 files. It supports fully linked ELF files with DWARF information and object (**.o**) files with either DWARF or Green Hills **.dbo** information.

The syntax for this utility is as follows:

gasmlist [*options*] *files*

where *files* is a whitespace-separated list of files and *options* listed in the table below.



Note

The **gasmlist** utility uses debug information to correlate an input file's assembly code with its original source file. If this information is out of date or incorrect, it may result in run-time errors or incorrect output. This can occur, for example, if a source file has been edited since the input file was built or if the source file contains incorrect `#line` directives.

--all

--all_with_asm

Controls which files have source lines printed. The default behavior is to only print source lines from each compilation unit's base source file.

If **--src_then_asm** is specified:

- **--all_with_asm** prints source lines for all files that produce assembly, in addition to source lines from base source files.
- **--all** prints source lines for all files.

If **--src_then_asm** is not specified, both **--all_with_asm** and **--all** print source lines for all files that produce assembly, in addition to source lines from base source files.

--file_num**--proc_num**

Restricts identification of output lines to one of the following:

- **--file_num** file line numbers
- **--proc_num** function line numbers

The default is to print both function and file line numbers with the Green Hills **.dbo** information. Function line numbers are not output if DWARF information is present.

--file_offset

Calculates offsets from the start of the present input 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 within assembly. 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 input 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 function line numbers are enabled, prints the function 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 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 457.

-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 <i>n</i>
Outputs a maximum of <i>n</i> bytes of the current module.
-module_name <i>name</i>
Refers to the current module as <i>name</i> in the module list.
-noconst
Removes <code>const</code> from the type of the array for the current module.
-signed
-unsigned
Adds <code>signed</code> or <code>unsigned</code> to the type of the array for the current module.
-skip <i>n</i>
Skips the first <i>n</i> bytes of the current module.
-static
Adds <code>static</code> to the type of the array for the current module.
-volatile
Adds <code>volatile</code> to the type of array for the current module.

Example 10.1. 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 10.2. 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 10.3. 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 10.4. 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** only applies to **mod.o** and **-signed** and **-array_name** only apply to **foo**. 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 10.5. 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 10.6. 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 ELF binary files. The two files can be object files, executable files, or archives.

If the contents of 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 to continue after initially detecting a difference. For each pair of sections compared between the input files, gbincmp will print a message if the names or sizes of the sections differ or if one of the sections has host file content while the other does not. Otherwise, gbincmp will print each difference it detects in the contents of the two sections.
-debug -nodebug Enables or disables comparison of the <code>.debug</code> (DWARF) and <code>.debug_info</code> (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 <code>.line</code> (DWARF) and <code>.debug_line</code> (DWARF2).
-normalize Ignores differences in encoded timestamps.
-prefixed_msgs Prefixes warnings with <code>WARNING</code> and prefixes errors with <code>ERROR</code> .
-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 <i>sections</i> list. For example, <code>-section=text,data</code> .
-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.
-allcfg
Builds the project in all the configurations defined in your Top Project (.gpj) file. For example, if a Debug configuration and a Release configuration are defined, this option builds both. If you do not specify this option or -cfg (see below), and your Top Project file defines build configurations, only the first configuration is built. For more information, see “The Header Section” on page 911.
-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 .

-cfg=*descriptive_name*

-cfg=*short_name*

Builds the project in the configuration specified by *descriptive_name* or *short_name*, where *descriptive_name* and *short_name* are the names defined for the configuration in the Top Project file. For example, given *descriptive_names* `Debug` and `Release`, **-cfg=Debug** builds the project in the `Debug` configuration, while **-cfg=Release** builds the project in the `Release` configuration.

If you do not specify this option or **-allcfg** (see above), and your Top Project file defines build configurations, only the first configuration is built.

For more information, see “The Header Section” on page 911.

-clean

Deletes all previous output files without building. Note that **gbuild** does not perform an implicit dependency analysis as part of this process. You must clean these dependencies separately, or clean a parent project. For more information, see “Implicit Dependency Analysis” on page 497.

-cleanexternals

Deletes all previous output files from the current project and from all **externalDepends** projects. This option is ignored if specified with a file.

-cleanfirst

Deletes all existing output files and then initiates a build. Note that **gbuild** does not perform an implicit dependency analysis as part of this process. You must clean these dependencies separately, or clean a parent project. For more information, see “Implicit Dependency Analysis” on page 497.

-commands

Displays commands before they are executed.

-Dmacro_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 910 for more information.

This option does not define a preprocessor symbol.

-file_library_ordering

Sets the ordering of file arguments on archive command lines to match project file ordering. (By default, such arguments are listed in reverse of project file ordering). Note that library file order impacts linker behavior when resolving identical symbols present in multiple files.

-find *filename*

Displays the first build path to the file *filename*. The build path specifies a route through the build structure to reach *filename*. If *filename* is not found in the current project, **externalDepends** projects (if any) are searched in order.

-findall <i>filename</i>
Displays all build paths to the file <i>filename</i> . The build paths specify routes through the build structure to reach <i>filename</i> . This option only displays build paths for the current project.
-fullexternals
Builds everything in the current project and all externalDepends projects. This option is ignored if specified with a file.
-H
Displays <code>#includes</code> (see “Display includes Preprocessor Directives Listing” on page 140).
--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.
-no_rebuild_for_opt_change
Prevents rebuilding multiply instanced files (see Example 10.13. Managing Multiple File Instances in a Project on page 497).
-nochecklibs
Disables Implicit Dependency Analysis. For more information, see “Implicit Dependency Analysis” on page 497.
-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.
-noprogess
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.
- `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 274).

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 10.11. Building One of Multiple Projects with the Same Name on page 496.

-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 timing information for individual child processes in the build. This includes elapsed time, user time (marked with <code>u</code>), and system time (marked with <code>s</code>) for each process. To display timing information for the gbuild executable, pass -time .
-stderr=filename
Redirects stderr output (status and error messages) to <i>filename</i> .
-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.
-strict_config
Forces a build error if output from a build will overwrite previously built output from a different build configuration. This option can be very helpful in verifying that outputs from different build configurations do not overlap.
-time
Displays basic summary timing information. This includes elapsed time, user time (marked with <code>u</code>) and system time (marked with <code>s</code>). On Linux hosts, the user and system times are shown in CPU-seconds for the gbuild process and all of its children. For parallel builds using multiple cores, the number of CPU-seconds used may be larger than the elapsed time. On Windows hosts, the user and system times are the times for the gbuild process only.
-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 <code>/</code> .
-verbose
Prints additional information during the build.
-within intermediate.gpj
Restricts searching for the specified <i>file</i> arguments to children of <i>intermediate.gpj</i> . 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 10.11. Building One of Multiple Projects with the Same Name on page 496.

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 10.7. 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 10.8. 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 10.9. 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 10.10. 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 10.11. 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 10.12. 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
```

Example 10.13. Managing Multiple File Instances in a Project

Building a project containing more than one instance of the same file may cause the file to be built multiple times, with **gbuild** stating "final command has changed". To stop **gbuild** from rebuilding the file, make sure each instance has identical build options or outputs to a different directory. (For example, by using **-object_dir** or **:outputDir**.)

To prevent **gbuild** from rebuilding all multiply instanced files, pass **-no_rebuild_for_opt_change** on the **gbuild** command line. Alternately, add the flag to the **defaultBuildOptions** directive in your top project. For example:

```
defaultBuildOptions=-no_rebuild_for_opt_change
```

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 example, because they are system libraries) are ignored.

When searching for dependencies, **gbuild** looks for children of the following items and builds the first instance of the implicit dependency it finds:

- 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.

If your project depends on code built in an external project tree, you can extend the implicit dependency analysis search to that external project tree by specifying an **externalDepends** directive in the header of your Top Project (see “Group 1 Directives” on page 914). External projects are searched in the order specified by **externalDepends** and are only searched if dependencies are not found in the Top Project of the program being analyzed. When searching external projects, **gbuild** looks in:

- Project and Subproject **.gpj** files that are not descendants of a buildable item.

Using **externalDepends** is especially recommended for INTEGRITY users whose applications depend on source code in the INTEGRITY project tree.

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 914. Disabling IDA may degrade parallel build

performance. For more information, see the documentation for **-parallel_level** in “The gbuild Utility Program” on page 488.

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 <i>file</i> . 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
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	Colors		Attributes	
	Value	Background or Foreground	Value	Attribute
info	-1	Default	-1	Default
warning	0	Black	0	None
error	1	Red	1	Bold
before	2	Green	4	Underscore
after	3	Yellow	5	Blink
	4	Blue	7	Reverse
	5	Magenta	8	Concealed
	6	Cyan		
	7	White		

Comment lines begin with `#`, or whitespace followed by `#`.

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-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 begin with `#`, or whitespace followed by `#`.

Example 10.14. 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 **cs** shells is different. If you are using a **cs** 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.

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), **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%
```

If **-i** is specified, **gcompare** will reverse its behavior and print results where the new number is better rather than worse.

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:

-l
Produces a report with comparisons sorted from worst to best. All comparisons (including those in which <i>newfile</i> is no larger than <i>oldfile</i>) are shown. In addition, a summary is provided at the top, showing the aggregate size comparison between <i>newfile</i> and <i>oldfile</i> .
-a
Prints comparisons even when items exist in only one file.
-help
Displays information about all options.

-i	Prints lines where the new number is better, rather than worse, when not producing a full report.
-ignore <i>name</i>	Suppresses comparisons of symbol <i>name</i> .
-L	Formats the report in 132-column (landscape) mode. The default is 80-column (portrait) mode.
-min=<i>n</i>	Suppresses comparisons where the difference between the compared items is less than <i>n</i> 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 -l , but with additional diagnostics.
-w	Suppresses warnings.
-width=<i>n</i>	Sets width (<i>n</i>) to any value greater than 32.
-x <i>cmd</i>	Specifies the command to execute on an input file recognized as an object, object library, or executable file. The default -x command is: <code>-x "gfunsize -gcompare -all"</code>
-z	Does not show cases where files are the same.

If the **-r**, **-l**, 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%	
-----						-----				
epsilon	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%		epsilon	20	20	0%

This output can require many columns. Use the **-L** option for a 132-column format. On Linux, 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 **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.

-dinfo

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 function 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.

-no_trunc_sec_names

Displays full section names with **-map**, instead of truncating to 20 characters.

-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=<i>name</i>
Displays the contents of the specified section only.
-sx -nx
Enables (-sx) or disables (-nx) attempted shorter C++ demangling.
-sym
Uses symbol names instead of numbers in relocation output.
-v1 -v2
Specifies display of DWARF Version 1 or 2. Otherwise, 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.
-yinfo -ninfo
Enables or disables the display of particular information, where <i>info</i> 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: function linkage table • s: symbol table information • sec: section contents • seg: segment 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 (<i>name</i>).
-d num [+d num2]
Displays section <i>num</i> or sections from <i>num</i> to <i>num2</i> .
-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 <i>num</i> or symbols from <i>num</i> to <i>num2</i> .
-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 **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.

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.
-nunderscores
Removes leading underscores from function names (on architectures where function symbols have a leading underscore).
The following options are mutually exclusive:
• -sect=name Displays functions in section <i>name</i>. • -sectnum=n Displays functions in section number <i>n</i>.
-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.

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:

-arm64_kernel

Checks for the existence of an ARM64 header inside an input ELF image, and (if present) adds the binary size to the header. This allows boot loaders, such as U-Boot, to use a custom text offset value described in the header when loading the image to RAM during the boot.

-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 **-fillx** is passed. You can specify a section name instead of an address, and the highest address of that section is used.

-fillx *start end value*

Fills holes before, after, or between sections of the executable in the specified range *start* to *end* with the specified *value*. If the input file is a COFF file, its byte order (big or little endian) is applied to the fill value. If the input file is an ELF file, the fill value uses a big endian order, regardless of the endianness of the input file.

-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 <i>value</i> 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 <i>sname</i> of the executable from the memory image.
-start address Specifies the <i>address</i> 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 10.20. Separating Low and High Bytes of Data into Separate PROMs on page 518 and Example 10.21. Separating the Low Half and High Half of Each 4-byte Interval on page 519).

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 10.15. 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 10.16. 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 10.17. Using the `-fill` Option

```
gmemfile prog1.elf -fill 0x3e7f .text 0x3e7f -o prog1.mem
```

In this example, the `-fill` 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 `-fill`. 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 `-fill` starts at zero).

Example 10.18. 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 0xffff) 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 0xffff and 0x1600 to 0x1fff are filled with zeros.

Example 10.19. 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 10.20. 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 10.21. 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` (`0x1000`) 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` (`0x1000`) 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 10.22. 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 **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 <i>file</i>
Writes the output from gnm to <i>file</i> .
-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 <code>..bof</code> , <code>..eof</code> , and <code>..lin</code> .
-no_dotdot	Does not print any symbols beginning with “ <code>..</code> ”.
-no_line	Does not print line debug symbols. For example, <code>..lin</code> .
-o	Prints value and size of symbols in octal.
-p	Uses 3-column output format.
-prefixed_msgs	Prefixes warnings with <code>WARNING</code> and prefixes errors with <code>ERROR</code> .
-r	Prefixes filename to each symbol name.
-R	Prefixes archive name and filename to each symbol name.
-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 (<code>.Ln</code>).
-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).

Column	Meaning
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

Type	Meaning
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	External informational symbol
n	Local 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 patches 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 [options] *patch_filenames*

The following table describes the most common command line options supported by **gpatch**.

-decrypt_key_file <i>key_file_name</i>
Decrypts <i>key_file_name</i> and uses the license keys within to decrypt encrypted patch files.
-help
Displays documentation for all gpatch command line options.

-key <i>installation_key</i>
Use <i>installation_key</i> as the patch decryption key or, if <i>key_file_name</i> is specified, the keyfile password.
-list
Lists the files contained in the archive. This does not install the patch.
-nouninstall
Does not create an uninstallation archive.
-target_dir <i>directory</i>
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.jff** without installing the patch:

```
> gpatch -list patch_1234.jff
```

To install **patch_2028.jff** into **C:\ghs\multi_xxx**:

```
gpatch -target_dir C:\ghs\multi_xxx patch_2028.iff
```

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:

-cfg= <i>configname</i>
Specifies a configuration <i>configname</i> , where <i>configname</i> is the descriptive or short name of a configuration definition. Note that setting this option when modifying files or other options does not cause the modifications to be configuration-specific.
-f
Forces the edit, even if errors would result.
-help
Displays help information.
-o <i>output.gpj</i>
Writes the edited file to <i>output.gpj</i> . By default, edits are written to the existing file.
-quiet
Suppresses warning messages.
-toolsdir <i>directory</i>
Specifies an alternative MULTI Compiler installation <i>directory</i> .

- *action* listed in the following table:

-add_after <i>child</i> [-filetype <i>type</i>] <i>files</i> [-sourcedir <i>dir_paths</i>]
Inserts <i>files</i> in the project's list of children, directly after <i>child</i> .
-add_before <i>child</i> [-filetype <i>type</i>] <i>files</i> [-sourcedir <i>dir_paths</i>]
Inserts <i>files</i> in the project's list of children, directly before <i>child</i> .
-add_customization_file <i>filename</i>
Adds a customization file to the Top Project.

-add_to_back [-filetype <i>type</i>] <i>files</i> [-sourcedir <i>dir_paths</i>]
Inserts <i>files</i> at the end of the project's list of children.
-add_to_front [-filetype <i>type</i>] <i>files</i> [-sourcedir <i>dir_paths</i>]
Inserts <i>files</i> at the beginning of the project's list of children.
-change_macro <i>name=value</i>
Changes the value of the existing macro <i>name</i> to <i>value</i> .
-make_tree_canonical
Canonicalizes filenames used in the project. This option applies to the entire project tree, not just the specified project file.
-modify <i>old_option new_option</i>
Replaces all instances of <i>old_option</i> in the specified project with <i>new_option</i> .
-remove_files
Deletes the specified <i>files</i> from the project's list of children.
-remove_customization_file <i>filename</i>
Removes the specified customization file from the Top Project.
-replace_with <i>existing_child</i> [-filetype <i>type</i>] <i>new_child</i>
Replaces the references to the file <i>existing_child</i> with references to <i>new_child</i> .
-set <i>options</i>
Inserts a list of build <i>options</i> .
-set_target <i>target_file</i>
Specifies the project's target file. This is only saved if you are modifying a Top Project file.
-unset <i>options</i>
Deletes a list of build <i>options</i> .

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 923. 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 <i>directory</i>
Specifies an alternative MULTI Compiler installation <i>directory</i> .
-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 488). 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 <i>directory</i>
Specifies an alternative MULTI Compiler installation <i>directory</i> .

**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 **gsiz**e utility program performs a similar function to the Linux **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, **gsiz**e 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 **gsiz**e are as follows.

-all
Causes all sections to be displayed, including empty sections and non-allocated sections. Note that this option does not imply -count_commons .
-commons
Displays a block of information describing each <code>COMMON</code> symbol in each output file (see “Text and Data Placement” on page 61). 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 <code>.bss</code> and the size of small-common variables in the total size of <code>.sbss</code> . This option has no effect if you pass the -nobss option.
-details — [default] -nodetails Displays or suppresses detailed information about each section in each file.
-help Displays information about all options.
-nobss Suppresses <code>bss</code> sections. This option makes -commons and -count_commons have no effect.
-nodata Suppresses <code>data</code> sections.
-notext Suppresses <code>text</code> sections.
-table Displays output in table format. This option implies -details and -nototals .
-text Only displays <code>text</code> sections, suppressing <code>data</code> and <code>bss</code> 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 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.

The syntax for this utility is as follows:

gsrec [*options*] *filenames*

where *filename* is the name of an input executable file to be converted to one of the output formats. Motorola S-Records is the default output format.



Note

The *s0*, *s5*, and *s7* records are not output if there are no *s3*, *s4*, or *s5* data records.

The **gsrec** *options* are:

-allow_overlap
Allows sections to overlap. By default with more than one executable file a fatal error will be given if any non-empty sections overlap.
-allow_relocated_sections
Allows sections with a non-empty matching <i>.rel</i> or <i>.rela</i> 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 <i>n</i>
Sets the maximum number of data bytes per record, not including the length, address, checksum or other special fields. The minimum value for <i>n</i> is 4 and the maximum is 248. For S-Record output, the default is 28. For Intel hexadecimal output, the default is 32.
-e <i>addr</i>
Sets entry point in the termination record to given address.
-end <i>address</i>
Specifies the highest <i>address</i> of data to be output.
-eol <i>char-seq</i>
Specifies the character sequence used to terminate each S-Record, where <i>char-seq</i> is one of:
• <i>cr</i>: a <i>\r</i> character. • <i>crlf</i>: a <i>\r\n</i> combination. This is the default in Windows. • <i>lf</i>: a <i>\n</i> character. This is the default in Linux.

-fillx n1 n2 v
Fills unused memory from address <i>n1</i> to address <i>n2</i> with the <i>x</i> -byte value <i>v</i> (where <i>x</i> is 1, 2, or 4).
-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.
-interval n[:m]
Places only those data bytes from the input file that occur within the specified interval in the output file. Values for <i>n</i> are 1 through 8. The default is 1, indicating that every byte will be output. The <i>m</i> 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 <i>string</i> 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.

-padx *n1 n2 v*

This option is identical to **-fill**.

-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

Prefixes warnings with `WARNING` and prefixes errors with `ERROR`.

-raw

Treats input files as raw data to be placed at the address specified by **-romaddr..**

-romaddr *addr*

Sets start address in ROM to place data. The default is the same address as in the input file.

-Stype

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.

-sec *s*

Only outputs data for section *s*. When repeated, sections used in prior invocations of **-sec** are included in output.

-skip *s*

Does not output data for section *s*. May be repeated.

-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 <i>address</i>
Specifies the lowest <i>address</i> 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 00E280FF04001FE80848074006384036000006360000B20580FF0000 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 10.23. 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 10.24. 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 10.25. 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 10.26. 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 10.27. 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 10.28. 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 10.29. 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 556.

**Note**

The **gstack** utility is designed only to compute the maximum possible stack usage of a program. It does not account for other programs that may also use the stack on the same target.

**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

The Green Hills toolchain produces debugging information that allows **gstack** to analyze the call graph of a program and determine the maximum amount of stack it needs. While this information should be sufficient for most code, **gstack** may require you to provide additional information in 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 132).
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 541.

- **MS** (missing stack) — An assembly function for which no stack information has been provided. See “Annotating Assembly Functions for *gstack*” on page 541.
 - **IC** (incomplete call) — A C function that requires additional call information. See “Annotating C Functions for *gstack*” on page 542.
 - **IS** (incomplete stack) — A C function that requires additional stack information. See “Annotating C Functions for *gstack*” on page 542.
4. If the **gstack** report contains an `Unknown Function Pointers` section, use strong function pointers to remove entries from the list. While not required to compute stack usage, this step can increase accuracy and eliminate or simplify recursive clusters. For more information, see “Using Strong Function Pointers” on page 545.
 5. If the report contains `Unbounded Recursive Clusters` or `Bounded Recursive Clusters` sections, attempt to break up as many such clusters as possible. For more information, see “Working With Recursive Clusters” on page 544.
 6. If the report lists unbounded recursive clusters that cannot be eliminated, place an upper limit on their recursion by specifying maximum instances values for the listed functions. For more information, see “Annotating Recursive Functions” on page 550.
 7. 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 552.
 8. 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 static_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

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.



Note

`#pragma` directives cannot be used to annotate C++ lambda functions.

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.

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 545).
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 549).



Note

If you are unable to modify the code, the **-a** and **-d** command line options can be used to manually change the call graph within **gstack**.

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 550).

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 646).

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 the `Unknown Function Pointers` section. To disable this behavior and other warnings related to unknown function pointers, pass the `--ignore_unknown_func_ptr_warnings` option to **gstack**.

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);

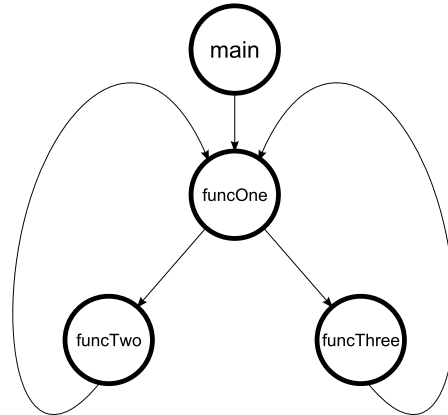
int funcThree(int x)
{
    return 1 + funcOne(x-1);
}

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);
}

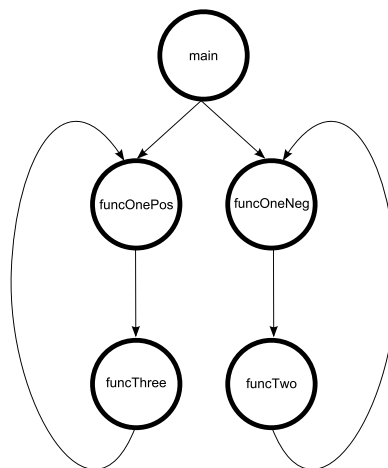
int funcTwo(int x)
{
    return 1 + funcOneNeg(x+1);
}

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 Clusters



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 recursive cluster containing the function in the `Unbounded Recursive Clusters` section. When each cluster is printed, the maximum instances value for each function is printed in parentheses after the function's name. Maximum instances values that are unknown are printed as `???` and must be provided before **gstack**

can accurately compute stack usage. To provide this information, add the following directive immediately before the definition of the function if it is defined in a C file:

```
#pragma ghs max_instances(n)
```

or add the following directive within the scope of the function if it is defined in an assembly file:

```
.maxinstances func, n
```

or use the following command line option:

```
-max_instances func=n
```



Note

The `.maxinstances` directive is ignored unless at least one `.scall` directive is specified for the same function.

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 544).

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 a 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. This section 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 results
Table of Contents	always	a list of the remaining sections of the report
Missing Information	gstack detects missing information	details about what information is missing
Unknown Function Pointers	gstack detects the use of an unknown function pointer	a list of the functions that call through unknown function pointers as well as strong function pointer types with assigned unknown function pointers
Unbounded Recursive Clusters	gstack detects unbounded recursive clusters	a list of unbounded recursive clusters
Miscellaneous Problems	gstack detects problems that could cause its results to be incorrect or conservative	an explanation of each problem
Stack Usage For Entry Point: <i>Name</i>	one time for each specified entry point	the worst stack usage, worst possible call stack, and worst interrupt call stack for the entry point
Bounded Recursive Clusters	gstack detects bounded recursive clusters	a list of bounded recursive clusters
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

Name	Appears When	Contains
Call Graph	-g is specified	the program's call graph starting from each specified entry point, followed by the call graph for each remaining root node (nodes without parents in the call graph), followed by the call graph for the <<address_taken>> node (if present), followed by the call graph for any unprinted node
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
User Supplied Information	gstack detects user-supplied information	details about what information was supplied by the user
Command Line	always	the command line used to invoke gstack

Using a *gstack* Options File

When using multiple **gstack** arguments, you may find that your command lines become overly long. To address this issue, you can pass most arguments to **gstack** using a plain text file and the *@filename* option. A file passed using this option will be parsed according to the following rules:

- All characters other than spaces, tabs, newline characters, #, and double quotes will be treated as literal.
- Arguments are separated by spaces, tabs, or newline characters.
- The first double quote (") on a line begins a quoted argument if the following conditions are true:
 - It is the first character on the line or follows a space or tab.
 - The last character on the line is also a double quote.

If these criteria are met, all characters on the line between the first and last double quote are literal, including # and ".

- If # is at the beginning of a line, the entire line is a comment.
- If # follows a space or tab and is not quoted as described above, the # begins a comment that continues to the end of the line.

gstack Options

-a <i>caller:callee1[,callee2]...</i>	Adds connections to the call graph. -a <i>caller:callee1, callee2</i> adds calls from <i>caller</i> to <i>callee1</i> and <i>callee2</i> to the call graph.
-c <i>func</i>	Displays all callers of <i>func</i> .
-constant_stack_usage <i>size</i>	Ignores the analyzed per-function stack usage and assumes a constant usage <i>size</i> (in bytes) for each function. This option is intended to be used on a separate invocation of gstack to calculate the maximum size of a shadow stack (which uses a constant number of bytes per function call). For more information, see “Shadow Stack” on page 931.
-d <i>callee:[caller,]...</i>	If you specify one function, this option deletes all calls to that function. For example, -d <i>callee</i> deletes all calls to <i>callee</i>. If you also specify a list of callers, this option deletes all calls from those callers to the specified callee. For example, -d <i>callee:caller1, caller2</i> deletes all calls from <i>caller1</i> and <i>caller2</i> to <i>callee</i>.
-f <i>func=size</i>	Specifies function maximum stack usage <i>size</i> .
-g	Displays the call graph.
-help	Displays information about all options.
-entry_point <i>func</i> , -e <i>func</i>	Specifies a function <i>func</i> 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 <i>entry opt1 [opt2]... -+</i> , -e_opts <i>entry opt1 [opt2]... -+</i>	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.

<i>@filename, -opt_file=filename</i>	Specifies a file containing additional options. For more information, see “Using a gstack Options File” on page 555.
--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 <i>func</i> 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 <i>func</i> .
-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.



Warning

This option is not recommended. In this mode, **gstack** does not take into account recursive functions or calls through function pointers along with other limitations. As a consequence, the maximum stack usage of the program will

be larger than the size reported by **gstack** in most cases.



Note

The option is deprecated and will be removed in a future release.

--link_names	Prints the function's global symbol, as opposed to its address, in parentheses when the function's name is not unique. See “Specifying Functions as Arguments to gstack ” on page 559.
-max_instances func=n	Specifies the maximum number of instances <i>n</i> of the function <i>func</i> 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 <i>n</i> to add to the known maximum stack value for function <i>func</i> . This option is the equivalent of <code>#pragma ghs extra_stack</code> .

Specifying Strong Function Pointer Types as Arguments to **gstack**

The **-a**, **-d**, and **-c** options may also be applied to nodes that represent strong function pointer types and the `<<address_taken>>` node that represents unknown function pointers. To specify a strong function pointer type, use the name of the type as it appears in the source. (Note that this is not the special `"(name *)"` syntax that **gstack** uses to distinguish strong function pointer types from functions.) If it is unclear whether the specified name refers to a function or strong function pointer type (because both are present in the program), **gstack** will issue an error. To resolve

the ambiguity, prefix the name with either `.f.` (for function node) or `.t.` (for type node). To refer to the `<<address_taken>>` node, specify "`<<address_taken>>`".

Specifying Functions as Arguments to gstack

Functions passed to **gstack** must be formatted identically to their **gstack** report listings. For example, if **gstack** lists a C++ function as `foo(int)`, the correct way to specify this function as a program entry point is `-entry_point foo(int)`, not `-entry_point foo`.

An important exception is when a program has non-unique function names, such as identically named static functions defined in different source files. When **gstack** prints the name of a non-unique function, it appends a parenthetical value. This value is either a memory address, or—if `--link_names` is used—the function's associated global symbol.

To pass non-unique functions as arguments to **gstack**, use only their parenthetical values (addresses or global symbols) rather than their function names.

The Miscellaneous Problems 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.

CodeFactor detected

This warning is given when **gstack** detects a function created due to code factoring, a linker optimization that is not supported with **gstack**. To disable this optimization, see “Code Factoring” on page 237. Note that code factoring can cause inaccuracies in **gstack** reporting without triggering this warning.

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.

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 link-time trivial function inlining.
- **gstack** does not support code that calls through an unknown function pointer whose value is derived from a hard-coded address. The caller will be treated as performing an unknown function pointer call, but the <<address_taken>> node will not necessarily have a reference to each of the call's possible destinations.
- Using Green Hills C++ libraries may result in errors about missing information and unbounded recursive clusters, or warnings about unknown function pointers.
- **gstack** is not supported for programs that use linker generated tables whose entries contain function pointers, unless each table is an array of function pointers or an array of pointers to function pointers. Tables of things like structs that contain function pointers are not supported, even if the fields of the structs are defined using strong function pointer types. Linker generated tables that are defined in or populated with entries that are defined in assembly files are not supported. For more information, see “Linker Generated Tables” on page 457.
- **gstack** is not supported for C/C++ programs that use `std::thread`, C threads, or thread-local storage.
- Some forms of exception handling may cause unbounded recursion errors to appear in the exception handling run-time. Specifically, this can occur if your program uses exception handling and destroys an exception object from within the destructor of another exception object.

The gstrip Utility Program

The **gstrip** utility program can remove DWARF line number, DWARF debugging, and symbol table information from an executable, relocatable, or library 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.

The **gstrip** utility processes executable files, relocatable object files, and libraries created by the Green Hills Software toolchain. The **gstrip** utility can also process some files made with non-GHS tools.

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, debugging information, the `.linfix` section, and the `.rominfo` section. The *options* for **gstrip** are as follows:

-a	Removes all unallocated sections (for which the <code>SHF_ALLOC</code> flag is not set) except <code>.shstrtab</code> .
-help	Displays information about all the options.
-l	Removes line number information only, without stripping the symbol table or debugging information.
-r=secname	Removes sections named <i>secname</i> , except allocated sections (for which the <code>SHF_ALLOC</code> flag is set) and <code>.shstrtab</code> .
-stripreloc	Allows stripping of relocatable objects and libraries. Without this option, using gstrip on relocatable objects or libraries will result in an error. If no other option is specified, this option also causes gstrip to remove debugging and line information, as well as the <code>.linfix</code> and <code>.rominfo</code> sections. The <code>.symtab</code> and <code>.strtab</code> sections are never removed when this option is used.
-V	Displays the version number of gstrip to standard error output.
-x	Removes debugging and line number information, without stripping the symbol table.

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 10.30. Using gversion on page 564). **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 10.31. Using -all on page 564.
-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 10.32. Using -mtime on page 564.
-quiet
Suppresses errors for files that are not stamped.
-value
Displays all stamp values without converting to date.

Example 10.30. 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 10.31. 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 10.32. 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 reports or updates the version information of a file based on what-strings, as described below.

The syntax for this utility is as follows:

gwhat [*options*] [*file1*] [*file2* ...]

The options to **gwhat** are as follows:

-all
Prints out all <code>what</code> -strings.
-c <i>what_str</i>
Overwrites the default <code>what</code> -string space reserved for customers with <i>what_str</i> .
-f
Overwrites the <code>what</code> -string, even if it is read-only.
-help
Displays help message.
-m <i>what_str</i>
Overwrites the Green Hills <code>what</code> -string space with <i>what_str</i> .
-s<i>n</i>
Stops after the <i>n</i> <i>th</i> occurrence of the pattern. The default is 1 (first occurrence). There is no space between the -s and <i>n</i> .
-w <i>marker</i>
Changes the <code>what</code> -string marker from <code>@ (#)</code> to <i>marker</i> .

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

Character	Character name
\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' ccarm64
gwhat -s2 ccarm64
    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' ccarm64
gwhat -s2 ccarm64
      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 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.
-b filename	Specifies <i>filename</i> as a bmon.out file containing coverage analysis profiling data. You can specify multiple files with multiple uses of this option.
-b64 filename	Specifies <i>filename</i> 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 filename	Specifies output <i>filename</i> as a .gpro profile recording that contains debug information and profile information.
-cc filename	Specifies <i>filename</i> as a raw memory dump of the .ghcovcz section containing coverage counts. You can specify multiple files with multiple uses of this option.
-cd filename	Specifies <i>filename</i> 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 filename	Specifies <i>filename</i> as a raw memory dump of the .ghcovfz section containing coverage flags. You can specify multiple files with multiple uses of this option.

Option	Meaning
-ccdir <i>dir</i>	Specifies <i>dir</i> as a directory that contains one or more raw memory dumps of the <code>.ghcovcz</code> section containing coverage counts.
-cddir <i>dir</i>	Specifies <i>dir</i> as a directory that contains one or more raw memory dumps of the <code>.ghcovdz</code> section containing 64-bit coverage counts.
-cfdir <i>dir</i>	Specifies <i>dir</i> as a directory that contains one or more raw memory dumps of the <code>.ghcovfz</code> section containing coverage flags.
-d <i>filename</i>	Specifies <i>filename</i> as a <code>.dbp</code> 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.

Example 10.33. Running the protrans Utility

Consider a big endian ARM64 program called **dylan** along with a collection of sample input files: **sampleinput1**, **sampleinput2**, and so on.

We intend to run and process coverage data for each test case. When using the **-coverage** option, coverage data is not dumped automatically at exit unless you are developing for Windows or Linux native. In order to dump the coverage data at exit, add code similar to the following to the end of your program (the example is for **-coverage=flag**):

```
extern char __ghsbegin_ghcovfz[];
extern char __ghsend_ghcovfz[];

FILE *f = fopen("ghcovfz.out", "wb");
if(f) {
    fwrite(__ghsbegin_ghcovfz, 1, __ghsend_ghcovfz-__ghsbegin_ghcovfz, f);
    fclose(f);
}
```

For information about dumping a memory section, see “Beginning, End, and Size of Section Symbols” on page 455.

Note that because the coverage dump happens during program execution, you will not get coverage data for the `fclose()` call or any further program execution.

Now consider the following **cs**h shell script for Linux hosts:

```
#!/bin/csh
foreach p (sampleinput*)
    simarm dylan $p
    protrans -a -B -q -cf ghcovfz.out dylan
end
rm -f ghcovfz.out
```

This script repeatedly runs the **dylan** program with the sample input using **simarm64**, 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 containing the summary profile. If the intermediate file has no extension, it is assigned a **.pro** 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 505.

The last line in the preceding script deletes any remaining profiling data files after executions of the profiled program.

You can also specify other specific data files to **protrans**.

Thus, the shell script example for Linux hosts that is given earlier could be rewritten to:

```
#!/bin/csh
mkdir coverage_dumps
foreach p (sampleinput*)
    simarm dylan $p
    mv ghcovfz.out coverage_dumps/ghcovfz.$p
end
protrans -B -q -cfdir coverage_dumps dylan
rm coverage_dumps/ghcovfz*
```

The first loop runs **dylan** with the sample input and stores the generated data files into uniquely named temporary files. The second loop calls **protrans** to read in the data from these files and produce a summary profile. The **-cfdir** option indicates that all files in the directory should be processed as memory dump files.

Part III

Language Reference

Chapter 11

Green Hills C

Contents

Specifying a C Language Dialect	576
ANSI C	577
ISO C99	583
ISO C11, ISO C17, and ISO C18	584
GNU C	588
Japanese Automotive C Extensions	595
Sequence Related Undefined Behavior Checking	596
Type Qualifiers	597
Assignment and Comparisons on struct and union Types	600
Bitfields	601
Enumerated Types	604
Functions with Variable Arguments	605
The asm Statement	606
The Preprocessor	607
Extended Characters	608
Compiler Limitations	611
C Implementation-Defined Features	613
Non-Conformance with C Standards	634
Attributes	637
Floating-Point Behavior	652
C Cautions	662

This chapter describes the Green Hills implementation of C, including aspects of the language particular to our compilers.

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 settings, see “C Language Dialect” on page 142.

Several dialects are available, depending on which OS you are using for your project:

- ANSI C “ANSI C” on page 577).
- ISO C99 (see “ISO C99” on page 583).
- ISO C11, ISO C17, and ISO C18 (see “ISO C11, ISO C17, and ISO C18” on page 584).
- GNU C (see “GNU C” on page 588).

In addition to supporting the aforementioned primary C dialects, we also provide support for:

- The Motor Industry Software Reliability Association (MISRA) C rules (see “MISRA C” on page 147)
- The Japanese Automotive C extensions (see “Japanese Automotive C Extensions” on page 595)



Note

Unless stated otherwise, the C dialect does not affect non-C source files.



Note

If **-act_like** is set to 2015.1 or earlier, the compiler will default to ANSI (**-ansi**).

All dialects use the same library and header files, which provide 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.

ANSI C

The ANSI C dialect conforms to *X3.159–1989*, including Amendment 1 and Technical Corrigenda 1 and 2. The ANSI C standard is the foundation of all supported C dialects, with each adding its own extensions or features on top of the basic ANSI language. If your program is written in standard ANSI C, any other dialect mode should compile your program correctly with few exceptions.

To specify use this dialect:



Set the **C/C++ Compiler→C Language Dialect** option to one of the following:

- **ANSI C (-ansi)** — ANSI C with extensions accepted.
- **Strict ANSI C (-ANSI)** — ANSI C with most extensions rejected.

In **Strict ANSI C** mode, the compiler will give errors when most non-standard language constructs are used. A few extensions to the standard are permitted, such as the `long long` type, which can be disabled with `--no_long_long`.

C Extensions for ANSI C

Below is an incomplete list of the extensions that are allowed with **-ansi**. Most of these extensions are also allowed in C99 or later (though often accompanied by warnings for non-standard feature use).

- 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, with the following caveats:
 - If the identifier begins with \$ and is for an object of non-automatic storage duration or a function, then:
 1. The identifier must not overlap with reserved identifiers from the target assembly language.
 2. The target assembly language must not use \$ as an operator.
- 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. This feature is only enabled in ANSI C mode.
- 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. 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 17, “Enhanced `asm` Macro Facility for C and C++” on page 857.

- Anonymous unions are allowed as the members of structs and unions, but not as freestanding variables. Likewise, anonymous structs 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. This is not allowed in either ISO C99 or Strict ISO C99 mode. 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. For a complete list of supported attributes, see “Attributes” on page 637.
- 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 123), or by using `#pragma ghs start_externally_visible` and `#pragma ghs end_externally_visible` (see “Green Hills Extension Pragma Directives” on page 752). To learn more about the use of the `externally_visible` attribute, see “Using Wholeprogram Interprocedural Optimizations” on page 338.
- 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.

- The preprocessor directive `#include_next` is deprecated and may be removed in future versions.
- Keyword `__inline__` is supported (see “Manual Inlining In C” on page 329).
- 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`.

- Keywords `__const__`, `__volatile__`, and `__signed__` are supported. The trailing underscores are optional. These keywords behave identically to the standard `const`, `volatile`, and `signed` keywords, respectively. They are accepted in all C and C++ dialects.
- 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`, `__bytereverse`, `__bigendian`, and `__littleendian` type qualifiers. For more information about these type qualifiers, see “Type Qualifiers” on page 597
- 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");
}
```

ISO C99

The ISO C99 is the default C dialect and conforms to *ISO/IEC 9899:1999*, including Technical Corrigenda 1, 2, and 3. We recommend this dialect for all new development, as well as for the compilation of any existing C code that approximates ISO C99.

To specify use of this dialect:



Set the **C/C++ Compiler**→**C Language Dialect** option to one of the following:

- **ISO C99 (-c99)** — [default] ISO C99 with extensions accepted.
- **Strict ISO C99 (-C99)** — ISO C99 with most extensions rejected.

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** 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 or Later Dialects

- The macro `__STDC_VERSION__` has the value of the standard version in use.
- `#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 or Later Dialects

- 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 ANSI C and C99 or Later Dialects

- 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`.

ISO C11, ISO C17, and ISO C18

The ISO C11, ISO C17, and ISO C18 dialects conform to *ISO/IEC 9899:2018*. The three dialects share a section because they use an identical implementation in the Green Hills compiler (excepting `__STDC_VERSION__`). These dialects are recommended in order to make use of threading, atomics, bounds-checking library functions, aligned memory, or generics.

To specify use of ISO C11:



Set the **C/C++ Compiler→C Language Dialect** option to one of the following:

- **ISO C11 (-c11)** — ISO C11 with extensions accepted.
- **Strict ISO C11 (-C11)** — ISO C11 with most extensions rejected.

To specify use of ISO C17:



Set the **C/C++ Compiler→C Language Dialect** option to one of the following:

- **ISO C17 (-c17)** — ISO C17 with extensions accepted.
- **Strict ISO C17 (-C17)** — ISO C17 with most extensions rejected.

To specify use of ISO C18:



Set the **C/C++ Compiler→C Language Dialect** option to one of the following:

- **ISO C18 (-c18)** — ISO C18 with extensions accepted.
- **Strict ISO C18 (-C18)** — ISO C18 with most extensions rejected.

Notable Differences in ISO C11, ISO C17, and ISO C18

Anonymous structs and unions

ISO C11 introduces anonymous `structs` and `unions` into the language. In earlier permissive C modes, a version of this feature is supported as an extension. However, this extension differs from the ISO C11 implementation in that it allows some non-standard forms, as described below. Starting with ISO C11 (both strict and permissive), only the standard forms are allowed by default. Non-standard forms will result in the following error:

```
error #619-D: nonstandard unnamed field
```

Non-standard forms of anonymous `structs` and `unions` can be enabled in ISO C11 and later by passing `--diag_suppress=619`. Standard forms are defined without a tag and with an inline type definition. For example:

```
struct A {  
    struct { /* ... */ }; // Standard in C11  
};
```

The following forms are accepted as an extension in **-c99** and earlier permissive modes, but rejected by default in ISO C11:

- An anonymous `struct` or `union` member defined inline with a tag:

```
struct A {  
    struct B { /* ... */ }; // Non-standard; includes a tag  
};
```

- An anonymous `struct` or `union` member that is the name of a type defined outside of the enclosing `struct` or `union` (either a `struct/union` tag or a `typedef`):

```
struct B { /* ... */ };  
typedef struct { /* ... */ } C;  
struct A {  
    struct B; // Non-standard; type defined out-of-line  
    C; // Non-standard; type defined out-of-line  
};
```

Note that type qualifiers are not supported on anonymous `structs` or `unions` in any language mode. Attempting to use type qualifiers on the standard form of anonymous `struct` or `union` in ISO C11 or later will result in the following error:

```
error #1481-D: anonymous union qualifier is ignored
```

When using type qualifiers with non-standard forms or in earlier language modes, diagnostic output may vary.

ISO C11, ISO C17, and ISO C18 Feature Support

The compiler supports nearly all features of ISO C11, ISO C17, and ISO C18. Some features are partially supported, or are only supported in certain circumstances. These features are listed in the section below. For unsupported features, see “Non-Conformance with C Standards” on page 634.

ISO C11, ISO C17, and ISO C18 Features with Qualified Support

Threads, Thread-Local Storage, and Synchronization Primitives

C11 threads include a subset of features provided by C++11 threads. For details of C++11 threading support, see “Threads, Thread-Local Storage, and Synchronization Primitives” on page 672. Any caveats described therein also hold for C11 threads, provided the caveats apply to features that exist in C11.

Floating-Point Environment Not Thread-Local

Regardless of whether thread-local storage is supported, all threads share the same floating-point environment. (See the C11 standard for the definition of “floating-point environment.”)

Namespace Member `atomic_flag` Type

The type `atomic_flag` may not be lock free. If your target lacks hardware support for atomic operations, `atomic_flag` will use a lock.

Memory Ordering with `memory_order_consume`

Memory orderings specified as consume operations (`memory_order_consume`) are treated as acquire operations (`memory_order_acquire`).

Atomic Operations

Atomic operations are supported but can only synchronize accesses within a single process or address space. Lock-free atomic operations are supported on ARM64. Lock-free types may be up to 64-bits in size.

Some String and Character Literals Require UTF-8

String literals prefixed by `u8`, `U`, or `u` (such as `u8"asdf"`, `U"asdf"`, and `u"asdf"`) are supported only when the multibyte character encoding is set to UTF-8 for input and output (**-kanji=utf8**). This is the default setting in C11, C++11, and later modes.

Character literals prefixed by `U` or `u` (such as `U'x'` and `u'x'`) require the same setting. For more information, see “Host and Target Character Encoding” on page 210.

Functions Defined in `<uchar.h>` Require UTF-8

The `<uchar.h>` functions `c16rtomb`, `c32rtomb`, `mbrtoc16`, and `mbrtoc32` are supported only when your target's multibyte character encoding is set to UTF-8 (**`-kanji=utf8`** or **`-kanji=shiftjis=utf8`**). This is the default setting in C11, C++11, and later modes. To indicate this, the macros `__STDC_UTF_16__` and `__STDC_UTF_32__` are not defined for other target multibyte character encodings. For more information, see “Host and Target Character Encoding” on page 210.

Annex K Library

The majority of Annex K ("Bounds-checking interfaces") is supported. All but the following functions are provided: `wctomb_s`, `mbstowcs_s`, `wcstombs_s`, `wcrtomb_s`, `mbsrtowcs_s`, and `wcsrtombs_s`.

INTEGRITY and `time.h`

Some versions of INTEGRITY provide their own versions of **`time.h`** that lack the new declarations added in C11. Contact Green Hills Support if you encounter errors related to C11 declarations missing from **`time.h`** on INTEGRITY. Typically, these errors are of the form "function declared implicitly" or "identifier is undefined".

GNU C

GNU C 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 “C Extensions for ANSI C” on page 577, together with the extensions listed in this section, and those listed in “GNU C/C++ Extensions” on page 591.

- 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 “Manually Indicating Functions for Single-Pass Inlining” on page 329.
- 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 589 and “GNU C++ Extensions” on page 682.

- 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 “C Extensions for ANSI C” on page 577.
- 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'
```

- 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

Differences Between GNU Mode and the GNU Compiler

Signedness of Bitfields

The Green Hills compiler differs from the GNU compiler in the treatment of bitfields without an explicit signed or unsigned qualifier. The Green Hills compiler defaults to determining the signedness of such bitfields on a per processor family basis. This is in contrast to the GNU compiler, which always defaults to using signed bitfields. Further, the Green Hills compiler also has a special case for the signedness of single-bit bitfields; the GNU compiler has no such special case. For more information, see “Signedness of Bitfields” on page 601.

Default Behavior of the GNU/C99 Dialects

The GNU compiler's default behavior of the `inline` keyword in its `-std={c99,gnu99}` modes is version-dependent. Newer GNU compilers default to using ISO C99 semantics for the `inline` keyword in these modes while older GNU compilers default to using GNU89 semantics for the `inline` keyword. The Green Hills compiler, when in the **-gnu99** dialect, always defaults to using ISO C99

semantics for the inline keyword, regardless of which GNU version is being emulated. To change the inline semantics used in the **-gnu99** dialect, see “-gnu99 Inline Semantics” on page 309.

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 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.

Sequence Related Undefined Behavior Checking

The Green Hills compiler attempts to detect undefined behavior that results from improperly sequenced expressions. For example, the following code results in undefined behavior because the stored value of `a` is modified more than once with no intervening sequence point.

```
int a = 0;  
a = a++;           // Undefined behavior
```

Likewise, an object may not have its stored value both modified and read without an intervening sequence point. The one exception to this is when the read of the stored value is used to determine the new value to be stored.


```
int a = 0, b = 0;
a = a + 1;           // Okay
a = a + a++;         // Undefined behavior
b = a + a++;         // Undefined behavior
```



Note

While the compiler is fairly good at detecting this case of undefined behavior, it does not do so exhaustively. That is, a diagnostic is not necessarily generated for all such cases of undefined behavior. This is especially true for large, complex expressions. Furthermore, the compiler may sometimes diagnose well-defined expressions, so such diagnostics should not be taken as guarantees of undefined behavior.

To disable this checking and its associated diagnostics (2017, 2018) altogether, the `--diag_suppress=2017,2018` option may be used. Note that these diagnostics are automatically disabled in C++17 due to different sequencing rules.

Type Qualifiers

The following type qualifiers are available:

`__bypaired`

The `__bypaired` 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 `__bypaired` qualifier should not be used for automatic variables or parameters, although automatic pointers to `bypaired` objects are allowed.

Example 11.1. Using the `__bypaired` Qualifier

```
__bypaired int x;
void foo(void)
{
    /* OK - automatic pointer to bypaired data */
    __bypaired int *p = &x;

    /* Not OK - __bypaired should not apply to an automatic variable */
    int * __bypaired p2;

    ...
}
```

`__byteversed` should be applied only to integral and pointer types wider than `char`. `__byteversed` variables should not be defined with an initializer.

`__bigendian` and `__littleendian`

Similarly, `__bigendian` and `__littleendian` can be used in place of `__byteversed` 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 30).

`volatile`

Use the `volatile` type qualifier to designate that an object may change due to an external process.

Enabling any optimization strategy save **-Onone** or **-Omaxdebug** will implicitly enable memory optimization (**-OM**). This option 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.

Using Volatiles with INTEGRITY TimeMachine Enabled

If you have the separately licensed INTEGRITY TimeMachine feature, you can additionally use `volatile __sideeffect` and `volatile __nosideeffect`. The primary purpose of these qualifiers is to improve program performance and the ease of debugging `volatile` objects. If `volatile __sideeffect` is used, the compiler assumes that additional, compiler-generated memory accesses may cause unwanted behaviors in the system. This assumption is explicitly negated with `volatile __nosideeffect`.

To understand why these qualifiers are helpful, consider the two most common circumstances in which the `volatile` keyword is used. In both instances, the `volatile` keyword behaves identically, but different aspects of its behavior are relevant:

1. An object residing in "normal" memory (RAM or ROM) is modified by other threads. Additional compiler-generated reads of the object will have no unwanted effects. The `volatile` keyword helps by ensuring that explicit object accesses will not be optimized away.
2. An object residing in "device" memory (an address controlling or controlled by an external device) is subject to value modifications by another thread or external source. Further, additional compiler-generated reads of the object may cause unwanted effects. The `volatile` keyword helps by ensuring that the compiler accesses the object no more times than the code explicitly dictates.

The C and C++ language standards do not disambiguate the above cases. As such, the compiler instruments `volatile` objects with a run-time pointer-check heuristic to differentiate the cases by checking whether an address is in normal or device memory. Executing this pointer check is inefficient, and thus `__sideeffect` and `__nosideeffect` can be used to pre-differentiate the cases. This can lead to better performance and debugging accuracy. In summary, if an object resides in normal memory (case 1), use `volatile __nosideeffect`. If an object resides in device memory (case 2), use `volatile __sideeffect`.

Conversion rules:

- Implicit conversions between pointer types are allowed if assignments go to stricter types. Unqualified pointer types (such as `int *`) are least strict, pointers to `volatile __nosideeffect` are more strict, and pointers to `volatile __sideeffect` are most strict.

- Because the plain `volatile` pointer object can mean either case 1 or case 2 above, pointers to `volatile __nosideeffect` can be assigned to pointers to plain `volatile` objects, and pointers to plain `volatile` objects can be assigned to `volatile __sideeffect`.
- All implicit conversions not specified above are disallowed.
- All explicit conversions are allowed.

Example:

```
void func(                                int *any_ptr,
        volatile __nosideeffect int *ram_ptr,
        volatile                int *vol_ptr,
        volatile __sideeffect   int *dev_ptr)
{
    dev_ptr = vol_ptr = ram_ptr = any_ptr; // allowed
    any_ptr = ram_ptr = vol_ptr = dev_ptr; // error
}
```

See also the `ghs volatile_sideeffect #pragma` directive in “Green Hills Extension Pragma Directives” on page 752.

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 `unsigned` 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 because **--signed_fields** was passed.
- `char c:2` has the same signedness as the `char` type (see “Signedness of Char Type” on page 204).
- `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 because **--signed_fields** was passed.

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 135).

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 “ARM64 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.

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 `<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 11.2. 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 the non-strict versions of ISO C99 (or later) and ANSI C. `__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 17, “Enhanced asm Macro Facility for C and C++” on page 857.

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 (<code>--asm_errors</code>) • Warnings (<code>--asm_warnings</code>) • Silent (<code>--asm_silent</code>)



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 **.s** or **.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.

See also “Preprocess Assembly Files” on page 224.

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++:

- *Multibyte 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. Multibyte 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 multibyte.
- *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 multibyte 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 multibyte 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 multibyte 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 multibyte 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**.

Compiler Support for Multibyte Characters

The Green Hills Software C/C++ compilers deal with multibyte character sets in two different places and reject them elsewhere. Multibyte 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 multibyte 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 multibyte strings and converted using functions such as `mbstowcs()`. Furthermore, if the multibyte character set has bytes which match certain ASCII punctuation characters, multibyte string literals may not be used either. In either case, it is possible to place multibyte 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 multibyte 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 210.

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 210.

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 ISO C standard requires that a conforming compiler implementation accept programs of a certain complexity and size. These requirements are listed in the “Translation limits” section of the ISO C standard and are expressed in terms of acceptable limits. The Green Hills compiler meets these requirements. For completeness, we list below all of the acceptable limits followed parenthetically with the limits supported by the Green Hills compiler. The code (U) indicates the compiler is theoretically unlimited, while the code (U > *n*) indicates that the compiler is both theoretically unlimited and tested to handle at least the value *n*.

- 127 nesting levels of compound statements, iteration control structures and selection control structures (U > 512)
- 63 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)
- 63 nesting levels of parenthesized declarators within a full declarator (U)

- 63 nesting levels of parenthesized expressions within a full expression ($U > 200$)
- 63 significant initial characters in an internal identifier or a macro name (U)
- 31 significant characters in an external identifier (U)
- 4095 external identifiers in one translation unit ($U > 32767$)
- 511 identifiers with block scope declared in one block ($U > 2048$)
- 4095 macro identifiers simultaneously defined in 1 translation unit ($U > 16000$)
- 127 parameters in one function declaration ($U > 30000$)
- 127 arguments in one function call (U)
- 127 parameters in one macro definition ($U > 2700$)
- 127 arguments in one macro call ($U > 2700$)
- 4095 characters in a logical source line ($U > 64535$)
- 4095 characters in a character string literal or wide string literal (after concatenation) ($U > 16384$)
- 65535 bytes in an object (variable) (in a hosted environment only)
- 15 nesting levels for `#include` files ($U > 256$)
- 1023 `case` labels for a `switch` statement, excluding those for any nested `switch` statements ($U > 8000$)
- 1023 members in a single structure or union ($U > 4096$)
- 1023 enumeration constants in a single enumeration ($U > 32767$)
- 63 levels of nested structure or union definitions in a single `struct-declaration-list` ($U > 1024$)



Note

Under heavy optimization, limitations on host resources (such as CPU time or memory) may lower the practical limit on some of these bounds. Because of this, **-Omax** (and to a lesser extent other optimization strategies) may not be appropriate for source files containing very large functions or data structures.

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 ARM64 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.2 Undefined Behavior

When an expression having signed promoted type is left-shifted and either the value of the expression is negative or the result of shifting would not be representable in the promoted type, the result is equivalent to the following sequence:

1. Converting the expression to an unsigned type with a size equal to the promoted type.

2. Performing an unsigned left shift by the specified amount.
3. Converting the result of the unsigned left shift back to the signed promoted type.

For more information, see section 6.5.7 of the ISO C99 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 259.

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 210.

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: "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 779.

As with C++11, support for multiple threads in C11 and later depends on the operating system, but can be added by customizing the file **`install_dir/src/libsys/ind_thrd11.c`** included in your compiler installation. For more information, see "Threads, Thread-Local Storage, and Synchronization Primitives" on page 672.

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 values that fit in the type `wchar_t`. To change the size of `wchar_t`, see “WChar Size” on page 310.

The escape sequence values are as follows.

Sequence	Value
<code>\a</code>	7
<code>\b</code>	8
<code>\f</code>	12
<code>\n</code>	10
<code>\r</code>	13
<code>\t</code>	9
<code>\v</code>	11

`char` is an 8-bit arithmetic type with the usual properties.

The character type `char` is unsigned by default. To control the signedness of `chars` manually, see “Signedness of Char Type” on page 204.

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 multibyte character has the same integer value as a wide character constant consisting only of the rightmost multibyte character. There is only one situation where a single multibyte character can map to multiple wide characters: when the wide character type is `char16_t` and the multibyte character would require a surrogate pair. A `char16_t` character literal containing such a character is rejected with the following error message:

```
error #1620: invalid character for char16_t literal
```

In contrast, in a `char16_t` string literal the character would be accepted and encoded using a surrogate pair.

If a multibyte character or escape sequence is outside the representable range of `wchar_t`, the value is truncated into range by discarding the high bits. Other than that, it is not possible for a multibyte character or escape sequence in a wide character constant to be absent from the multibyte character set, because the multibyte character set includes all values that fit in the type `wchar_t`.

The C locale is used to convert multibyte 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 the ISO C99 standard under section 7.11.1.1. Only the C locale is implemented.

Any byte `B` which does not begin a valid multibyte character is treated as a single-byte multibyte sequence mapping to the value `(unsigned char)B`.

Differently-prefixed wide string literal tokens cannot be concatenated.

The encoding of `wchar_t` is determined by the **-kanji** option (see “Host and Target Character Encoding” on page 210).

When the target character encoding is set to UTF-8 (**-kanji=utf8** or **-kanji=shiftjis=utf8**), the types `char16_t` and `char32_t` are encoded using UTF-16 and UTF-32 respectively. For other target character encodings, both types use the same encoding as `wchar_t`.

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, $-128 \dots 127$ If char is unsigned, $0 \dots 255$ For information about the signedness of char, see “Signedness of Char Type” on page 204.
signed char	8	$-128 \dots 127$
unsigned char	8	$0 \dots 255$
short	16	$-32768 \dots 32767$
unsigned short	16	$0 \dots 65535$
int	32	$-2147483648 \dots 2147483647$
unsigned int	32	$0 \dots 4294967295$
long	32 or 64	$-2147483648 \dots 2147483647$ or $-9223372036854775808 \dots 9223372036854775807$
unsigned long	32 or 64	$0 \dots 4294967295$ or $0 \dots 18446744073709551615$
long long	64	$-9223372036854775808 \dots 9223372036854775807$
unsigned long long	64	$0 \dots 18446744073709551615$

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.

Conversion of an unsigned integer to a signed integer of an unequal length is equivalent to converting the unsigned integer to an unsigned integer of an equivalent size to the signed type, then performing an unsigned to signed conversion in accordance with the previous paragraph.

Conversion of a signed integer A to a signed integer B of shorter length is equivalent to the following steps:

1. Converting A to an unsigned value of equal length.
2. Converting the result to an unsigned value of an equivalent size to B.
3. Performing an unsigned to signed conversion (as explained above).

Extended integer types are not supported on stand-alone targets.

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

For more detailed information about the toolchain's floating-point behavior, see “Floating-Point Behavior” on page 652.

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.

When a decimal floating constant is converted to a floating-point number at translation time, the value chosen is normally the nearest representable value. For more information, see “Translation-Time Accuracy” on page 653.

The `FP_CONTRACT` `#pragma` directive is recognized in C99 (or later dialects) 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 (or later dialects) 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	1.175494350822287508e-38 to 3.40282347e+38
double	64	2.2250738585072014e-308 to 1.7976931348623157e+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 601.

Members of structures are padded and aligned as described in “ARM64 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. They may not have atomic 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 `ints` 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 longs` 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 longs` 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 `ints` are 32 bits, `shorts` are 16 bits, `chars` 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 604.

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 implementation does not define any particular behavior for accesses to automatic (local) volatile variables that do not have their addresses taken. Therefore, such variables may have their accesses optimized in a way that will not affect the behavior of the program. 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 56.

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 747.

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 790.

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"
```

No floating-point status flags are stored by `fegetexceptflag()`.

`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.

For information about domain errors, pole errors, and range errors in the mathematics functions, see “Special-Case Values” on page 655.

When an `fmod` function has a second argument of zero, a domain error occurs and a NaN is returned.

When a `remainder` function has a second argument of zero, a domain error occurs and a NaN is returned.

The base-2 logarithm of the modulus used by the `remquo` functions is 3.

When a `remquo` function has a second argument of zero, a domain error occurs and a NaN is returned.

Signal handling does not happen on stand-alone targets.

`NULL` normally expands to `((void*)0)` for the C language. For C++, or when certain MISRA checks are enabled, the `NULL` macro expands to one of `0`, `0L`, or `0LL`.

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 on INTEGRITY. For information about buffering support on stand-alone targets, see “Less Buffered I/O” on page 779.

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 for multibyte characters in files, use the driver option **-kanji** (see “Host and Target Character Encoding” on page 210).

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 `freopen()` function does not permit any mode changes when `NULL` is passed as the filename. When passing the current file's name explicitly, the semantics of `freopen()` depend on the file system. In general, any file can be reopened with any mode.

When printing with the `%f` conversion specifier, any NaN is printed as `nan` (regardless of sign bit). 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 set `errno` to `ERANGE` when underflow occurs.

The functions `calloc()`, `malloc()`, and `aligned_alloc()` 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()`, `_Exit()`, or `quick_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 local time zone is determined by `__gh_timezone()`. The local daylight saving time is determined according to the rules for the USA.

The precision of the type `time_t` is one second. A `time_t` with value 0 represents midnight, January 1, 1970 (GMT). If `time_t` is a 32-bit type, the range of times representable is from December 13 20:45:52 1901 (-2147483648 or 0x80000000) to January 19 03:14:07 2038 (2147483647 or 0x7fffffff). If `time_t` is a 64-bit type, the range of times is from -9223372036854775808 (0x8000000000000000) to 9223372036854775807 (0x7fffffffffffffff). The size of the type `time_t` can be affected by the **-time64** option (see “Size of `time_t`” on page 207).

The precision of the type `clock_t` is 1/CLOCKS_PER_SEC seconds. The range (as a multiple of that precision) is the same as the type `long` (see “J.3.5 Integers” on page 618).

The era for the `clock` function is dependent on the `times` function in **libsys**, which is expected to be implemented by the user.

The `TIME_UTC` epoch is dependent on the operating system, or (if customized) the function `_Xtime_get_ticks` in *install_dir/src/libsys/ind_thrd11.c*. For more information, see “Threads, Thread-Local Storage, and Synchronization Primitives” on page 672.

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()`.

For information about which functions in **<math.h>** honor the rounding direction mode, see “Rounding Modes” on page 659.

J.3.13 Architecture

The macros specified in **<float.h>**, **<limits.h>**, and **<stdint.h>** are documented in their respective header files.

An object with automatic or thread storage duration may be accessed indirectly from a thread other than its associated thread. Such an access works the same as if executed from the thread associated with the object. As always, if multiple threads access the same object they must synchronize with each other to avoid data races.

The sizes of the basic types are specified in “J.3.5 Integers” on page 618 and “J.3.6 Floating Point” on page 619. See also “ARM64 Characteristics” on page 26 and “Data Type Macros” on page 738.

For the layout of struct and union types, see “Structure Packing” on page 30.

The extended alignments supported for objects with static, thread, and automatic storage duration (via `_Alignas`) are the same ones supported with `__attribute__((aligned(x)))`. For more information, see “Attributes” on page 637.

The extended alignments supported for objects with allocated storage duration (via `aligned_alloc`) depend on the version of `malloc` in use. For **Real-Time** and **Arena** `malloc`, `aligned_alloc` supports arbitrarily large alignments, assuming sufficient heap space for the alignment padding. For other `malloc` settings, `aligned_alloc` supports only those alignments already guaranteed by `malloc`.

The `_Alignof` operator returns the alignment requirement of its operand type. This value depends on the ABI and certain compiler options.

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 616.

There are no shift states in the encoding of multibyte 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`.

The following table explains the return values of the listed functions for certain characters:

Function	Characters for which non-zero value is returned
<code>isalnum()</code>	<code>'A'...</code> <code>'Z'</code> , <code>'a'...</code> <code>'z'</code> , and <code>'0'...</code> <code>'9'</code>

Function	Characters for which non-zero value is returned
<code>isalpha()</code>	'A'..'Z' and 'a'..'z'
<code>isctrl()</code>	0..31 and 127
<code>isdigit()</code>	'0'..'9'
<code>islower()</code>	'a'..'z'
<code>isprint()</code>	32..126
<code>isupper()</code>	'A'..'Z'

`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)` if `wc` is representable as a `char`; otherwise `iswfoo(wc)` returns false.

Only the C locale is implemented. The C locale is documented in *X3J11/90-013* under section 7.4.1.1 and in the ISO C99 standard under section 7.11.1.1.

For a detailed description of the C locale's values related to `strftime()`, see the ISO C99 standard section 7.23.3.4, paragraph 7.

The native environment (the one that results from calling `setlocale()` with a locale argument of "") is equivalent to the C locale.

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. Note that if one of the macros listed below is not defined, the corresponding string will not be returned by `strerror`.

0	"Success"
EPERM	"Operation not permitted"
ENOENT	"No such file or directory"
ESRCH	"No such process"
EINTR	"Interrupted system call"
EIO	"I/O error"
ENXIO	"No such device or address"
E2BIG	"Argument list too long"
ENOEXEC	"Exec format error"
EBADF	"Bad file descriptor"
ECHILD	"No child processes"
EAGAIN	"Resource temporarily unavailable"
ENOMEM	"Cannot allocate memory"
EACCES	"Permission denied"
EFAULT	"Bad address"
ENOTBLK	"Block device required"
EBUSY	"Device or resource busy"
EEXIST	"File exists"
EXDEV	"Invalid cross-device link"
ENODEV	"No such device"
ENOTDIR	"Not a directory"
EISDIR	"Is a directory"
EINVAL	"Invalid argument"
ENFILE	"File table overflow"
EMFILE	"Too many open files"
ENOTTY	"Inappropriate ioctl for device"
ETXTBSY	"Text file busy"
EFBIG	"File too large"
ENOSPC	"No space left on device"
ESPIPE	"Illegal seek"
EROFS	"Read-only file system"

EMLINK	"Too many links"
EPIPE	"Broken pipe"
EDOM	"Argument too large"
ERANGE	"Result too large"
EDEADLK	"Resource deadlock avoided"
EINPROGRESS	"Operation now in progress"
EALREADY	"Operation already in progress"
ENOTSOCK	"Socket operation on non-socket"
EDESTADDRREQ	"Destination address required"
EMSGSIZE	"Message too long"
EPROTOTYPE	"Protocol wrong type for socket"
ENOPROTOOPT	"Protocol not available"
EPROTONOSUPPORT	"Protocol not supported"
ESOCKTNOSUPPORT	"Socket type not supported"
EOPNOTSUPP	"Operation not supported"
EPFNOSUPPORT	"Protocol family not supported"
ENOLCK	"No locks available"
EAFNOSUPPORT	"Address family not supported by protocol family"
EADDRINUSE	"Address already in use"
EADDRNOTAVAIL	"Can't assign requested address"
ENETDOWN	"Network is down"
ENETUNREACH	"Network is unreachable"
ENETRESET	"Network dropped connection on reset"
ECONNABORTED	"Software caused connection abort"
ECONNRESET	"Connection reset by peer"
ENOBUFS	"No buffer space available"
EISCONN	"Socket is already connected"
ENOTCONN	"Socket is not connected"
ESHUTDOWN	"Can't send after socket shutdown"
ETOOMANYREFS	"Too many references: can't splice"
ETIMEDOUT	"Operation timed out"

ECONNREFUSED	"Connection refused"
ENAMETOOLONG	"Name too long"
EHOSTDOWN	"Host is down"
EHOSTUNREACH	"No route to host"
ENOTEMPTY	"Directory not empty"
EPROCLIM	"Too many processes"
EUSERS	"Too many users"
EDQUOT	"Disk quota exceeded"
EILSEQ	"Illegal byte sequence"
ECANCELED	"Operation canceled"
ENOTSUP	"Unsupported part of standard"
ENOSYS	"No such system call"
ESTALE	"Stale file handle"
EAUTH	"Authentication error"
EBADRPC	"Bad RPC structure"
EPROGUNAVAIL	"RPC program is unavailable"
EPROCUNAVAIL	"Bad RPC procedure for program"
EPROGMISMATCH	"Incorrect RPC program version"
ERPCMISMATCH	"Incorrect RPC version"
ENEEDAUTH	"Authenticator required"
EFTYPE	"Inappropriate file type"
EIDRM	"Identifier removed"
ENOMSG	"No such message"
EBADMSG	"Bad message"
ELOOP	"Too many symbolic links"
EPROTO	"Protocol error"
EMULTIHOP	"Multihop attempted"
EOVERFLOW	"Value too large for defined data type"
ENOLINK	"Link severed"
EBADDISK	"Bad disk"

The standard library's `towctrans()` provides only what is required by the Standard. `toupper(wc)` returns `(unsigned char)toupper((char)wc)`, and `tolower(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"

Implementation-Defined Behavior for Annex K

If the macro `__STDC_WANT_LIB_EXT1__` is not defined when a header is first included, that header will not declare any of the functions, macros, or types from Annex K. These declarations must be explicitly requested by defining `__STDC_WANT_LIB_EXT1__` to 1.

When a library function calls the current runtime-constraint handler, the second argument passed is `NULL`.

The default runtime-constraint handler is `__ghs_default_constraint_handler_s()`. Normally this function just calls `abort_handler_s()`. However, its behavior may be customized. For more information, see .

Non-Conformance with C Standards

This section covers instances of non-conformance with the various C standards. Wherever possible, workarounds and strategies for increasing conformance are provided.

All C Versions

Floating-Point Behavior

For details on standards-relevant aspects of floating-point behavior in C, see “Floating-Point Behavior” on page 652.

Behavior of `printf` with `%lc` and Null Wide Character

When a narrow-string `printf` function is called with the `%lc` format specifier and a null wide character (`L'\0'`), it will write a null byte (`'\0'`) to its output. For example, the following call will write `"a \0 b"` to `buf`:

```
snprintf(buf, sizeof(buf), "a %lc b", (wint_t)L'\0');
```

The standard would treat the above example differently, such that `snprintf` would not write anything for the `%lc` specifier. Instead, `"a b"` would be written to `buf`.

Note that because the non-standard behavior is more consistent with the behavior of `%c`, the approach taken by the Green Hills libraries is common among other C implementations.

Behavior of `scanf` with `c` Specifier and Field Width

If a `scanf` function is called with the `c` format specifier and a field width, and end-of-file is reached after fewer characters than the field width, the function will report a successful match. For example, consider the following code snippet:

```
FILE *stream = /* ... */;
char buf[8];
int ret = fscanf(stream, "%8c", buf);
```

If this is executed when the only unread characters in `stream` are `abcd` followed by end-of-file, then `ret` will be assigned the value 1. The first four elements of `buf` will be populated with `abcd`, while the last four elements of `buf` will be unmodified. In contrast, the C standard would consider this a matching failure, meaning that `ret` should be assigned the value 0. For the above call, the behavior matches the standard if there are eight or more unread characters remaining in `stream`. In that case, the next eight characters from `stream` will be read into `buf` and `ret` will be assigned the value 1. This applies to all variants of `scanf`.

C11 or Later

Synchronization Across Memory Allocation

There are numerous memory allocation routines that allocate or deallocate a particular unit of memory; these include `malloc()`, `free()`, `realloc()`, and the C++ `new` and `delete` operators. If such memory is deallocated in one thread, then overlapping memory may be returned by a subsequent allocation in another thread. In this case, these routines all ensure that accesses to that block of memory in the two threads will not lead to a data race (specifically, they ensure that any accesses to the old block before it is deallocated are ordered before accesses to the new block after it is allocated). Note, however, that ordering is not guaranteed with respect to other memory.

Boundaries for `scanf_s` Runtime-Constraint Violations

When the Annex K function `scanf_s` is used with the `%lc`, `%ls`, or `%l[` specifier, it will report a runtime-constraint violation if the provided array length is greater than `RSIZE_MAX`, rather than `RSIZE_MAX/sizeof(wchar_t)`. This applies to all variants of the function from Annex K, but not pre-Annex K variants like `scanf` (which have no runtime-constraints).

Annex L Library

Analyzability features from the optional Annex L are not supported.

TSS_DTOR_ITERATIONS Definition

`TSS_DTOR_ITERATIONS` is defined to be 0. To determine the maximum number of times that TSS destructors can be called, refer to your operating system's documentation.

<stdatomic.h> Brings in Additional Definitions

The header `stdatomic.h` defines `NULL`.

Type Qualifiers Not Allowed in Anonymous Structs or Unions

Anonymous structs and unions may not be declared with type qualifiers. For example, the following declaration:

```
union A {  
    const union { int x; };  
};
```

will result in this error:

```
error #1481-D: anonymous union qualifier is ignored
```

For more information about support for anonymous structs and unions, see “Notable Differences in ISO C11, ISO C17, and ISO C18” on page 585.

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. The value *x* must be a power of 2 and must be less than or equal to 4096. An `__attribute__((aligned(x)))` annotation will only increase the alignment requirement of the entity. 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. This attribute may not be used to reduce the alignment of fields, classes, structs, or unions. In these instances, the compiler will issue a warning for the attribute and ignore it. (However, structure packing will still override the alignment of a field, even if the type of the field was explicitly aligned by this attribute). 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
}
```

Arrays must be laid out contiguously in memory (without gaps between elements). As such, the alignment of the elements cannot be honored if the element type's alignment is greater than its size. This typically occurs when a typedef is annotated with `__attribute__((aligned(x)))` and an array is declared using that typedef name as the element type. For example:

```
// Attribute applies to typedef
typedef struct { int a; } T1 __attribute__((aligned(8)));
// Attribute applies to struct
typedef struct { int a; } __attribute__((aligned(8))) T2;

T1 t1[10]; // Array is 8-byte aligned, but elements are not.
T2 t2[10]; // Elements are 8-byte aligned (and therefore, so is the array)
```

Some caveats applicable to the `alignas` keyword are also applicable to this attribute. For more information, see “Over-Alignment with the `alignas` Keyword” on page 675.

`always_inline`

Annotating a function with `__attribute__((always_inline))` causes the compiler to inline calls to the function if it is able to do so, regardless of the function's size and complexity. This attribute will also cause the compiler to ignore command line options that would otherwise disable inlining for the annotated function. Note that `always_inline` does not guarantee that all calls to the function will be inlined. For example, the compiler will not inline calls to the function if it is called recursively or declared as weak. Related diagnostics are 2021, 2022, and 2023. For more information about controlling inlining, see “Inlining Optimizations” on page 326.

`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
};
```

const

Annotating a function prototype with `__attribute__((const))` tells the compiler that the function is `const`—a distinct concept from being a `const` member function. The `const` attribute is similar to the `pure` attribute, but it may allow for further optimization and it imposes additional restrictions on functions. Specifically, functions annotated with the `const` attribute may not read from global memory, either by reading a global variable or by dereferencing a pointer argument.

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__((noccommon))` forces it to be defined as a non-common symbol, even when **--commons** is in effect. For more information about common symbols, see the **Target→Text and Data Placement→Allocation of Uninitialized Global Variables** option in “Text and Data Placement” on page 90.

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_history

Annotating a function with `__attribute__((entry_exit_history(1)))` causes the compiler to instrument the function for entry/exit history, as if **-gen_entry_exit_history** were passed.

Annotating a function with `__attribute__((entry_exit_history(0)))` causes the compiler not to instrument the function for entry/exit history, as if **-no_gen_entry_exit_history** were passed.

For more information, see “Function Entry/Exit History” on page 45.

`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 “Using Wholeprogram Interprocedural Optimizations” on page 338).

`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 $n * \text{sizeof}(e)$ 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 $n * \text{sizeof}(e)$ 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.

no_variadic_floats

Annotating the declaration of a function that takes variable arguments with `__attribute__((no_variadic_floats))` causes the compiler not to save floating-point registers onto the stack for the purpose of variable arguments. You must not attempt to consume variable arguments of floating-point type from a `va_list` based on such a function.

naked

Annotating a function with `__attribute__((naked))` tells the compiler not to generate a prologue, epilogue, or other non-asm code for a given function. Instead, the compiler is directed to derive all function code from an `asm` block in the function. When this attribute is applied to a function, only basic `asm` statements may be used in the function. For example:

```
int add(int a, int b) __attribute__((naked));
int add(int a, int b)
{
    asm("add r0, r0, r1");
}
```

```
    asm("ret");  
}
```

The following code is illegal for the reasons indicated below:

```
int add(int a, int b) __attribute__((naked));  
int add(int a, int b)  
{  
    int ret;                                // Illegal local variable  
    asm("add %0, r0, r1" : "=r" (ret));    // Illegal GNU extended asm  
    return ret;                             // Illegal return statement  
}
```

`noinline`

Annotating a function with `__attribute__((noinline))` tells the compiler and linker not to consider the function for inlining. For more information about controlling inlining, see “Inlining Optimizations” on page 326.

`instrument_prologue_epilogue`

Annotating a function with `__attribute__((instrument_prologue_epilogue(1)))` causes the compiler to instrument the function with prologue/epilogue calls, regardless of compiler options.

Annotating a function with `__attribute__((instrument_prologue_epilogue(0)))` causes the compiler not to instrument the function with prologue/epilogue calls, regardless of compiler options.

For more information, see “Instrument Prologue/Epilogue” on page 96.

`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.

`packed`

Annotating a structure or structure field with `__attribute__((packed))` changes its packing alignment. When applied to a structure, this attribute has an effect similar to `#pragma pack (1)`: all fields of the structure are reduced to their minimum possible alignment (typically 1 byte). When applied to a structure field, its packing alignment is reduced to the minimum possible alignment (again, typically 1 byte). In this instance, the packing alignment of other fields is left unchanged. For more information, see “Structure Packing” on page 30.

`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 declaration 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. Externally defined variables should be annotated at the point of declaration. 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 62.

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 539.

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_fp)) P2 {
    struct P1 p1; /* Illegal - P1 contains non-strong function pointer */
};
```

And in C++:

```
class Base {
    int (*myfp) (void);
};

class __attribute__((strong_fp)) 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_fp)) 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. This attribute does not affect linker optimization (see “Deletion of Unused Functions” on page 235). To prevent linker deletion, consider using `#pragma ghs reference` or the `-u` option. 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 747.

Floating-Point Behavior

Basic

Floating-point values use the IEEE-754 format (binary32 and binary64) and are in the same byte order as scalar values. The `long double` type uses the same format as `double`.

The toolchain includes the operators and functions defined in the C and C++ standards. Some floating-point standards describe features for which there is no corresponding syntax in C or C++. Unless explicitly documented otherwise, these features are not included.

The term “floating-point arithmetic” is used in the following subsections to refer to floating-point addition, subtraction, multiplication, division, comparison, and conversion between types. This definition does not include the standard library functions.

Floating-Point Rigor

The Floating-Point Rigor option (**-frigor**) specifies the overall accuracy/performance tradeoff made when calculating and comparing floating-point values. When **-frigor** is set to `default` or `fast`, code may exhibit the following types of behavior:

- Floating-point results may be rounded differently than specified by floating-point standards.
- Comparisons may not return the “unordered” result when either argument is a NaN (see “Not a Number (NaN)” on page 658).
- Optimizations may change the sign of zero (see “Signed Zeros” on page 658).


```
(1.0e+30 + -1.0e+30) + 1.0
```

evaluates to 1.0 because the parenthesized addition exactly cancels to zero. The expression

```
1.0e+30 + (-1.0e+30 + 1.0)
```

gives 0.0 because the parenthesized addition rounds to -1.0e+30. In the latter case, there is not enough precision to represent a difference of 1 at that magnitude. This optimization may be disabled using **-no_float_associativity** or **-frigor=accurate**.



Note

If a single floating-point calculation appears multiple times in a program, the compiler may optimize some instances differently. These optimizations can cause the same inputs to produce slightly different outputs.

Accuracy of Floating-Point Libraries

The standard library's floating-point routines are optimized for speed over accuracy.

Converting Between Floating-Point Values and Character Strings

Conversion between floating-point values and decimal strings may be inexact. This includes:

- `printf` with the `%e`, `%f`, or `%g` specifiers.
- `scanf` with the `%f` or equivalent specifiers.
- The `strtod` family of functions.

For instance, printing a floating-point value in decimal format will print up to 14 digits accurately. Printed digits after this point may be inaccurate. As a consequence, using `scanf` or `strtod` to parse a decimal string produced by `printf` may generate a value slightly different than the original.

To print floating-point values in a form suitable for reading back in, use the `%a` hexadecimal format. This will accurately give back the original value except in two cases:

1. If the original value is not supported by the hardware, then the hardware may change the value during printing or parsing (see “Target and Configuration Caveats” on page 661).
2. If the input is a NaN, the result may be a non-identical NaN.

If a hexadecimal string was not generated by `printf`, it may have more significant digits than would fit in the mantissa. In this case:

- `strtod`, `strtod`, and `strtold` will round the value according to the current rounding mode.
- `scanf` with the `%f` or equivalent specifier will return one of the two nearest representable values, but the manner in which that value will be chosen is unspecified.

Special-Case Values

The IEEE floating-point format includes the following special types of values:

- **Infinities** — The values `+inf` (positive infinity) and `-inf` (negative infinity) are recognized. Infinities can result from evaluating operations at poles (as when dividing by zero) or from overflow (when an operation's result is finite, but too large to be represented in the result type).
- **Not a Number (NaN)** — A NaN is returned by certain operations for inputs that have no usefully definable result. Examples include the expression `0/0` or taking the (real) square root of a negative value.
- **Signed zeros** — The IEEE floating-point format provides two ways to represent zero: one with a positive sign and the other with a negative sign. These values are treated the same in some respects, but not in others. For example:

`+0 == -0`

evaluates to `true`. However,

`1/+0`

and

`1/-0`

give differently-signed infinities (`+inf` and `-inf`, respectively).

- **Subnormals** — This special representation is used for values that are too close to zero to be represented using the normal format. Such values have fewer significant bits than normal values. Note that subnormals have also been called "denormals". Also, the term "gradual underflow" is sometimes used to describe the idea of returning a subnormal instead of zero when the result of an operation is too small to represent as a normal value.

Optimizations may impact the behavior of some types of special floating-point values. If this proves problematic, most can be manually disabled using options (see below). Note that **-frigor** settings also enable or disable these same optimizations (see “Floating-Point Rigor” on page 85).

Many FPUs do not have full support for special floating-point values. For example, it is common for FPUs to treat subnormal values as zero (that is, to "flush to zero"). For more information, consult your hardware's reference manual as well as “Target and Configuration Caveats” on page 661.

On systems using software floating-point, accurate handling of special floating-point values in arithmetic may be sacrificed for emulation speed, particularly for NaNs and subnormals.

The toolchain's floating-point math libraries will usually handle special input values and error conditions using recognizable and common behaviors. Specifically, those conditions are:

- **Infinite input** — If the limit of the function for large values exists, then the function returns that value. If not, this is a domain error; see below.
- **NaN input** — If the return type is a floating-point type, the function returns NaN. For other return types, NaN input is a domain error.
- **Pole error** (the input is finite and the mathematical result is an exact infinity) — The function sets `errno` to `ERANGE` and returns the (infinite) mathematical result. An example of this is `log(0.0)`. Note that this does not include cases where the limit of a function is positive infinity from one side of the input and negative infinity from the other, as with `tgamma(-1.0)`. Such cases are instead domain errors.

- Overflow (the mathematical result of a function call is finite but its magnitude is too large to represent) — The function sets `errno` to `ERANGE` and returns an infinity with the same sign as the mathematical result.
- Underflow (the mathematical result is non-zero but its magnitude is too small to represent as a normal value) — The function returns either a subnormal or zero, depending on which is closer to the mathematical result. If the result would be subnormal but the target does not support subnormals, then zero may be returned instead. It is unspecified whether the function sets `errno` to `ERANGE` or leaves it unchanged.
- Domain error (the input is outside of the function's domain) — The function sets `errno` to `EDOM`. If the return type is a floating-point type, NaN is returned. Other return types have unspecified return values (unless they are defined by the C standard).

Overflow and underflow are collectively referred to as range errors. The exceptions to the above behaviors are as follows:

- The `fma` family of functions do not set `errno`. Doing otherwise would interfere with some optimizations.
- The `sin` and `cos` families of functions do not set `errno` on infinite input. Doing otherwise would interfere with some optimizations.
- The `sin` and `cos` families of functions and the double-precision function `tan` may not return NaN on NaN input. This allows for smaller and faster code in the normal case.
- If **-ffunctions** is enabled, calls to math library functions may be replaced with faster inline instructions. These instructions will not set `errno` and may not return the standard results for special input values.

If the correct value is smaller than the smallest normal number, the `strtof()`, `strtod()`, and `strtold()` functions return the closest non-zero value with the same sign as the correct value and set `errno` to `ERANGE`.

Additional implementation details for special values are described in the following subsections.

Not a Number (NaN)

NaNs are unusual among floating-point values in that they are defined to compare as unordered with all values. This means that if `x` or `y` is NaN, then all three of the following expressions are defined to evaluate to `false`:

```
x < y
x > y
x == y
```

However, by default, the compiler may optimize comparisons in ways that assume neither operand is NaN. For example, the compiler may change `x < y` to `!(x >= y)`. You can prevent such transformations by using **-fNaN_cmp_unordered** or **-frigor=accurate** where supported.

In a given floating-point format, there are many different representations corresponding to NaN. The toolchain's implementation treats all NaNs as equivalent. In particular:

- There are no signaling NaNs.
- No interpretation is given to the mantissa bits of NaNs.
- For operations that take in a NaN as input and produce a NaN as output, the output NaN is not guaranteed to have the same representation as the input NaN. For example, it may be changed to a canonical NaN representation.
- Library functions that parse floating-point values from string input (such as `strtod`) recognize the form `nan(...)`, as specified by C99. However, the characters inside the parentheses are ignored.

Signed Zeros

Unless **-precise_signed_zero** or **-frigor=accurate** is used, the compiler may transform code in a way that results in the same computed value except for the sign of zero.

Floating-Point Exceptions

No support is provided for the floating-point exceptions accessible through `<fenv.h>`. The functions from the header are provided, but `FE_ALL_EXCEPT` is

defined as 0 to indicate that none of the exception flags can be checked. Furthermore, the expression `math_errhandling & MATH_ERREXCEPT` is 0 to indicate that the math libraries do not use floating-point exceptions to signal errors. Instead, error conditions in the math libraries are signaled by setting `errno`.

Note that many of the situations described by floating-point exceptions are diagnosed if detected at compile time. For examples, see diagnostics 30 (constant out of range), 222 (result out of range) and 1818 (division by zero) in *MULTI: Toolchain Error Messages and Other Diagnostics*.

Rounding Modes

The functions `fegetround` and `fesetround` are provided for getting and setting the rounding mode. Setting the rounding mode via `fesetround` changes the behavior of some library functions, but does not change the behavior of floating-point hardware. As such, the value of `FLT_ROUNDS` is not affected by calls to `fesetround`. Affected library functions include:

- `nearbyintf`, `nearbyint`, `nearbyintl`.
- `rintf`, `rint`, `rintl`.
- `lrintf`, `lrint`, `lrintl`.
- `llrintf`, `llrint`, `llrintl`.
- `strtof`, `strtod`, `strtold`, `atof` (only when parsing hexadecimal input).
- `wcstof`, `wcstod`, `wcstold` (only when parsing hexadecimal input).

Note that while `scanf` with the `%f` specifier is similar in function to the `strtod` family of functions, `scanf` is not affected by the rounding mode. (The C99 standard requires that the `strtod` family of functions round hexadecimal inputs according to the current rounding mode, but has no such requirement for `scanf`.)

For those library functions affected by the rounding mode, we provide the four standard IEEE rounding modes:

1. `FE_UPWARD` — Round upward (toward positive infinity).
2. `FE_DOWNWARD` — Round downward (toward negative infinity).
3. `FE_TOWARDZERO` — Round downward in magnitude (toward zero).

4. `FE_TONEAREST` — Round to the nearest representable value, with ties going to the value with a 0 in the low bit of its mantissa ("roundTiesToEven"). This rounding mode is the default.

Note that in threaded environments, only one rounding mode is stored across all threads.

Complex Numbers

C99 and later provide support for complex numbers using the `_Complex` types. In general, the descriptions of floating-point behavior elsewhere in this section apply equally to `_Complex` types.

The Green Hills toolchain supports `_Imaginary` types, similar to those described in C99 Annex G. These allow simplified formulas for certain operations, which can improve the behavior when NaNs and infinities are involved. For example, multiplying a real value `a` times an `_Imaginary` value `b*I` produces the `_Imaginary` value `(a*b)*I`, without expanding the full formula for `_Complex*_Complex` multiplication. Thus, the following code produces a complex number with real component `3.0` and imaginary component `INFINITY`:

```
double complex z = 3.0 + INFINITY*I;
```

Such formulas are not implemented for all possible expressions involving `_Imaginary` types. In other cases, `_Imaginary` values will simply be promoted to the corresponding `_Complex` type. This normally has little effect on the result, but can produce unexpected NaN results when the input includes NaNs or infinities. To contrast with the above example, multiplying real with `_Complex` values uses the full `_Complex` formula:

```
double complex z = 3.0 + INFINITY*(double complex)I;
```

The above code produces a value with a real component `NAN` and imaginary component `INFINITY`. This is because:

```
INFINITY * (double complex)I
= (INFINITY + 0*I) * (0 + 1*I)
= (INFINITY*0 - 0*1) + (INFINITY*1 + 0*0)*I
= NAN + INFINITY*I
```

Note that the above is not an issue for addition and subtraction.

The following operations use simplified formulas for `_Imaginary` types:

- Multiplication and division of real with `_Imaginary` or `_Imaginary` with `_Imaginary` types.

The following operations instead promote their operands to `_Complex` types:

- Multiplication and division where one operand is `_Complex` and the other is real or `_Imaginary`.
- Macros from **tgmath.h** that permit `_Complex` arguments, when passed at least one `_Complex` or `_Imaginary` argument.

Target and Configuration Caveats

The following is a non-exhaustive list of target- and configuration-specific issues that may cause unexpected floating-point behavior:

On systems where the compiler generates code for a floating-point unit (FPU), computational accuracy will be limited by the underlying hardware. Hardware floating point is used by default.

On systems where floating-point emulation routines are used instead of hardware, computational accuracy may be sacrificed for emulation speed. This is particularly true when dealing with NaNs and subnormals.

Changing a processor's default floating-point behavior (such as its rounding mode) or altering default startup code can impact floating-point calculations in undefined, non-compliant ways.

The exact limitations of hardware floating-point vary greatly across ARM64 processors. For example, many FPUs flush subnormals to zero. Consult your processor's reference manual for specifics.

When the compiler evaluates (folds) floating-point constant expressions at compile time, it will not replicate any non-standard behavior specific to the target. This means that constant-folded expressions may not precisely match the values that would have been calculated on the target, because the compiler uses a more standard method of evaluation. For example, on a target that flushes subnormals to zero, the sum of two subnormals will come out as zero at run-time. But if that same expression is constant-folded, it will result in the actual sum of the given two values.

Other General Notes

The `FP_CONTRACT` and `FENV_ACCESS` `#pragma` directives are recognized in C99 and later dialects, but are both ignored.

C Cautions

Cautions when using C are listed below. This section covers difficulties that may arise from lesser-known aspects of the C language standards.

General Undefined Behavior

- If an operation would cause a signed integer value to overflow or underflow, the result is undefined.
- 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.

Preprocessing a File Twice May Produce Unexpected Output

In certain rare cases, preprocessing a file twice can result in different output than preprocessing it once. That is, preprocessing a file can produce output that would be further changed by preprocessing it again. This follows from the way that preprocessing is specified in the C standard. One example is the following C snippet:

```
const char *s = "?\  
?=";
```

When this is preprocessed, the two lines are joined together due to the backslash on the first line. No trigraph conversion occurs because trigraphs are resolved before line splices, at which point the characters `?` and `?=` are not adjacent. This results in the following preprocessed output:

```
const char *s = "??=";
```

If this snippet is preprocessed again, the trigraph is no longer broken by a line splice, and thus will be converted. This results in:

```
const char *s = "#";
```

For this reason, we recommend against passing preprocessed output files as input to the compiler. Instead, use **-keepcppfiles** or pass the original source file to the compiler twice: first to preprocess and second to compile.

Chapter 12

Green Hills C++

Contents

Specifying a C++ Language Dialect	666
Specifying C++ Libraries	667
Features Supported by Each C++ Dialect	668
Standard C++	669
C++11	672
C++14	676
C++17	677
Embedded C++	679
Extended Embedded C++	681
Standard C++ with ARM Extensions	681
GNU C++	681
Template Instantiation	685
Namespace Support	688
Linkage	691
Post Processing in C++	691
The C++ decode Utility	692
C++03 Implementation-Defined Features	693
Deprecated C++ Headers	698
Non-Conformance with C++ Standards	699
General C++ Cautions	715
C++ Compatibility Across Green Hills Compiler Versions	715

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 143.

Several dialects are available, depending on which OS you are using for your project:

- Standard C++ (see “Standard C++” on page 669).
- C++11 (see “C++11” on page 672).
- C++14 (see “C++14” on page 676).
- C++17 (see “C++17” on page 677).
- Standard C++ with ARM Extensions (see “Standard C++ with ARM Extensions” on page 681).
- GNU C++ (see “GNU C++” on page 681).
- Embedded C++ (see “Embedded C++” on page 679).
- Extended Embedded C++ (see “Extended Embedded C++” on page 681).



Note

Unless stated otherwise, the C++ dialect does not affect non-C++ files.

**Note**

C++ dialects are evolving standards and therefore the Compiler may not implement all known defect resolutions. (This applies to both "Core" and "Library" defects.)

**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, --C++11, --c++11, --C++14, --c++14, --c++17, --C++17, --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

Note that, for EC++ and EEC++, you must pass **--no_large_vtbl_offsets** to use **--exceptions**.

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 315.

Features Supported by Each C++ Dialect

The following table compares the feature set of several C++ dialects. Features that are not listed in this table can be individually controlled through Builder and driver options. For more information about features marked with C++/EC++, see “Differences Between Standard C++ and Embedded C++” on page 679.

Feature	Standard and C++11 or Later	EEC++	EC++
Exception support	Optional	Optional	Optional
Complex numbers	C++	EC++	EC++
Streams and buffers	C++	EC++	EC++
<code>string</code>	C++	EC++	EC++
<code>wchar_t</code>	Yes	Yes	Yes
<code>cin</code> , <code>cout</code>	Yes	Yes	Yes
Template support	Yes	Yes	No
Standard container classes	Yes	Yes	No

Feature	Standard and C++11 or Later	EEC++	EC++
Namespace support	Yes	Yes	No
<code>std:: namespace</code>	Yes	Yes	No
<code>bitset<></code>	Yes	Yes	No
<code><algorithm></code> , <code><functional></code> , <code><memory></code>	Yes	Yes	No
<code>mutable</code> keyword	Yes	Yes	No
<code>static_cast<></code> , <code>const_cast<></code> , <code>reinterpret_cast<></code>	Yes	Yes	No
Multiple inheritance	Yes	No	No
<code>dynamic_cast<></code>	Yes	No	No
<code><locale></code>	Yes	No	No
<code><limits></code>	Yes	No	No
<code>wcin</code> , <code>wcout</code> , etc.	Yes	No	No
<code>cerr</code> , <code>clog</code>	Yes	No	No
<code>iostream</code> , <code>stringstream</code>	Yes	No	No
<code><typeinfo></code>	Yes	No	No

Standard C++

The Standard C++ dialect is defined by *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 one of the following:

- **Standard C++ (Violations Give Errors) (--STD)**
- **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 **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`).

By default, the `--STD` option enables **C/C++ Compiler→C++→Templates→Dependent Name Processing** (`--dep_name`). For more information about `--dep_name`, see “Namespace Support” on page 688.

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 214.
- 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 313.

- 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*.

C++11

The C++11 dialect conforms to *ISO/IEC 14882:2011*. C++11 improves on its predecessor in numerous ways, including enhancements to performance, multithreading, and template metaprogramming. This dialect is recommended for users looking to port existing C++11 code to work with the Green Hills toolchain.

To specify use of this dialect:

Set the **C/C++ Compiler**→**C++ Language Dialect** option to one of the following:

- **C++11 (Violations Give Errors) (--C++11)**
- **C++11 (Violations Give Warnings) (--c++11)**

C++11 Feature Support

The compiler supports nearly all features of C++11. Some features are partially supported, or are only supported in certain circumstances. These features are listed in the section below. For unsupported features, see “Non-Conformance with C++ Standards” on page 699.

Note that most new C11 features also exist in C++11 (see “ISO C11, ISO C17, and ISO C18 Feature Support” on page 586).

C++11 Features with Qualified Support

Threads, Thread-Local Storage, and Synchronization Primitives

Threads, thread-local storage, and synchronization primitives are only supported on some versions of INTEGRITY and u-velOSity (see their associated release notes for more information).

Support for threads, thread-local storage, and synchronization primitives can be added to other operating systems or runtimes, but the process requires implementing custom functions to interact with a compiler-provided interface. Without these

custom functions, linking a C++ program that makes use of threading will cause an error. For example:

```
> driver -o a.out file.cc --c++11

[elxr] (error #412) unresolved symbols: 1
__ghs_provides_cpp11_os_threading_support from
libc++11thnoe.a(threadcall.o)

[elxr] (error) errors during processing
```

For more information about using multithreading features on unsupported operating systems, see the file *install_dir/src/libsys/ind_thrd11.c* included in your compiler installation.



Note

The `__STDCPP_THREADS__` macro is always defined when using C++11. This is true even for environments that do not support threading by default.

Finally, because neither the C11 nor C++11 standard includes a way to specify the priority or stack size of new threads, the Green Hills implementations include a correcting extension. For C++11, use the following prototype:

```
thread(long Priority, size_t StackSize, &function, ...)
```

where ... is a set of parameters appropriate for function. For example:

```
#include <stdio.h>
#include <thread>
void foo(int *ret, int a, int b) {
    *ret = a + b;
}
int main(void) {
    int result = 0;
    std::thread t(8/*priority*/, 0x4000/*stack size*/,
        foo, &result, 2, 3);
    t.join();
    printf("%d\n", result); /* prints "5" */
    return 0;
}
```

For C11 or later, the analogous additional routine is:

```
int thrd_create_priority_stack(thrd_t *thr, long Priority, size_t StackSize,
    thrd_start_t func, void *arg)
```

In contrast, if the standard constructor is used, the operating system must provide the default values for `Priority` and `StackSize`. These values are implemented via the `_Thrd_start` function. For implementation instructions, see the **ind_thr11.c** file. Similarly, see the **install_dir/src/libsys/ind_condvar.c** file for information about implementing condition variables. By default, u-velOSity and INTEGRITY implementations give the newly created thread the same priority as the thread that created it. The default stack size varies, but is at least 0x2000 bytes to account for thread creation overhead.

Note that the thread interface involves the parent thread waiting for a semaphore to be released by the child after it is created. If the child is low priority and the parent is high priority, this may result in a small window during which a high priority thread is pended on a low priority thread. This leaves the potential for a medium priority thread to starve out the higher priority parent thread.

If your operating system support layer uses Highest Locker semaphores, then above should not be an issue. However, the current INTEGRITY and u-velOSity implementations do not use Highest Locker semaphores. A possible workaround for this problem is to create a task at the same or higher priority as the caller and then reduce it to the desired priority using an operating system call from within the task.

Standard Errors Defined in `std::future_errc`

Standard errors defined in `std::future_errc` (for instance, `std::future_errc::broken_promise`) are set in the shared state of `std::promise`, as required by the standard. When exception handling is enabled, an appropriate exception will be thrown. When exception handling is disabled, thrown exceptions will be substituted with calls to `abort()`.

Default Constructor for `std::mutex`

The default constructor for `std::mutex` is `not constexpr`.

Alternate Operator Definition Location

The following operator is defined in `<ostream>` instead of in `<system_error>`:

```
template< class CharT, class Traits >
std::basic_ostream<CharT,Traits>&
operator<<( basic_ostream<CharT,Traits>& os,
const error_code& ec );
```

Over-Alignment with the `alignas` Keyword

Although the `alignas` keyword may be used to over-align a given type or variable, the `new` operator does not honor over-alignment when allocating memory. As is the case with `malloc()`, the `new` operator only guarantees that the memory allocated is sufficiently aligned for any fundamental type. This may not be sufficient for a given over-aligned type, such as when its alignment exceeds that of any fundamental type. Similarly, the alignment of the call stack may be insufficient to guarantee alignment of over-aligned local variables. This can cause the compiler to perform optimizations that overestimate pointer alignment, which can result in misaligned memory accesses that trigger run-time exceptions or performance degradation.

Note that the above alignment restrictions apply equally to `__attribute__((aligned(x)))`. For more information about this attribute, see “Attributes” on page 637.

Important Backward Incompatibilities

C++11 has a number of important backward incompatibilities with its predecessor:

- All narrowing conversions now produce errors.
- String literals can no longer be implicitly converted to `non-const char *`.
- Any macro immediately following a string literal (with no intervening whitespace) will be interpreted as a user-defined literal instead. This issue is most likely to affect code that uses the `print` and `scan` macros from `<inttypes.h>`, such as `PRId32`.
- User-provided destructors are implicitly `noexcept(true)` if the compiler provided destructor would have been `noexcept(true)`. Consequently, user-provided destructors designed to throw exceptions should be explicitly

marked as `noexcept(false)`. Note that `std::terminate` is called if an exception is thrown from a `noexcept(true)` function.

C++14

The C++14 dialect conforms to *ISO/IEC 14882:2014*. C++14 is an incremental improvement over its predecessor, and includes new features such as variable templates, binary literals, and relaxed `constexpr` restrictions. This dialect is recommended for users looking to port existing C++14 code to work with the Green Hills toolchain.

To specify use of this dialect:

Set the **C/C++ Compiler**→**C++ Language Dialect** option to one of the following:

- **C++14 (Violations Give Errors)** (`--C++14`)
- **C++14 (Violations Give Warnings)** (`--c++14`)

C++14 Feature Support

All new C++14 features are supported. Features carried forward from C++11 are unchanged in their support status (see “C++11 Features with Qualified Support” on page 672 and “Non-Conformance with C++ Standards” on page 699). Note that if you plan to use the `<shared_mutex>` header, you must implement condition variables via **ind_condvar.c**. For more information, see “Threads, Thread-Local Storage, and Synchronization Primitives” on page 672.

Important Backward Incompatibilities

C++14 has a number of important backward incompatibilities with its predecessor:

- Digit separators are now allowed. In rare cases, this can cause issues with well-formed C++11 code. To understand why, first note that digit separators allow you to treat the two statements below as equivalent:

```
int x = 1234567;  
int x = 1'234'567;
```


Now consider the following:

```
#define SKIP_FIRST(skipped, ...) __VA_ARGS__

int arr[3] = { SKIP_FIRST(1'3,9'7) }
```

In C++11, `1'3,9'7` is interpreted as three tokens: an integer literal of value 1, a character literal `'3,9'` and an integer literal of value 7. These tokens are skipped by the macro, resulting in the statement being equivalent to:

```
int arr[3] = {}
```

In contrast, C++14 would interpret `1'3,9'7` as an integer literal of value 13, a comma, and an integer literal of value 97. This would resolve as:

```
int arr[3] = {97};
```

- Member functions declared `constexpr` are no longer implicitly `const`. For example, the following code is acceptable in C++11, but not C++14:

```
struct S { constexpr int foo() { return 10;} };
const S c{};
void f() {
    c.foo(); // fails in C++14
}
```

- The `C::gets()` function is no longer a part of the language. Consider substituting `gets()` with a safer mechanism such as `fgets()`.

C++17

The C++17 dialect conforms to *ISO/IEC 14882:2017*. C++17 is an incremental improvement over its predecessor, and includes new features such as hexadecimal floating-point literals, `if constexpr` statements, and enhanced support for overaligned types with `new` expressions. This dialect is recommended for users looking to port existing C++17 code to work with the Green Hills toolchain.

To specify use of this dialect:



Set the **C/C++ Compiler**→**C++ Language Dialect** option to one of the following:

- **C++17 (Violations Give Errors)** (`--C++17`)
- **C++17 (Violations Give Warnings)** (`--c++17`)

C++17 Feature Support

Nearly all C++17 features are supported. Some features are partially supported, or are only supported in certain circumstances. These features are listed in the section below. Features carried forward from C++14 are unchanged in their support status. For unsupported features, see “Non-Conformance with C++ Standards” on page 699.

C++17 Features with Qualified Support

Namespace Member `std::atomic_flag`

Namespace member `std::atomic_flag` may not be lock free. If your target lacks hardware support for atomic operations, `std::atomic_flag` will use a lock.

Important Backward Incompatibilities

C++17 has a number of important backward incompatibilities with its predecessor:

- Trigraphs are no longer supported.
- The `register` keyword is no longer permitted on local variables. Note that this does not affect the Global Registers feature in configurations where this is permitted.
- The `increment` operator may no longer be applied to `bool` types.
- Dynamic exception specifications (such as those used in the function below) are no longer allowed. In configurations where exception handling is enabled, use the `noexcept` keyword instead.

```
void foo(void) throw(int)
{
```

```
        throw 42;  
    }
```

Implementation-Defined Features

- **21.6.5: Hardware interference size** —
std::hardware_constructive_interference_size and
std::hardware_destructive_interference_size are defined to be
alignof(std::max_align_t).

Embedded C++

The Embedded C++ dialect (EC++) is 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 668.

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 668, 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 668.

To specify use of this dialect:



Set the **C/C++ Compiler**→**C++ Language Dialect** option to **Extended Embedded C++ (--ee)**.

Standard C++ with ARM Extensions

The Standard C++ with ARM Extensions dialect is ARM-compliant C++ as defined by *The Annotated C++ Reference Manual* (ARM) by Ellis and Stroustrup (Addison-Wesley, 1990). This includes the templates, exceptions, and anachronisms listed in Chapter 18. This is essentially the same language defined by the language reference for Cfront version 3.0x, 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)**.



Note

This dialect is not supported on INTEGRITY projects.

GNU C++

The GNU C++ dialect closely emulates the C++ 2003 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++)**.



Note

This dialect is not supported on INTEGRITY projects.

GNU C++ Extensions

This dialect enables the extensions listed below, together with those in “GNU C/C++ Extensions” on page 591.

- 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 instantiations of template functions, member functions of template classes, and static data members of template classes (hereafter referred to as *template entities*) generally require special treatment by the compiler in order to ensure that only one copy of each ends up in the program.

The Green Hills C++ compiler provides one method of template instantiation, which is described in the following section.

Link-Once Template Instantiation

In link-once template instantiation, 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 “C Extensions for ANSI C” on page 577). These link-once sections allow for multiple copies of the given entity to be passed to the linker without triggering a multiple definition error. The linker then arbitrarily chooses one of the link-once copies to be included in the output and discards the rest. These discarded copies are mapped to a special `/DISCARD/` segment, which is not included in the final image. It is your responsibility to ensure that all copies of a given entity are compiled with identical

source code and compatible options. Doing otherwise will cause the behavior of the program to depend on the linker's choice of a link-once copy.

Link-once template instantiation is always enabled, but individual instantiations can be affected by the pragma directives mentioned below.

Instantiation Pragma Directives

Instantiation `#pragma` directives can be used to control the instantiation of specific template entities or sets of template entities. 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.

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 688).
- The algorithm that existed before the dependent name lookup was implemented (“Lookup Using the Referencing Context” on page 689).

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 16, “Mixing Languages and Writing Portable Code” on page 837.

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++03 Implementation-Defined Features

- **1.3.2: Diagnostic message** — For a complete list of the C++ compiler diagnostic messages, see the *MULTI: Toolchain Error Messages and Other Diagnostics* 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 57.
- **2.13.2: Character literals** — The **C/C++ Compiler→Special Tokens→Host and Target Character Encoding** option enables recognition of multibyte 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 multibyte 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 613).
- **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()`. 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 “ARM64 Characteristics” on page 26.
- **3.9.1: Fundamental types** — `char` and `unsigned 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 613).
- **4.9: Floating-integral conversions** — Same as the C compiler (see “C Implementation-Defined Features” on page 613).
- **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 “ARM64 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 unsigned by default. Bitfields are unsigned 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 17, “Enhanced asm Macro Facility for C and C++” on page 857.
- **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 57.
 - `#include <h-char-sequence>` — searches all `#include` directories. All identifiers in *h-char-sequence* will be resolved.
 - `#include "q-char-sequence"` — searches the directory containing the source file, 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 747.
- **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`.
- **22.1.2.2: Supported Locales** — Only the "C" locale is supported.
- **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 [69,99] represent years 1969 to 1999. Numbers in the range [100,9999] represent literal offsets from 1900. For example, 1000 would be parsed as 2900. 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>
```

Non-Conformance with C++ Standards

This section covers instances of non-conformance with the various C++ standards. Wherever possible, workarounds and strategies for increasing conformance are provided.

For a variety of practical reasons, some features of C++ that can maximize conformance are turned off by default and may need to be reenabled. These features and the reasons for their default state are listed below:

Exception handling (**--exceptions**):

- Exception handling results in more code, even for functions that neither throw nor catch exceptions.
- Throwing exceptions takes longer than explicit checks for error conditions.
- Code with exceptions is often harder to analyze than code without exceptions.

All C++ Versions

Floating-Point Behavior

For details on standards-relevant aspects of floating-point behavior in C++, see “Floating-Point Behavior” on page 652.

C++03 or Later

Using Digit Characters as Decimal Points or Thousands Separators

The compiler does not strictly follow the C++ standard for number parsing using the `std::num_get` facet. Specifically, the decimal point and the thousands separator are not honored when assigned values that conflict with acceptable digit characters (0 through 9 for decimal; 0..9, a..f, and x for hexadecimal).

For instance, if you create a custom locale where the decimal point is 4, or the thousands separator is 2, the compiler will still recognize these characters as digits.

While such cases have little practical value, the standard requires that these characters be given their assigned role of decimal point or thousands separator.

Note that only custom locales can be affected by this issue. All of the standard C++ predefined locales use non-digit characters for their separators.

Example:

```
struct Punct1: std::numpunct<char>
{
    std::string do_grouping() const { return "\\1"; }
    char do_thousands_sep() const { return '2'; }
    // thousands separator is '2'
    char do_decimal_point() const { return '4'; }
    // decimal point is '4'
};

// ...
// setup a user-specified locale
istringstream iss1;
iss1.imbue(locale(iss1.getloc(), new Punct1));
const num_get<char>&
ng1 = use_facet<num_get<char> >(iss1.getloc());

double d = 0.0;

// parse the string "1234"
// store the 'double' value in 'd'
iss1.str("1234");
ng1.get(iss1.rdbuf(), 0, iss1, err, d);
// d according to standard ought to have the value 13.0
// ('2' should be ignored and '4' is a decimal point)
// d with the Green Hills compiler has the value 1234.0
```

Block Scope Declarations Not Ignored by Qualified Lookup

When performing qualified lookup of a function name, the compiler does not ignore block scope declarations that are known to refer to the same function as the qualified name. Normally this is not an issue, as the same function is found regardless. However, a block scope declaration may differ in that it provides default arguments that the other declaration does not. In this instance, failing to ignore the block scope declaration can cause a change in behavior. To mitigate this issue, the compiler will give error 2039 for any block scope declaration that provides a different default argument than a previous, non-block-scope declaration of the same function.

Virtual Function Thunks Must Have Complete Return Type

In certain cases, the compiler must define a *thunk*: a small function that serves to adjust the `this` pointer in a virtual call before passing control to the proper virtual function. Thunks are often needed when multiple inheritance is used in a class hierarchy, but they have other uses as well. The compiler requires that the return type of a thunk (the same as the virtual function's return type) be defined or instantiated in the translation unit in which the thunk is output. This is required even if the associated virtual function is not defined in the translation unit.

If you see an error message like the following, your code is encountering this limitation in the given translation unit:

```
function "B::foo" requires a virtual table thunk but its return type, "Ret",
is incomplete. Please ensure that "Ret" is defined/instantiated in this
translation unit.
```

A workaround for this issue is to force the return type `Ret` to be defined in this translation unit (if it is a non-template class) or to force it to be instantiated in this translation unit (if it is an otherwise non-instantiated class template).

Below is an example file (**example.cc**) that encounters this limitation, followed by a workaround:

```
struct Ret; /* Declaration, but not definition, of Ret. */

template <class T>
struct TRet { };

struct A {
    virtual Ret foo();
    virtual TRet<int> bar();
};

struct Filler {
    int val;
};

struct B : Filler, A /* multiple inheritance */ {
    virtual ~B();
    virtual Ret foo();          /* override */
    virtual TRet<int> bar();    /* override */
};

B::~~B() { }
```

The above code will result in the following diagnostic output:

```
"example.cc", line 18: error #2045: function "B::bar" requires a virtual table
      thunk but its return type, "TRet<int>", is incomplete. Please ensure
      that "TRet<int>" is defined/instantiated in this translation unit.
      virtual TRet<int> bar();    /* override */
      ^

"example.cc", line 17: error #2045: function "B::foo" requires a virtual table
      thunk but its return type, "Ret", is incomplete. Please ensure that
      "Ret" is defined/instantiated in this translation unit.
      virtual Ret foo();         /* override */
      ^
```

To work around the issue with the non-template class `Ret`:

- Ensure that the `Ret` class is defined in this translation unit. Since most classes are defined in headers this typically just means that you need to `#include` whichever header includes the definition of `Ret`.

To work around the issue with the template class `TRet<int>`:

- Ensure that the class template `TRet` is defined in this translation unit (same as above for the non-template case).
- Ensure that the specific instance `TRet<int>` is instantiated in this translation unit. Below are several code examples of how to force the instantiation of `TRet<int>`. To work around the issue in this case you would choose one of the code approaches below and add that code to the translation unit in question. That would cause the type to be instantiated and thus avoid the issue.

```
/* Force TRet<int> to be instantiated. There are several ways to do this.
   Using the class in sizeof( ) is a helpful method because it does not create
   any objects/variables of the type being instantiated. As a consequence, it
   does not take up space (.data or stack) in the program. */

/* Approach #1 */
/* In C++11 and later (and also in non-strict C++03 mode: --std) the sizeof
   can be wrapped in a dummy static assert so that it can be placed at file
   scope. */
static_assert(sizeof(TRet<int>) > 0, "");

/* Approach #2 */
/* In strict C++03 mode (--STD), a dummy typedef in an anonymous namespace
   can also be used to wrap the sizeof. The anonymous namespace serves to
   prevent the name 'dummy' from clashing with any other names. */
namespace { typedef int dummy[sizeof(TRet<int>)]; }

/* Approach #3 */
/* An explicit instantiation directive can also be used. This is likely the
```

```
most succinct approach, but has a downside: it instantiates all members of
the class. This may not be valid, or may cause unnecessary .text bloat
from the instantiation of many unreferenced member functions. */
template struct TRet<int>;
```

Function Templates Differentiated Only by Dependent Name in Non-Type Template Parameter

In certain cases, the compiler will generate the same mangled name (that is, same link symbol) for two distinct function templates. This can result in either "multiple definition" errors at link time or the use of only one of the definitions by the program (with the other discarded by the link-once mechanism). GHS tested many other compiler implementations and found identical behavior, indicating a likely design flaw in the standard. Nonetheless, a strict reading of the standard seems to indicate that a compiler implementation should treat such templates as separate entities and generate different mangled names for them.

This issue occurs when two function templates satisfy the following criteria:

1. They have identical names.
2. They are declared in the same scope.
3. They have the same number and kind (type, non-type, template) of template parameters.
4. They differ in that a non-type template parameter is declared with a dependent name that might resolve to the same type as that of the non-type template parameter in the other template.

```
template<class T, int>
int func() {return 1;} // #1

template<class T, typename T::other> // If T::other resolves to 'int', then
                                   // has same mangled symbol as #1
int func() {return 2;} // #2
```

Incorrect Lookup for Friend Declarations in Local Classes

If a friend declaration inside a local class uses an unqualified name, the lookup for that name is supposed to consider the surrounding scopes until the first non-class scope is reached. The compiler diverges from this expected behavior in two ways:

1. Friend function declarations that specify the name of member functions in an enclosing class do not properly give friendship to that member function. The declaration will be accepted but the member function will not gain access to protected/private members of the class.
2. Friend function declarations do not consider member functions in the same class as "prior declarations" of the friend function, and the friend function declaration can thus be rejected for not having a matching prior declaration. Such code is suspect, as there is no use for a class to befriend its own member functions, but it nonetheless appears to be legal according to a strict reading of the standard.

```
void foo() {  
  
    // Case #1  
    class A {  
        void member_function() {  
            (void)b->p; // Spurious error: inaccessible  
        }  
        class B {  
            int p;  
            /* This friend declaration does not give A::member_function friendship,  
             * even though it should. A workaround is to use the qualified name  
             * 'A::member_function' in the friend declaration. */  
            friend void member_function();  
        };  
  
        B *b;  
    };  
  
    // Case #2  
    class C {  
        void m() {}  
        /* This friend declaration is pointless, but superficially allowed by the  
         * standard. */  
        friend void m(); // Spurious error - no prior declaration found  
    };  
}
```

C++11 or Later

Namespace Member `std::chrono::steady_clock` and Certain Clock Configurations

Namespace member `std::chrono::steady_clock` functions as an alias to `std::chrono::system_clock`. As such, `std::chrono::steady_clock` will only be steady if `system_clock` is steady.

On INTEGRITY, all clocks (`std::chrono::system_clock`, `std::chrono::high_resolution_clock`, `std::chrono::steady_clock`) are backed by `HighestResStandardClock`. Whether this clock is steady or high resolution is dependent on its BSP implementation. Note that the library interface represents ticks in units of microseconds, which sets an upper bound on the resolution available.

On stand-alone targets, all clocks are unimplemented and backed by the `_Xtime_get_ticks` function in *install_dir/src/libsys/ind_thrd11.c*. (If you provide your own version of *ind_thrd11.c*, the `_Xtime_get_ticks` function can be customized to serve a different purpose.) On u-velOSity, all clocks are backed by the `uv_time_get()` function. The output of `uv_time_get()` is dependent on your configuration.

Collating Elements Not Supported in `std::basic_regex`

Collating elements are not supported in any of the named locales used by the compiler (currently only "C"). As such, the library-supplied specializations of `std::regex_traits::lookup_collatename` will always return an empty string. Consequently, any use of these specializations in instances of `std::basic_regex` are unsupported.

Stack Checking Not Performed By `<regex>` Components

When constructing or matching against a regular expression, `std::regex` and other variations of `std::basic_regex` do not check for stack overflow conditions. As such, the `error_stack` code is not used. Care should be taken to ensure sufficient stack space for constructing and matching against a given regular expression.

Class `std::random_device`

Class `std::random_device` is not backed by hardware in any configuration. It is implemented as a simple wrapper around `rand()`.

Type Template Parameter Pack Used as Types for Non-Type Template Parameters (Referenced But Not Expanded)

The compiler may reject a type template parameter pack when used as the set of types for a list of non-type template parameters. The compiler will issue a spurious error: "parameter pack ... was referenced but not expanded".

Example:

```
template <int> struct A { };
// line 4: spurious error: parameter pack "T" was referenced
// but not expanded
template <typename ...T, T... vals> int foo(A<vals...>);
int i = foo<int>(A<0>{});
```

SFINAE ("Substitution Failure is Not an Error") Behavior When Instantiating Alias Templates

The compiler is expected to follow SFINAE rules when instantiating an alias template from within an SFINAE context. (This circumstance could arise, for example, when considering a template function as part of overload resolution.) Instead, the compiler will diagnose substitution problems as "hard" errors, rather than silently discarding the template from consideration.

Example:

```
struct bar { };

template <typename T1> struct test {
    template<typename T2> using member = typename T2::undefined_type;
};

// SFINAE should occur here when T = bar, since bar
// bar::undefined_type does not exist
template<class T> constexpr bool baz( typename test<T>::template
                                   member<T>* ) { return false; }

// This function template should be selected by the below call.
template<class T> constexpr bool baz(...) { return true; }
```

```
// This incorrectly gives a hard error: class "bar" has no member
// "undefined_type"
static_assert(baz<bar>(0), "");
```

Failure to Extend Friendship

In limited circumstances, the compiler may fail to extend friendship to members of specializations of class templates. Specifically, if a given class *C* gives friendship to member *f* of class template *A*, then the compiler may not extend friendship to the corresponding member *f* in partial or explicit specializations of *A*. Normally, this failure would only result in spurious access errors, but if such code were used in a C++11 SFINAE context then it may result in a valid function template being spuriously ignored during overload resolution of a function call. If the spuriously ignored function template was the most appropriate overload, and there are other valid (though less appropriate) functions in the overload set, then it may result in the wrong function being called. Green Hills has never observed this possible SFINAE failure, and has only observed spurious access errors inside its internal test environment. Example:

```
template <class T> struct A {
    void f();
};

template <> struct A<char> {
    void f();
};

class C {
    template <class T> friend void A<T>::f();

private:
    int i;
};

void A<char>::f() {
    C c;
    c.i = 0; // Rejected with error: member "C::i" is inaccessible
    (void) c;
}
```

Function Parameters Declared with `decltype` of Previous Function Parameter Pack

In certain cases, the compiler may reject valid calls to a function template or generic lambda. When this happens, it will give a spurious error:

```
no instance of function template [...] matches [...].
```

This issue will occur when three criteria are met:

1. A parameter (`args1`) of a function template (or parameter of a generic lambda) is a function parameter pack.
2. Another parameter (`args2`) is declared with a `decltype` referencing that pack (`args1`).
3. The function template (or generic lambda) is called in such a way that the template parameter pack (`Ts`) contains zero elements.

Example:

```
template <typename ...Ts>
void foo(Ts ...args1, decltype(args1) ...args2) {}

void bar()
{
    foo<>(); // Spurious error: no instance of function template
           // 'foo' matches ...
}
```

C++14 or Later

Signedness of Plain Bitfields

Prior to C++14, it was implementation-defined whether a plain integer type bitfield was signed or unsigned. ("Plain" here refers to being neither explicitly signed or unsigned.) C++14 removed this freedom and now specifies that such plain bitfields must be signed. The compiler does not implement this change; it instead continues to determine whether a plain bitfield is signed or unsigned based upon the use of `--signed_fields` or `--unsigned_fields`. Either of these options may be the default, depending on your target processor. For more information, see “Signedness of Bitfields” on page 204.

Core Issue 1778 Partially Implemented

The compiler only partially implements the resolution to Core Issue 1778. Specifically, if a function that is explicitly defaulted has an explicit exception-specification that is incompatible with that of the implicit declaration—and the function was explicitly defaulted on the first declaration—then the compiler will correctly define the function as deleted. However, if the function was instead explicitly defaulted on a secondary declaration, then the compiler will incorrectly accept the declaration and define the function normally, except that the explicitly specified exception-specification will be used regardless of the implicit exception-specification. The correct behavior, which is not implemented, would be to reject such a declaration.



Note

The P1286R2 paper directly reverts the changes to Core Issue 1778, so this behavior may be subject to further changes in future Compiler versions.

Core Issue 1330 Fix

Core issue 1330's standard fix (delaying instantiation of `noexcept` specifiers) is not implemented.

C++17 or Later

Use of filesystem Header

The `<filesystem>` header file is not supported.

Select Policies from the algorithm Header

Parallel and unsequenced execution policies used for some of the functions in `<algorithm>` are not supported. If any of these policies are used, they will be ignored and execution will be performed sequentially.

Special Mathematical Functions

Special mathematical functions, such as `assoc_legendre()`, are not supported.

Select Using-Declaration typenames

A using-declaration that uses the same `typename` to refer to the same type will be rejected by the compiler:

```
struct X { typedef int type; };
struct Y { typedef int type; };

struct Z : X, Y {
    using X::type;
    using Y::type; // error: "type" has already been declared in the current
                  // scope
}
```

Select Address Constants Used as Non-Type Template Arguments

Address constants used as non-type template arguments are limited to `&<id>`:

```
template<typename T, T v> struct X {};

X<const int*, &n>();
X<const int*, &n + 1 - 1>(); // invalid nontype template argument
                          // of type "const int *"
```

Select Template Fold Expressions

Template fold expressions inside of casts using functional notation may give spurious errors:

```
template<int ...N>
void f() {
    extern int k;
    (void) (k = ... = N); // Okay
    void ((k = ... = N)); // Spuriously rejected
}
```

Template Deduction Guides with extern or static

No error message is given when `extern` or `static` is used on a template deduction guide (the storage-class specifier is ignored):

```
extern A() -> A<int>; // no error is given
static B() -> B<int>; // no error is given
```

constexpr if and Boolean Conversions

Core issue 2320, described by P1401r0, is implemented by the compiler in a way that allows contextually converted constant `bool` expressions. This means the following are accepted (similar to a non-`constexpr` `if` statement):

```
if (5) {} // Okay
if constexpr ((bool)5) {} // Okay
if constexpr (5) {} // Okay with the proposal of CWG2320 and the
                  // Green Hills compiler
```

Rewording Inheriting Constructors

Core issue 1941 "Rewording inheriting constructors" (resolved with paper P0136R1) is not implemented.

Keywords and Alternative Tokens in Attribute-Tokens

Keywords and alternative tokens are not permitted in attribute-tokens:

```
int b [[ using bitand: bitor, xor]]; // not permitted (bitand, bitor, and xor
                                     // are all keywords)
```

Select Uses of Init-Capture

An init-capture with the same name as a variable in an enclosing scope cannot be used in the trailing return type:

```
int val = 0;
auto lam1 = [val = val + 1](auto arg) ->
    decltype(val) { return 0; }; // error: init-capture cannot be captured
                              // here
lam1(0);
```

Select Local Static constexpr Initializer Lists

Local static constexpr `std::initializer_list` variables are rejected when the initializer overtly references the `std::initializer_list` constructor, rather than doing so implicitly with a braced list:

```
void foo(void) {
    static constexpr std::initializer_list<int> x1{ 1, 2, 3 }; // Accepted
    static constexpr std::initializer_list<int> x2 = { 1, 2, 3 }; // Accepted
    static constexpr std::initializer_list<int> x3 =
        std::initializer_list<int>{ 1, 2, 3 }; // Rejected
}
```

Address Taking of Capture-By-Reference in a constexpr Lambda

Taking the address of a capture-by-reference of a local variable is not allowed in a constexpr lambda:

```
void f() {
    int n;
    constexpr bool B = [&]{ return &n; }() == &n; // error: expression must
                                                // have a constant value
}
```

Address Taking of a Structured Binding in a constexpr Expression

Taking the address of a structured binding is not allowed in a constexpr expression:

```
auto& [ x, y ] = i;
static_assert (&x == &i[0]); // error: expression must have a constant
                             // value
```

Select Uses of std::visit

Namespace member `std::visit` only supports zero or one `std::variant` arguments. Passing more than one argument will result in a `static_assertion` failure.

Floating-Point Overloads

Floating-point overloads for `std::from_chars` are not supported.

Changes to `std::common_type`

Changes to `std::common_type` as described in P0435R1 and P0548R1 are not implemented for library working issues 2463 and 2465.

Select Instances of Using the `decltype` of an auto template Parameter

Providing a default `template` argument for a non-type `template` parameter will cause spurious deduction failures if the type of the non-type `template` parameter is the `decltype` of a `template` parameter with a placeholder type (for example, with the `auto` keyword). This occurs even if the default `template` argument is not used.

```
template<auto A, decltype(A) B = 0> struct SubstFailure {};  
// cannot deduce "auto" template parameter type "auto" from "int"  
  
SubstFailure<0, 0> s;
```

Exception Specifier of `std::invoke`

Namespace member `std::invoke` has the exception specifier of `noexcept(false)` instead of `noexcept(std::is_nothrow_invocable_v<F, Args...>)`.

Select Auto-Typed Non-Type Template Parameters

An error may be given erroneously when using auto-typed non-type `template` parameters that are pointers-to-members of previous `template` types.

```
template<typename T, auto *T::*p> struct B {};  
template<typename T, auto *T::*p0> char &f(B<T, p0> b);  
  
struct X { int *p; };  
  
char &u = f(B<X, &X::p>()); // Error given here
```

Select Fold Expressions Within Alias Templates

Some fold expressions within alias templates (P1 in the example below) are not resolved when the invocation of the alias template (P2) takes in a parameter pack and adds an extra argument.

```
template<int> struct typ1 {};  
template<int ...I> using alias = typ1<(I + ...)>; // P1  
struct typ2 {  
    template<int ...J> void func(alias<0, J...>);    // P2  
};  
void use(typ2 param) {  
    param.func<4>({});  
}
```

Compiler Issues Diagnostics for Uninstantiated Discarded Statement of "if constexpr"

When parsing an `if constexpr` statement inside of a template, the compiler is supposed to ignore the discarded statement that corresponds to an `if constexpr` condition that evaluates to `FALSE` (as long as the condition was not value-dependent after instantiation). This means that the compiler should not issue diagnostics for any semantically invalid code in the discarded statement. (Note that syntax is still checked regardless.)

However, when parsing an `if constexpr` statement inside of a generic `lambda`, which itself is inside of a function template (`foo()`), the compiler will issue diagnostics for any invalid code that it sees regardless of whether the condition was value-dependent.

```
template<class T>  
void foo() {  
    auto lambda = [](auto p) {  
        /* This condition is no longer value-dependent after instantiation of  
         * foo<T> and it evaluates to FALSE, so we should not diagnose anything  
         * in the discarded statement. If the condition were still dependent (for  
         * example, because it referenced 'p') then it would be okay to issue the  
         * diagnostics, but that is not the case. */  
        if constexpr (sizeof(T) == 1) {  
            static_assert(sizeof(T) == 1, ""); // Do not diagnose static_assert  
        }                                     // fail  
    };  
}  
  
int main() {  
    foo<int>();  
}
```

Unable to Deduce Template Arguments From Pack Expansion Involving Multiple Packs

When performing class template argument deduction, the compiler will be unable to deduce arguments from a pack expansion if both of the following criteria are true:

1. The pack expansion simultaneously expands multiple packs.
2. At least one of the packs is from an enclosing template (such as an enclosing class template).

When this occurs, argument deduction will fail and the template will be removed from consideration. This may result in either an error (such as "no instance of constructor ... matches ...") or the compiler selecting a different valid constructor or deduction guide. This different constructor may be a lower ranking match than the template that was spuriously removed from consideration.

```
template<typename ...T> struct B {  
    template<T ...V> B(const T (&...p)[V]) {  
    }  
};  
int arr[3];  
B<int> b1(arr);           // Okay - does not use class template argument  
                           // deduction  
B b2(arr);               // Spuriously rejected
```

General C++ Cautions

General cautions when using C++ are listed below. This section covers difficulties that may arise from lesser-known aspects of the C++ language standards.

- When using a placement `new[]` expression, ensure that the pointer you use in the expression is sufficiently aligned for any data type. Passing an insufficiently aligned pointer will result in undefined behavior.

C++ Compatibility Across Green Hills Compiler Versions

In general, object files containing C++ code are not binary compatible with object files created by a different version of the Green Hills Compiler. A significant effort is made to minimize these incompatibilities, but they may be the unavoidable result

of necessary bug fixes or of changes to the implementation of the standard library (for example, as prescribed by defect reports against the C++ standard).

When upgrading to a new version of the Green Hills Compiler, you should rebuild all files containing C++ code to ensure binary compatibility. Note that the Green Hills Compiler may implement version checks between releases, but the absence of such checks does not indicate binary compatibility.

Chapter 13

Coding Standards

Contents

GHS Standard Mode	718
MISRA C 1998	718
MISRA C 2004	721
MISRA C 2012	723
Creating Your Own Coding Standards with Coding Standard Profiles	728

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 899.

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 253).

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 engineering 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 149.

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 these 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 752).

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 forms listed below. Note that some rules may be unenforceable or only partially enforceable by the toolchain. For these and other details, see “MISRA C 1998” on page 149.

- **--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 through 10.
- **--saferc=7,10** — Enables checking of rules 7 and 10.

MISRA C 1998 Rules Enforced by the Linker

The following MISRA C 1998 rules are enforced (either wholly or partially) by the linker. As such, their enforcement in the linker is not governed by the `#pragma ghs startnomisra` directive.

- Rule 11
- Rule 22
- Rule 23
- Rule 25
- Rule 70
- Rule 118
- Rule 121
- Rule 122
- Rule 123
- Rule 124
- Rule 125
- Rule 126

- Rule 127

MISRA C 2004

The MISRA C 2004 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 engineering 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:

- *required* rules — mandatory requirements placed on the programmer that will, by default, generate an error if enabled and breached.
- *advisory* rules — 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 159.

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 forms listed below. Note that some rules may be unenforceable or only partially enforceable by the toolchain. For these and other details, see “MISRA C 2004” on page 159.

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*[,*n*1]...** — 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 example:

- **--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.

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**.

For a list of all supported MISRA C 2004 rules, see “MISRA C 2004” on page 159.

MISRA C 2004 Rules Enforced by the Linker

The following MISRA C 2004 rules are enforced (either wholly or partially) by the linker. As such, their enforcement in the linker is not governed by the `#pragma ghs startnomisra` directive.

- Rule 5.1
- Rule 8.7
- Rule 8.9
- Rule 8.10
- Rule 16.2
- Rule 20.4
- Rule 20.7
- Rule 20.8
- Rule 20.9
- Rule 20.10
- Rule 20.11
- Rule 20.12

MISRA C 2004 Implementation

For information about the MISRA C 2004 implementation, see “MISRA C 2004 and 2012 Implementation” on page 726.

MISRA C 2012

The MISRA C 2012 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 engineering practices in the development of embedded automotive systems. You can find details in the book *Guidelines For The Use Of The C Language In Critical Systems, Motor Industry Software Reliability Association, March 2013*, as well as *MISRA C 2012 Amendment 1, April 2016*. For more information, see the MISRA Web site [<http://www.misra.org.uk/>].

There are 173 guidelines, represented in four categories by the toolchain:

- *mandatory* rules — requirements placed on the programmer that will, by default, generate an error if enabled and breached.
- *required* rules — requirements placed on the programmer that will, by default, generate an error if enabled and breached. Required rules are different from

mandatory rules in that an organization may formally allow a deviation of the former, but not the latter. For more information, see Section 5.4 of *Guidelines For The Use Of The C Language In Critical Systems*, Motor Industry Software Reliability Association, March 2013.

- *advisory* rules — suggestions to the programmer that will, by default, generate a warning if enabled and breached.
- *directives* — guidelines for situations where a complete test for compliance is not possible without additional information.

To change which rules will generate diagnostics, see the options in “MISRA C” on page 147. The MISRA guidelines are organized into 23 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 guidelines, see “MISRA C 2012” on page 175.

Enforcing MISRA C 2012 Guidelines

A single driver option allows you to enable any or all of the MISRA guidelines. It may be used in any of the forms listed below. Note that some guidelines may be unenforceable or only partially enforceable by the toolchain. For these and other details, see “MISRA C 2012” on page 175.

- **--misra_2012=all** — Enables checking of all guidelines.
- **--misra_2012=none** — [default] Disables checking of all guidelines.
- **--misra_2012=n[,n1]...** — Enables checking of a comma-separated list of guidelines.
- **--misra_2012=-n[,n1]...** — Disables checking of a comma-separated list of guidelines.
- **--misra_2012=n-n1** — Enables checking of guidelines *n* through *n1*.
- **--misra_2012=-n-n1** — Disables checking of guidelines *n* through *n1*.

For example:

- **--misra_2012=all,-R2.3** — Enables checking of all guidelines except rule 2.3.
- **--misra_2012=R2.3-R4.2** — Enables checking of rules 2.3 through 4.2.
- **--misra_2012=R2.3,R4.2** — Enables checking of rules 2.3 and 4.2.

- **--misra_2012=-D4.1,-D4.3** — Disables checking of directives 4.1 and 4.3.

For effective use of this option, note the following:

- The argument string is parsed from left to right. In the case of conflicting guidelines and/or ranges, guidelines specified later will overrule those specified earlier. For example, **--misra_2012=R3.1,-R3.1** will disable rule 3.1, while **--misra_2012=-R3.1,R3.1** will enable rule 3.1. The same logic also applies to multiple invocations of the **--misra_2012** option.
- When the **--diag_*** options and **--misra_2012** options are both used in a single project, both options are passed to the compiler and both are used.
- If a MISRA guideline is enabled based on the **--misra_2012** options, the compiler follows MISRA-specific logic to determine if a diagnostic should be output. The compiler then typically applies the **--diag_*** options to determine the severity and suppression of the diagnostic. If the **--diag_*** severity is higher than the MISRA severity, the diagnostic will be given in the non-MISRA diagnostic form. If the **--diag_*** severity is lower than the MISRA severity, the MISRA severity will be lowered to the **--diag_*** severity. Note that it is not always possible to change a guideline's severity using **--diag_***.
- Linker-validated MISRA guidelines are not affected by the **--elxr_diag_*** options. Use MISRA severity options instead, such as **--misra_req**.
- MISRA guidelines enforced with linker or run-time error checking must be enabled/disabled at the link step or project level to take effect. These guidelines cannot be controlled on a per-file basis.
- Coding standard profiles, such as the Green Hills Standard Mode, are implemented using the same mechanism as **--diag_*** and therefore have a similar interaction with **--misra_2012**.

For a list of all supported MISRA C 2012 rules and directives, see “MISRA C 2012” on page 175.

MISRA C 2012 Guidelines Enforced by the Linker

The following MISRA C 2012 guidelines are enforced (either wholly or partially) by the linker. As such, their enforcement in the linker is not governed by the `#pragma ghs startnomisra` directive.

- Rule 8.6
- Rule 8.7
- Rule 8.9
- Rule 17.2

MISRA C 2012 Guidelines Enforced at Run-Time

For a list of MISRA 2012 guidelines enforced at run-time, see “MISRA C - Enforcement and Run-Time” on page 148.

MISRA C 2004 and 2012 Implementation

The Green Hills tools provide options to control the severity of messages that alert you to violations of MISRA C guidelines. Most of the messages are given at compile- and link-time, although some are generated at run-time. Because run-time checking can adversely affect code performance, an option is provided to disable it (see “MISRA C - Enforcement and Run-Time” on page 148).

Most of the guidelines 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 guidelines 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 guideline if it exceeds the default severity for that message.

In some cases a MISRA guideline 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 guideline might cause multiple diagnostics to be issued if it violates more than one of the checks used to implement the guideline. 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 guideline without disabling it entirely. For example, if you have MISRA 2004 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 guidelines by disabling features or changing logic that is not specific to MISRA rule checking. If you enable checking for one of these guidelines, and your code is in violation, 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 guideline directly, the error message will not contain a reference to that guideline. These MISRA guidelines are not affected by `#pragma ghs startnomisra` or `#pragma ghs endnomisra` (see "Green Hills Extension Pragma Directives" on page 752).

For example, you might enable checking for MISRA 2004 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 options change other command line controllable options. These MISRA options can override the explicit settings of other command line options, even if these other options appear later on the command line.

Using MISRA C 2012 with Operating Systems and Libraries

If an operating system or library has MISRA violations, the following techniques are available:

- Headers from the code may be wrapped in `#pragma ghs startnomisra/endnomisra`. For example:

```
#pragma ghs startnomisra
#include <INTEGRITY.h>
#pragma ghs endnomisra
```

- Link-time checks can be disabled to prevent the linker from issuing errors for external code linked in. For example:

```
--misra_2012=-R8.6,-R8.7,-R8.9,-R17.2 # Disable linker checks
```

- Any rules that cannot be suppressed with `#pragma ghs startnomisra/endnomisra` can be disabled at the project level. For example:

```
--misra_2012=-R1.2 # INTEGRITY.h requires extensions
--diag_suppress=177 # Rule 2.2 violation in INTEGRITY.h
--diag_suppress=1546 # Rule 1.1 violation in armv7_general.h
```

To set up MISRA 2012 with INTEGRITY, please contact Green Hills Support.

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 900
- **misra1998.csp** — (provided as an example; does not fully implement MISRA 1998)

You can also create your own (see “Coding Standard Profile Syntax” on page 729).

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 253).

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 259 and “Green Hills Extension Pragma Directives” on page 752).



Note

MISRA compliance checking is available through a set of driver options. For more information, see “MISRA C 1998” on page 149, “MISRA C 2004” on page 159, and “MISRA C 2012” on page 175.

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: Toolchain Error Messages and Other Diagnostics*.
- *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**:

```
// Place external declarations in headers  
7    "My Standard 1.1" my_tag my_tag_1_1  
1816 "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:

```
> ccarm64 -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:

```
> ccarm64 -c test.c --coding_standard=my_profile.csp
"test.c", line 2: warning #1816-D: My Standard 1.2: external declaration
    should be in header file
    void bar();
        ^
```

You can specify a tag for the **--diag_suppress**, **--diag_remark**, **--diag_warning**, or **--diag_error** option to override the levels defined in the severity section:

```
> ccarm64 -c test.c --coding_standard=my_profile.csp --diag_warning=my_tag
"test.c", line 1: warning #7-D: My Standard 1.1: unrecognized token
    #define foo @
                ^
```

```
"test.c", line 2: warning #1816-D: My Standard 1.2: external declaration
    should be in header file
    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:

```
> ccarm64 -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 #1816-D: My Standard 1.2: external declaration
    should be in header file
    void bar();
        ^
```

If you pass a tag that does not exist in any specified profile, the compiler issues a warning:

```
> ccarm64 -c test.c --coding_standard=my_profile.csp --diag_warning=bad_tag
Command-line warning #1966-D: unused coding standard tag or malformed error
    number bad_tag
```

```
"test.c", line 2: warning #1816-D: My Standard 1.2: external declaration
    should be in header file
    void bar();
        ^
```


Chapter 14

Macros, Pragma Directives, and Intrinsic

Contents

Predefined Macro Names	734
Pragma Directives	747
Intrinsic Functions	762
Static Assertions	772

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++
- 201103L — in C++11
- 201402L — in C++14
- 201703L — in C++17
- 1 — in 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 (or later) mode

`__STDC_VERSION__`

The value of this macro is:

- 199901L — in C99
- 201102L — in C11
- 201710L — in C17
- 201710L — in C18
- 199409L — in ANSI C and GNU C
- Undefined in C++

<code>__STDC_HOSTED__</code> The value of this macro is: • 1 — in C++ and all modes of C
<code>__TIME__</code> A string literal representing the local time of the current compilation, which is 8 characters long, and in the form: <i>hh:mm:ss</i> . For example: "11:46:51"

Language Identifier Macros

These macros identify the language used:

Primary Language Macros

<code>__LANGUAGE_ASM__</code> Language is assembly.	*
<code>__LANGUAGE_C__</code> Language is C.	*
<code>__LANGUAGE_CXX__</code> Language is C++.	*

C Language Macros

<code>__GNUC__</code> GNU C or C++ mode (-gcc or --g++).
<code>__GNUC_GNU_INLINE__</code> GNU C or GNU C99 mode (-gcc or -gnu99) when GNU-style inlining is used.
<code>__GNUC_STDC_INLINE__</code> GNU C or GNU C99 mode (-gcc or -gnu99) when GNU99-style inlining is used.
<code>__Japanese_Automotive_C__</code> Japanese Automotive C (see“Japanese Automotive C Extensions” on page 595).
<code>__STRICT_ANSI__</code> Language is Strict ANSI C (-ANSI) or Strict C99 (-C99). Undefined in C++.

C++ Language Macros

<code>__cplusplus</code>	Language is C++ (this is for backwards compatibility with some older C++ implementations). When writing new code, use <code>__cplusplus</code> .
<code>__embedded_cplusplus</code>	Language is Embedded or Extended Embedded C++.
<code>__EMBEDDED_CXX</code>	Language is Embedded C++ (for example, option <code>--e</code>).
<code>__EMBEDDED_CXX_HEADERS</code>	The Embedded C++ header files and libraries are used (for example, options <code>--el</code> or <code>--ele</code>).
<code>__EXTENDED_EMBEDDED_CXX</code>	Language is Extended Embedded C++ (ESTL) (for example, option <code>--ee</code>).
<code>__EXTENDED_EMBEDDED_CXX_HEADERS</code>	The Extended Embedded C++ header files and libraries are used (for example, options <code>--eel</code> or <code>--eele</code>).
<code>__STANDARD_CXX</code>	Language is Standard C++.
<code>__STANDARD_CXX_HEADERS</code>	The Standard C++ header files and libraries are used (for example, options <code>--stdl</code> or <code>--stdle</code>).

Green Hills Toolchain Identification Macros

<code>__EDG__</code>	C or C++ compilers.
<code>__ghs__</code>	Any Green Hills Software compiler or preprocessor. *
<code>__ghs_asm</code>	Indicates that the Green Hills assembler is in use.
<code>__GHS_REVISION_DATE</code>	The date of the compiler, represented as a string similar to <code>__DATE__</code> .
<code>__GHS_REVISION_VALUE</code>	Represents the date of the compiler build in the <code>time_t</code> format.

<code>__GHS_VERSION_NUMBER</code>
Represents the version of the toolchain as a six-digit decimal integer. For example, in Compiler 2016.5.4, this macro would be defined as 201654.
<code>__gnu_asm</code>
Indicates that the GNU assembler is in use.
<code>__unix_asm</code>
Indicates that the UNIX assembler is in use.

ARM64-Specific Predefined Macro Names

The following table lists predefined macros specific to the ARM64 processor family.

<code>__ARM64__</code>
ARM Aarch64 architecture.
<code>__ARM_NEON__</code>
Supports the NEON SIMD instruction set.

Endianness Macros

One of these symbols is always defined to specify the endianness of the target processor.

<code>__BIG_ENDIAN__</code>
Big Endian byte order.
<code>__LITTLE_ENDIAN__</code>
Little Endian byte order.

Data Type Macros

These macros specify the size and signedness of certain data types.

Sizes

<code>__CHAR_BIT</code>
Defined as the size of <code>char</code> in bits.

<code>__FUNCPTR_BIT</code>
Defined as the size of a function pointer in bits.
<code>__INT_BIT</code>
Defined as the size of <code>int</code> in bits.
<code>__LONG_BIT</code>
Specifies the size of <code>long</code> in bits.
<code>__LLONG_BIT</code>
Specifies the size of <code>long long</code> in bits. Defined only if the <code>long long</code> type is supported.
<code>__PTR_BIT</code>
Specifies the size of a pointer in bits.
<code>__REG_BIT</code>
Specifies the size of an integer register in bits.
<code>__SHRT_BIT</code>
Specifies the size of <code>short</code> in bits.
<code>__WCHAR_BIT</code>
Specifies the size of <code>wchar_t</code> in bits.

Signedness

<code>__Field_Is_Signed__</code>
<code>__Field_Is_Unsigned__</code>
Bitfields are signed or unsigned.
<code>__Ptr_Is_Signed__</code>
<code>__Ptr_Is_Unsigned__</code>
Pointers are signed or unsigned.
<code>__Char_Is_Signed__</code>
<code>__SIGNED_CHARS__</code>
<code>__Char_Is_Unsigned__</code>
<code>__CHAR_UNSIGNED__</code>
Type <code>char</code> is signed or unsigned.

<code>__WChar_Is_Signed__</code> <code>__WChar_Is_Unsigned__</code> Type <code>wchar_t</code> is signed or unsigned.

Floating-Point Macros

These macros specify the manner in which floating-point operations are performed.

<code>__DOUBLE_HL__</code>	*
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.	
<code>__IeeeFloat__</code>	*
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.

<code>__NoFloat__</code>	*
No floating-point mode.	
<code>__SoftwareDouble__</code>	*
Double precision floating point uses integer instructions.	
<code>__SoftwareFloat__</code>	*
All floating point is done with integer instructions.	

Half-Precision Floating-Point Macros

The following macros are provided in `float.h` only when `-half_precision_type` is specified.



Note

The macros `HLF_HAS_SUBNORM`, `HLF_DECIMAL_DIG`, and `HLF_TRUE_MIN` are only defined in C11/C++11 and later language modes.

HLF_HAS_SUBNORM	Defined to 1 if <code>__fp16</code> supports subnormals, 0 if not, -1 if subnormal support cannot be determined at compile time.
HLF_MANT_DIG	Number of bits in the mantissa.
HLF_DECIMAL_DIG	Minimum number of decimal digits to represent <code>__fp16</code> values in decimal.
HLF_DIG	Maximum number of decimal digits to represent decimal values in <code>__fp16</code> .
HLF_MIN_EXP	Minimum negative integer such that 2 raised to one less than that power is a normal number in <code>__fp16</code> .
HLF_MIN_10_EXP	Minimum negative integer such that 10 raised to that power is a normal number in <code>__fp16</code> .
HLF_MAX_EXP	Maximum integer such that 2 raised to one less than that power is a finite number in <code>__fp16</code> .
HLF_MAX_10_EXP	Maximum integer such that 10 raised to that power is a finite number in <code>__fp16</code> .
HLF_MAX	Maximum finite number in <code>__fp16</code> .
HLF_EPSILON	Difference between 1 and the least value greater than 1 representable in <code>__fp16</code> .
HLF_MIN	Minimum positive normal number in <code>__fp16</code> .
HLF_TRUE_MIN	Minimum positive number in <code>__fp16</code> . This is a subnormal value if subnormals are supported, otherwise it is the same as <code>HLF_MIN</code> .

MISRA 1998 Macros

These macros relate to the MISRA 1998 variant. They will not function with other MISRA versions.

`__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 1998 rules, see “MISRA C 1998” on page 149.

`__ONLY_STANDARD_KEYWORDS_IN_C`

MISRA rule No. 1

Memory Model Macros

These macros relate to various different forms of memory model.

Global Register Macros

`__GlobalRegisters`

Corresponds to the `-globalreg=n` option, where *n* is a non-negative 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`

The value 16, which is the alignment in bytes of the most-aligned type (a NEON type).

`__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. 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.

`__TIMESTAMP__`

A string literal representing the date and time of the last modification of the current source file. For example, "Tue Aug 4 01:00:07 2020".

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.

<code>__GHS_Inline_Memory_Functions</code>
Inlining of C Memory Functions (see “Inline C Memory Functions” on page 287).
<code>__GHS_Inline_String_Functions</code>
Inlining of C String Functions is enabled (see “Inline C String Functions” on page 288).
<code>__GHS_Optimize_Debug__</code>
Optimizations that do not affect debugging are enabled (-Odebug). For more information, see “Optimization Strategy” on page 119.
<code>__GHS_Optimize_General__</code>
Optimizations that improve size and performance are enabled (-Ogeneral). For more information, see “Optimization Strategy” on page 119.
<code>__GHS_Optimize_Inline</code>
Two-pass inlining is enabled (see “Indicating Functions for Two-Pass Inlining” on page 331).
<code>__GHS_Optimize_Max_Debug__</code>
All optimizations, including inlining, are disabled (-Omaxdebug). For more information, see “Optimization Strategy” on page 119.
<code>__GHS_Optimize_More_Debug__</code>
Optimizations that do not affect debugging are enabled (-Omoredebug), with a greater emphasis on debugging ability than -Odebug . For more information, see “Optimization Strategy” on page 119.
<code>__GHS_Optimize_None__</code>
All optimizations are disabled (-Onone). For more information, see “Optimization Strategy” on page 119.
<code>__GHS_Optimize_Size__</code>
Optimizations that improve size and performance are enabled, with more emphasis on size than speed (-Osize). For more information, see “Optimization Strategy” on page 119.
<code>__GHS_Optimize_Speed__</code>
Optimizations that improve performance and size are enabled, with more emphasis on speed than size (-Ospeed). For more information, see “Optimization Strategy” on page 119.

C++ Macros

These macros relate to various features of C++.

<code>__ARRAY_OPERATORS</code>	Defined when array <code>new</code> and <code>delete</code> are enabled (that is, <code>operator new[]</code> and <code>operator delete[]</code>).
<code>__AUTO_TYPE</code>	Defined when <code>auto</code> is treated as a type specifier for C++11-style type deduction rather than as a storage class specifier. For example, when using the option --auto_type .
<code>_BOOL</code>	Defined when <code>bool</code> is recognized as a basic type. For example, when using the option --bool .
<code>__EDG_IMPLICIT_USING_STD</code>	Defined when the standard namespace <code>std</code> is used implicitly. For example, when using the option --using_std .
<code>__EDG_RUNTIME_USES_NAMESPACES</code>	Indicates that the C++ Run-time libraries use namespaces.
<code>__EXCEPTION_HANDLING</code> <code>__EXCEPTIONS</code>	Indicates C++ compiler is running in a mode that allows exception handling.
<code>__NAMESPACES</code>	Indicates C++ namespaces are accepted.
<code>__RANGE_BASED_FOR</code>	Defined when support for C++11-style range-based <code>for</code> loops is enabled. For example, when using the option --range_based_for .
<code>__RTTI</code>	Indicates Run-Time Type Identification code accepted.
<code>_WCHAR_T</code>	Defined if <code>wchar_t</code> is a keyword.

Operating System-Specific Macros

These macros identify the target operating system.

INTEGRITY Macros

`__INTEGRITY`

INTEGRITY real-time operating system.

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`

*

Target version of Microsoft Windows.

`__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.

<code>__Ptr_Is_32</code>
Pointers are 4 bytes.
<code>__Ptr_Is_64</code>
Pointers are 8 bytes.
<code>__Reg_Is_32</code>
CPU has 32-bit registers.
<code>__Reg_Is_64</code>
CPU has 64-bit registers.
<code>__WChar_Is_Int__</code>
Type <code>wchar_t</code> is <code>int</code> or unsigned <code>int</code> .
<code>__WChar_Is_Long__</code>
Type <code>wchar_t</code> is <code>long</code> or unsigned <code>long</code> .
<code>__WChar_Is_Short__</code>
Type <code>wchar_t</code> is <code>short</code> or unsigned <code>short</code> .

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. For Green Hills `#pragma` directives, comments are not allowed between the `ghs` keyword and the keyword that follows. If present, such comments will cause the syntax checks to fail. 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:

<code>#pragma alignfunc (n)</code>
Instructs the compiler to increase the next function to be <i>n</i> -byte aligned, where <i>n</i> is 1, 2, 4, 8, 16, 32, 64, or 128. Specifying a value less than the current alignment has no effect.
<code>#pragma alignvar (n)</code>
Instructs the compiler to make the next variable at least <i>n</i> -byte aligned, where <i>n</i> 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 685.

```
#pragma message "string"
```

```
#pragma message("string")
```

Prints *string* to the error output while compiling the file.

```
#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})
```



Note

Support for structure packing in C++ is limited. See the release notes for more information.

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.



Warning

When building applications for INTEGRITY, you must ensure that there is no structure packing in effect when including an INTEGRITY or system header.

```
#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. 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.



Note

Specifying a C++ mangled symbol name for *foo* is deprecated. To make C++ symbols weak, use `__attribute__((weak))`.

```
#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 **Target→Text and Data Placement→Allocation of Uninitialized Global Variables** option in “Text and Data Placement” on page 90).

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 **Target→Text and Data Placement→Allocation of Uninitialized Global Variables** option in “Text and Data Placement” on page 90).

`#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 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



Note

This pragma only works properly if you also build with `-check=alloc`. See “Run-Time Memory Checks” on page 138.

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)
```

For information about using run-time checks, see the **Debugging→Run-Time Error Checks** option in “Debugging Options” on page 132, 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 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 542.

```
#pragma ghs entry_exit_history=option
```

```
#pragma ghs entry_exit_history push option
```

```
#pragma ghs entry_exit_history pop
```

Enables or disables function entry/exit history 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 function entry/exit history in header files where you probably do not want to modify the including files's state. For example:

```
#pragma ghs entry_exit_history=on
// The state is: (0) on
void foo1() { } // Function entry/exit on
```

```
#pragma ghs entry_exit_history push off
// The state is: (0) on (1) off
void foo2() { } // Function entry/exit off
```

```
#pragma ghs entry_exit_history=default
// The state is: (0) on (1) default
void foo3() { } // Function entry/exit state based on compiler options
```

```
#pragma ghs entry_exit_history pop
// The state is: (0) on
void foo4() { } // Function entry/exit on
```

For more information, see “Function Entry/Exit History” on page 45. This feature is licensed separately from the Green Hills toolchain.

```
#pragma ghs instrument_prologue_epilogue=option
```

```
#pragma ghs instrument_prologue_epilogue push option
```

```
#pragma ghs instrument_prologue_epilogue pop
```

Enables or disables prologue/epilogue instrumentation on a function-by-function basis, where *option* has the same values and meanings as the identically named argument used with `#pragma ghs entry_exit_history`. For more information, see “Instrument Prologue/Epilogue” on page 96.

```
#pragma ghs volatile_sideeffect=on|off|default
```

Directs the compiler to make an assumption about plain volatile objects, treating them as either volatile `__sideeffect` objects (on) or volatile `__nosideeffect` objects (off) within the indicated code block. To end the code block, use `volatile_sideeffect=default`. This `#pragma` directive does not override existing `__sideeffect`/`__nosideeffect` keywords. That is, a variable declared with `volatile __sideeffect` or `volatile __nosideeffect` will always be treated as such.

```
#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 550.

```
#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 541.

```
#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 nowarning n
#pragma ghs endnowarning [n]
```

Temporarily suppress diagnostic number *n*. (A tag or coding standard profile tag may be given in place of the diagnostic number.) You may use multiple and/or nested `#pragma ghs nowarning` directives, but each should have a correctly matched `#pragma ghs endnowarning` in order to deactivate its effect. For example:

```
#pragma ghs nowarning 1
#pragma ghs nowarning 2
... /* 1 and 2 are suppressed here */
#pragma ghs endnowarning 2
#pragma ghs endnowarning 1
/* 1 and 2 are active again here */
```

The `#pragma ghs endnowarning` directive can optionally take a diagnostic number (or tag) as an argument. If this argument is present, it must match the most recently active `#pragma ghs nowarning` directive, otherwise a warning will be issued and the `#pragma ghs endnowarning` directive will have no effect. If this argument is not present, the directive applies to the most recently active `nowarning` directive. For example:

```
#pragma ghs nowarning 3
#pragma ghs nowarning 4
#pragma ghs nowarning 5
... /* 3, 4, and 5 are suppressed here */
#pragma ghs endnowarning
#pragma ghs endnowarning 4
/* 3 is suppressed here, but 4 and 5 are active again */
```

The complete set of diagnostic numbers and tags is also listed in the *MULTI: Toolchain Error Messages and Other Diagnostics* book. For more information about coding standard profiles, see Chapter 13, “Coding Standards” on page 717.

To display diagnostic numbers with diagnostic messages, pass `--display_diag_number`.



Note

The location where `#pragma ghs nowarning` must be active is dependent on the implementation of the diagnostic in question, and is not always at the position shown in the diagnostic output.

In some cases `#pragma ghs nowarning` must be active at the end of the translation unit. In these cases, attempts to suppress the diagnostic will still fail if improperly implemented. For example, consider a source file with the following code:

```
#pragma ghs nowarning n
// code that triggers diagnostic n
#pragma ghs endnowarning n
```

This will still result in the diagnostic being issued since the `#pragma ghs endnowarning`

n will reactivate the diagnostic at the end of the translation unit.

```
#pragma ghs Ostring
```

```
#pragma ghs ZO
```

`#pragma ghs Ostring` turns on optimizations, while `#pragma ghs ZO` turns them off. (This includes many of the optimizations implied by command line options.) The optional *string* can contain any or all of the following letters:

- L — Loop optimizations
- M — Memory optimizations
- S — Small (but Slow) optimizations

Although the possible values of *string* are similar to several optimization-related command line options, their effects may be different.



Note

You can enable optimizations by omitting *string* from the directive.

For information about Builder and driver optimization options, see “Optimization Options” on page 119. For descriptions of many of our optimizations, see Chapter 4, “Optimizing Your Programs” on page 325.

```
#pragma ghs reference sym
```

Creates a reference to symbol *sym*, which can force the symbol to be pulled in from a library. It will also prevent *sym* from being deleted by the optimizations `-delete` and `-data_delete`. This `#pragma` is not intended for use with static symbols.



Note

For C++, the symbol name given must match the mangled name of the function or variable.

```
#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. Note that you should not attempt to remap multiple default sections to the same custom section. Additionally, you should not choose a custom section name that has the same name as a default section. For more information, see “Custom Program Sections” on page 62.

```
#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 “Using Wholeprogram Interprocedural Optimizations” on page 338.

```
#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.

If block coverage is enabled for this file, functions within this directive will still be displayed as uncovered within MULTI.

For more information about coverage profiling, see “Enabling Instrumented Coverage or Performance Profiling” on page 47.

```
#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.

For more information about controlling inlining, see “Inlining Optimizations” on page 326.

```
#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. Note that these directives only prevent the compiler from performing MISRA checking; they do not prevent MISRA checking in other parts of the toolchain (see “MISRA C 1998 Rules Enforced by the Linker” on page 720, “MISRA C 2004 Rules Enforced by the Linker” on page 722, and “MISRA C 2012 Guidelines Enforced by the Linker” on page 725).

These directives do not disable the run-time checking that is done for certain MISRA rules (for a list of such rules, see “MISRA C - Enforcement and Run-Time” on page 148).

For more information, see “MISRA C 2004” on page 159 and “MISRA C 2012” on page 175

```
#pragma ghs startnostrongfpnr
```

```
#pragma ghs endnostrongfpnr
```

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 544.

```
#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 542 and “Annotating Assembly Functions for **gstack**” on page 541.

```
#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.

```
#pragma ghs unroll=n
```

Provides the compiler a hint for the unroll factor to be used with the next loop. (This directive does not influence whether the compiler actually performs loop unrolling.) The specified unroll factor, *n*, must be a positive power of 2 and less than or equal to 64.

This directive is only allowed within the body of a function. When multiple `#pragma ghs unroll` directives apply to a single loop, only the most recent directive will take effect. If a `#pragma ghs vectorize` directive applies to the same loop as a `#pragma ghs unroll` directive, the vectorize directive will be given precedence over the unroll directive.



Note

This directive is not currently supported with C++11 range-based for-loops; when applied to a range-based for-loop, the directive will be accepted (that is, not trigger a diagnostic) but will otherwise be ignored.

Conditional Pragma Directives

This feature is deprecated and may be removed in future versions.

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 Level” on page 132).
- `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 119).
- `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(aligned) PRAGMA( alignvar(aligned) )
#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(aligned) \
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 that 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/arm64/arm64_ghs.h`**, and can be included as follows:

```
#include <arm64_ghs.h>
```

```
void *__builtin_frame_address(unsigned int level);
```

Only valid when used in functions compiled with target-walkable stack support enabled. Returns the address of the stack frame for the current function (if the *level* parameter is a constant value of 0) or for the caller of the current function (if the *level* parameter is a constant value of 1).

```
void *__builtin_return_address(unsigned int level);
```

If the *level* parameter is the constant 0, returns the return address of the current function. In a function compiled with target-walkable stack support enabled, the *level* argument may be set to the constant 1, indicating the return address of the caller of this function.

```
void *__get_stack_pointer(void);
```

Returns the current value of the stack pointer.

Exception Intrinsic

```
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. Specifically, `__DI` sets the `I` and `F` bits of the `DAIF` register, disabling IRQ and FIQ interrupts. Conversely, `__EI` clears the `I` and `F` bits of the `DAIF` register, enabling IRQ and FIQ interrupts.

```
unsigned long long __DIR(void);
```

```
unsigned long long __EIR(void);
```

Like `__DI()` and `__EI()`, these functions disable and enable interrupts. In addition, they return an unsigned long long key that represents the state of interrupts prior to their operation. For more information, see the row above.

```
void __RIR(unsigned long long key);
```

Takes a *key* previously returned by `__DIR()` or `__EIR()` and restores interrupts to the state it represents. When used with `__DIR()` and `__EIR()`, interrupts can be safely enabled/disabled and then later restored to their original state when that state is not known at compile time.

Arithmetic Operation Intrinsic

```
unsigned long long __ADD_WITH_CARRY64(unsigned long long a, unsigned
long long b, unsigned char carry_in, unsigned char *carry_out);
```

Takes 64-bit integer *a*, 64-bit integer *b*, 8-bit integer *carry_in* (0 or 1) and returns their 64-bit sum. The *carry_out* value (0 or 1) is returned by pointer.

```
unsigned int __CLZ32(unsigned int a);
```

```
unsigned int __CLZ64(unsigned long long a);
```

`__CLZ32` takes a 32-bit integer argument and returns the count of leading zeros, which is a number from 0 to 32.

`__CLZ64` takes a 64-bit integer argument and returns the count of leading zeros, which is a number from 0 to 64.

```
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.

```
unsigned long long __MULUH64(unsigned long long a, unsigned long long b);
```

```
signed long long __MULSH64(signed long long a, signed long long b);
```

The `__MULUH64(a, b)` function takes as arguments two 64-bit unsigned integers *a* and *b* and returns the high 64-bit half of their 128-bit unsigned product.

The `__MULSH64(a, b)` function takes as arguments two 64-bit signed integers *a* and *b* and returns the high 64-bit half of their 128-bit signed product.

Synchronization Primitives

```
unsigned char __LDXR8(unsigned char *addr);
```

```
unsigned short __LDXR16(unsigned short *addr);
```

```
unsigned int __LDXR32(unsigned int *addr);
```

```
unsigned long long __LDXR64(unsigned long long *addr);
```

Generates a load exclusive synchronization instruction for the address *addr* that attempts to claim exclusive access to an address of 1, 2, 4, or 8 bytes, respectively.

```
int __STXR8(unsigned char value, unsigned char *addr);
```

```
int __STXR16(unsigned short value, unsigned short *addr);
```

```
int __STXR32(unsigned int value, unsigned int *addr);
```

```
int __STXR64(unsigned long long value, unsigned long long *addr);
```

Generates a store exclusive synchronization instruction that attempts to store *value* to *addr*. Returns 0 on success or 1 otherwise.

NEON Intrinsics

The following table lists the supported NEON vector data types:

User Type Name	Internal Structure Name	Elements	Base Type
<code>int8x8_t</code>	<code>struct __simd64_int8_t</code>	8	signed char
<code>int16x4_t</code>	<code>struct __simd64_int16_t</code>	4	signed short
<code>int32x2_t</code>	<code>struct __simd64_int32_t</code>	2	signed int
<code>int64x1_t</code>	<code>struct __simd64_int64_t</code>	1	signed long long
<code>uint8x8_t</code>	<code>struct __simd64_uint8_t</code>	8	unsigned char
<code>uint16x4_t</code>	<code>struct __simd64_uint16_t</code>	4	unsigned short

User Type Name	Internal Structure Name	Elements	Base Type
uint32x2_t	struct __simd64_uint32_t	2	unsigned int
uint64x1_t	struct __simd64_uint64_t	1	unsigned long long
float32x2_t	struct __simd64_float32_t	2	float
poly8x8_t	struct __simd64_poly8_t	8	8-bit polynomial
poly16x4_t	struct __simd64_poly16_t	4	16-bit polynomial
int8x16_t	struct __simd128_int8_t	16	signed char
int16x8_t	struct __simd128_int16_t	8	signed short
int32x4_t	struct __simd128_int32_t	4	signed int
int64x2_t	struct __simd128_int64_t	2	signed long long
uint8x16_t	struct __simd128_uint8_t	16	unsigned char
uint16x8_t	struct __simd128_uint16_t	8	unsigned short
uint32x4_t	struct __simd128_uint32_t	4	unsigned int
uint64x2_t	struct __simd128_uint64_t	2	unsigned long long
float32x4_t	struct __simd128_float32_t	4	float
float64x2_t	struct __simd128_float64_t	2	double
poly8x16_t	struct __simd128_poly8_t	16	8-bit polynomial
poly16x8_t	struct __simd128_poly16_t	8	16-bit polynomial

For implementation details of the NEON vector data types and prototypes of supported intrinsics, see *[install_dir/include/arm64/arm64_neon.h](#)*.

You can use these intrinsics to speed up existing code with NEON vector operations. For example:

```
// Without NEON intrinsics
void vector_multadd_original(short *dest, short *srca, short *srcb, short base)
{
    for (int i = 0; i < 64; i++)
        dest[i] += srca[i] * srcb[i] - base;
}

// With NEON intrinsics
#include <arm64_neon.h>
void vector_multadd_neon(short *dest, short *srca, short *srcb, short base)
{
    int16x8_t basevect = vdupq_n_s16(base);
    for (int i = 0; i < 64; i += 8) {
        int16x8_t a = vld1q_s16(srca + i);
```

```
    int16x8_t b = vld1q_s16(srcb + i);
    int16x8_t d = vld1q_s16(dest + i);
    d = vmlaq_s16(d, a, b);
    d = vsubq_s16(d, basevect);
    vst1q_s16(dest + i, d);
}
}
```

Cryptographic Extension Intrinsics

For implementation details and prototypes of supported intrinsics for cryptographic extension instructions, see *install_dir/include/arm64/arm64_neon.h* and *install_dir/include/arm64/arm64_crypto.h*.

Endianness Intrinsics

<code>unsigned long long __REV16(unsigned long long Rm);</code> Reverses the bytes in all four of the 16-bit words in the 64-bit parameter, using the REV16 instruction.
<code>unsigned long long __REV32(unsigned long long Rm);</code> Reverses the bytes in both the upper and lower 32-bit words in the 64-bit parameter, using the REV32 instruction.
<code>unsigned long long __REV64(unsigned long long Rm);</code> Reverses the bytes in the 64-bit parameter, using the REV instruction.
<code>signed int __REVSH(signed int Rm);</code> Reverses the bytes of the lower 16 bits and sign-extends the result to 32 bits.

Memory Barrier Optimization Intrinsics

The following intrinsics require you to include *ghs_barrier.h*.

In the descriptions of the following intrinsics, *low-level operations* are defined as assembly statements, intrinsics that access non-user register state, and intrinsics that may affect loads or stores. While these intrinsics affect the ordering of instructions, they do not introduce memory fence instructions into the instruction stream.

```
void __memory_changed(void);
```

Adds a barrier that forces all loads, stores, and low-level operations sequenced before the barrier to occur in instruction execution order before the barrier, and all loads, stores, and low-level operations sequenced after the barrier to occur in instruction execution order after the barrier. This intrinsic should be used in lieu of the other barriers.

The following intrinsics are implemented for compatibility with other toolchains. We do not recommend using them.

```
void __force_stores(void);
```

Adds a barrier that forces all stores and low-level operations sequenced before the barrier to occur in instruction execution order before the barrier, and all stores and low-level operations sequenced after the barrier to occur in instruction execution order after the barrier.

```
void __schedule_barrier(void);
```

Adds a barrier over which the pipeline optimizer will not reschedule instructions. However, other optimizations may schedule across the barrier, perhaps defeating the point of the barrier.

Barrier Instruction Intrinsics

```
void __ISB(void);
```

The Instruction Synchronization Barrier flushes the processor's pipeline.

```
void __DMB_OPT(__ghs_c_int__ option);
```

The Data Memory Barrier ensures ordering of observation of memory accesses.

This intrinsic inserts a memory fence instruction into the instruction stream, but will not prevent the compiler from reordering other instructions across it. For an intrinsic suited to such a purpose, see `__memory_changed` in “Memory Barrier Optimization Intrinsics” on page 766.

Possible values for *option* are:

- `__BARRIER_SY`
- `__BARRIER_ST`
- `__BARRIER_LD`
- `__BARRIER_ISH`
- `__BARRIER_ISHST`
- `__BARRIER_ISHLD`
- `__BARRIER_NSH`
- `__BARRIER_NSHST`
- `__BARRIER_NSHLD`
- `__BARRIER_OSH`
- `__BARRIER_OSHST`
- `__BARRIER_OSHLD`

```
void __DSB_OPT(__ghs_c_int__ option);
```

The Data Synchronization Barrier ensures completion of memory accesses.

This intrinsic inserts a memory fence instruction into the instruction stream, but will not prevent the compiler from reordering other instructions across it. For an intrinsic suited to such a purpose, see `__memory_changed` in “Memory Barrier Optimization Intrinsics” on page 766.

Possible values for *option* are the same as for `__DMB_OPT()` above.

Hint Intrinsics

```
void __NOP(void);
```

The No-Op hint inserts a `nop` instruction into the instruction stream. This does nothing save advance the program counter to the next instruction.

```
void __SEV(void);
```

The Send Event hint signals other threads that may be in the Wait For Event state.

```
void __WFE(void);
```

The Wait For Event hint allows the target to enter a low-power state pending a signal.

```
void __WFI(void);
```

The Wait For Interrupt hint allows the target to enter a low-power state pending an interrupt.

```
void __YIELD(void);
```

The Yield hint signals a good point to suspend a thread in a multi-threaded environment.

System Intrinsics

```
unsigned long long __MRS(__ghs_c_int__ regnum);
```

Returns the current value of the system register specified by *regnum*. The *regnum* argument should be specified as follows:

```
__GENERIC_SYS_REG(op0, op1, CRn, CRm, op2)
```

where *op0*, *op1*, *CRn*, *CRm*, and *op2* are the numerical values that the architecture uses to encode the system register. These values can be obtained from the ARM64 Architecture Reference Manual.

In assembly, these values would be used when specifying a register in the form *Sop0_op1_CCRn_CCRm_op2*. For example, the floating-point control register (FPCR) can be named in assembly as *S3_3_C4_C4_0*. To use it with this intrinsic, pass

```
__GENERIC_SYS_REG(3, 3, 4, 4, 0).
```

Alternately, system register names from the ARM64 Architecture Reference Manual may be used directly in assembly (for example, *FPCR* instead of *S3_3_C4_C4_0*). System registers may also be specified in C by prefixing a name with a double underscore (such as *__FPCR*). Note that specifications in C should follow the formatting given in Chapters C5 and D7 in the ARM64 Architecture Reference Manual.

```
void __MSR(__ghs_c_int__ regnum, unsigned long long Rn);
```

Moves the 64-bit value of *Rn* into the system register specified by *regnum*. The *regnum* argument should be given in the same form as for *__MRS*.

```
void __MSR_IMM(__ghs_c_int__ pstatefield, __ghs_c_int__ val);
```

Moves the constant value *val* to selected bits of the *PSTATE*. Corresponds to the *msr* (immediate) instruction.

Possible values for *pstatefield* are:

- *__PSF_SPSel*
- *__PSF_DAIFFSet*
- *__PSF_DAIFFClr*

```
void __SYS(__ghs_c_int__ op1, __ghs_c_int__ Cn, __ghs_c_int__ Cm,
__ghs_c_int__ op2, unsigned long long Rt);

void __SYS_XZR(__ghs_c_int__ op1, __ghs_c_int__ Cn, __ghs_c_int__ Cm,
__ghs_c_int__ op2);
```

The `__SYS` intrinsic executes a system instruction, specified by *op1*, *Cn*, *Cm*, and *op2*, on source value *Rt*. The effect of this intrinsic is dependent on the specific chip it is run on.

The `__SYS_XZR` intrinsic is equivalent to `__SYS`, but with a source value of zero. That is, it corresponds to executing the `sys` instruction with no source register specified.

Built-In Intrinsics

The following intrinsics do not require a prototype or header file:

```
bool __builtin_constant_p(arg);
```

Evaluates to true if the argument is recognized by the compiler as representing a known constant value. The 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.

```
void __ghs_noprofile_func(void);
```

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.

```
TYP __builtin_speculation_safe_load  
    (const volatile TYP *ptr,  
     const volatile void *lower,  
     const volatile void *upper,  
     const volatile void *cmpptr);
```

Where *TYP* is any integer or pointer type.

This intrinsic is used to mitigate against cache-speculation exploits based on conditional branches (also known as Spectre variant 1). It inserts a barrier and will return a value loaded from *ptr* only if *cmpptr* is within the following range: *lower* <= *cmpptr* < *upper*. Outside of this range, the behavior is undefined.

Example:

```
int foo(int x[256], int n) {  
    if ((n >= 0) && (n < 256))  
        return __builtin_speculation_safe_load(&x[n], x, x+256, x+n);  
    return 0;  
}
```

For more information, see the Spectre white paper [<https://spectreattack.com/spectre.pdf>].

Floating-Point Bit Manipulation Intrinsics

Prototypes for these intrinsics are included in **math.h**.

```
uint64_t __ghs_getdbits(double value);
```

Returns the double-precision input value in a 64-bit integer. For example, `-0.0` will return `0x8000000000000000`.

```
double __ghs_setdbits(uint64_t bits);
```

Returns the double-precision value as encoded in the 64-bit input value. For example, `0x8000000000000000` will return `-0.0`.

```
uint32_t __ghs_getfbits(float value);
```

Returns the single-precision value as encoded in the 32-bit integer. For example, `-0.0F` will return `0x80000000`.

```
float __ghs_setfbits(uint32_t bits);
```

Returns the single-precision value as encoded in the 32-bit input value. For example, `0x80000000` will return `-0.0F`.

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 15

Libraries and Header Files

Contents

The Green Hills Header Files and Standard Libraries	775
Advanced Library Topics	779
Green Hills Standard C Library Functions	790
Customizing the Run-Time Environment Libraries and Object Modules ...	825

This chapter describes the standard language libraries provided with the Green Hills tools for ARM64. 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 **libname.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 9, “The ax Librarian” on page 465.

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***). Note that this reference is provided as an additional resource, not as an official extension of the toolchain's documentation.
- P. J. Plauger, *Dinkum Abridged Library Reference* (for EC++ and EEC++; included with your distribution, at ***install_dir/ecxx/html/index.html***). Note that this reference is provided as an additional resource, not as an official extension of the toolchain's documentation.



Note

The Green Hills standard libraries that are not used in supported RTOS projects (such as INTEGRITY or ThreadX) contain only the following licenses:

- Green Hills Software license (see example in *install_dir/ansi/errno.h*).
- Edison Group license (see “Edison Group License” on page 992).
- Dinkumware Ltd. license (see *install_dir/scxx/html/license.txt*).



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 ARM64 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/arm64** — Contains header files for ARM64 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 ARM64 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:

- 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).

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 (**cxarm64**), or if you specify **-language=cxx** (see also “Specifying C++ Libraries” on page 667). 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++ and extended embedded C++, without exceptions.
 - **libsedge** — Run-time support for standard C++ and extended embedded C++, with exceptions.
 - **libedgnoe** — Run-time support for embedded C++, without exceptions.
 - **libedge** — Run-time support for embedded C++, with exceptions.
- In language modes for C++11 or later, one of the following libraries is used, depending on selected options. For more information, see “Threads, Thread-Local Storage, and Synchronization Primitives” on page 672.
 - **libcpp11thnoe.a** — C++11 threading library, without exceptions.
 - **libcpp11the.a** — C++11 threading library, with exceptions.

Standard C Libraries are searched in the following order (unless you pass the **-nostdlib** driver option):

- **libfmalloc**, **libarena_malloc**, **librtmalloc**, **libdbmem** — Provides the functions `malloc`, `calloc`, `realloc`, `aligned_alloc`, and `free`. At most, one of these libraries is used, depending on the value of the **-malloc_version** option:
 - **-malloc_version=legacy** — None (this `malloc` implementation is provided in **libansi**).
 - **-malloc_version=smallblock** — **libfmalloc**.
 - **-malloc_version=arena** — **libarena_malloc**.
 - **-malloc_version=realtime** — **librtmalloc**.

- **-malloc_version=memcheck** — **libdbmem**. Note that this library is also selected if you pass **-check=alloc** or **-check=memory**.
- **libwchar*** — Standard C library functions that use `wchar_t`.
- **libansi** — Standard C library. This is the primary C library.
- **libc11.a** — Standard C11 library functions.
- **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 828).
- **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 829).
- **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 **-G**, unless **-nostdlib** is also passed.

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 stand-alone C library defaults to a less buffered I/O mode, described below. This provides some of the performance benefits of buffering without the risk of output loss due to characters left in a buffer. Among the I/O functions, only `ungetc()` calls `malloc()`, and only when `ungetc()` is called twice on a stream with no intervening read. Only `scanf()` variants which operate on a stream call `ungetc()`. If these functions are not used in a program, `malloc()`

can be avoided, possibly reducing the program's size by a few kilobytes. Three buffering modes are supported:

- Full buffering — Characters are only written to the file when the buffer is full or when calling `fflush()` or `fclose()`. This mode can be enabled for a file by calling either `setbuf()` with a buffer of at least `BUFSIZ` bytes or `setvbuf()` with a mode of `_IOFBF` and a non-NULL buffer. For example:

```
char buffer[BUFSIZ];
setbuf(f, buffer);

char buffer[8196];
setvbuf(f, buffer, _IOFBF, sizeof(buffer));
```

- Partial buffering — Characters are buffered as with full buffering, except that `fflush()` is called at the end of `fputs()` and all forms of `printf()`. This mode can be enabled for a file by calling `setvbuf()` with a mode of `_IOPBF` and a non-NULL buffer. For example:

```
char buffer[256];
setvbuf(f, buffer, _IOPBF, sizeof(buffer));
```

- Less buffered I/O — In this mode, no characters are left in a buffer once a library function returns, eliminating both the risk of output loss and the need to flush buffers when a file is closed or the program exits. To achieve some of the same performance benefits, functions that write multiple characters to a file will do so using fewer output system calls. This is done by either passing the string to `write()` directly (for functions like `fputs()`) or by building the output string on the stack before calling `write()` (for functions like `printf()`). One function call often requires only one output system call. The output functions that have this behavior are: `fprintf()`, `fwrite()`, `fputs()`, `puts()`, `printf()`, `vprintf()`, `vfprintf()`, and their wide character forms. Among input functions, only `fread()` has similar behavior.

This mode is the default, and can be explicitly specified for a file by calling `setvbuf()` with either a mode of `_IONBF` or a buffer of `NULL`.

Line buffering (where the buffer is flushed whenever a newline character is written) is not provided on stand-alone targets. If `setvbuf()` is called with a mode of `_IOLBF`, it will be treated the same as `_IOPBF` (partial buffering).

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. Otherwise, files are not flushed or closed upon program termination.

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. For `stdout` and `stderr`, they enable line buffering in INTEGRITY or partial buffering on stand-alone targets.

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", 0x7fffffffffffffffffLL);
```

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.

Calling printf from Assembly Code

To reduce code size, the portion of `printf` responsible for printing floating-point values is only linked in when the compiler determines that it might be used. This is done automatically for C and C++ programs. However, the compiler will not analyze handwritten assembly calls to `printf`. To ensure that the necessary library code is linked in, you must use the following directive:

```
.need __gh_float_printf
```

This directive should be manually placed inside every function that calls `printf` with a floating-point argument. Otherwise, `printf` may print floating-point arguments as `(float)` rather than formatting their values.

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 51 and Chapter 5, “The DoubleCheck Source Analysis Tool” on page 357.

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 784
- “Using File Locks to Ensure I/O Is Thread-Safe” on page 785
- “Allocating Space in Each Thread for Static Data” on page 786
- “Using Alternate Thread-Safe Library Functions” on page 787

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**.

<code>void __ghsLock(void);</code> Acquires global lock. Blocks until the lock becomes available.
<code>void __ghsUnlock(void);</code> Releases global lock.
<code>void __gh_lock_init(void);</code> Initializes the global lock data structure before it is used.
<code>void __ghs_flock_file(void *addr);</code> Acquires per-file lock <i>addr</i> . Blocks until the lock becomes available.
<code>void __ghs_funlock_file(void *addr);</code> Releases per-file lock <i>addr</i> .
<code>int __ghs_ftrylock_file(void *addr);</code> Attempts to acquire per-file lock <i>addr</i> . 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.
<code>void __ghs_flock_create(void **addr);</code> <code>void __ghs_flock_destroy(void *addr);</code> Initializes and cleans up per-file lock data structure. The <code>__ghs_flock_create()</code> implementation may store a pointer in <i>addr</i> . That pointer is passed to the other per-file lock calls.
<code>void *__ghs_GetThreadLocalStorageItem(int specifier);</code> 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 787.

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_thrld.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
<code>__ghs_TLS_asctime_buff(0)</code>	<code>char asctime_buff[30]</code>
<code>__ghs_TLS_tmpnam_space(1)</code>	<code>char tmpnam_space[L_tmpnam]</code>
<code>__ghs_TLS_strtok_saved_pos(2)</code>	<code>char *strtok_saved_pos</code>
<code>__ghs_TLS_Errno(3)</code>	<code>int errno</code>
<code>__ghs_TLS_gmtime_temp(4)</code>	<code>struct tm gmtime_temp</code>
<code>__ghs_TLS___eh_globals(5)</code>	<code>void *__eh_globals</code>

Specifier (Value)	Example Data Item Declaration
<code>__ghs_TLS_SignalHandlers (6)</code>	<code>SignalHandler</code> <code>SignalHandlers[_SIGMAX]</code>

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.

Customizing the Value of FOPEN_MAX

The C input/output library has a limit on how many files can be open simultaneously, expressed in the macro `FOPEN_MAX`. By default, the Green Hills libraries use a moderate size limit intended to accommodate most applications. To increase the limit, follow the instructions below.

On stand-alone:

- When creating your project, select the option to customize **libsys.a** (see “Creating a Project with a Customizable Run-Time Environment” on page 825).
- Add the following build option to **libsys.a**:
-D__GHS_OVERRIDE_FOPEN_MAX=N, where *N* is the new value of `FOPEN_MAX`.
- Pass the same option when building your application.

On INTEGRITY, there is no **libsys.a**, so an equivalent file must be added manually. To do this, create a new stand-alone project with customizable **libsys.a** and then

manually copy the file **ind_io_arrays.c** into your main project. Finally, as described above, pass `-D__GHS_OVERRIDE_FOPEN_MAX=N` to the whole project.

Note the following:

- To function correctly, the option **-D__GHS_OVERRIDE_FOPEN_MAX** must be uniformly applied to all parts of a program. This includes **libsys.a** or the file **ind_io_arrays.c**.
- This feature is not supported with the INTEGRITY shared libraries.
- This feature is not supported on u-velOSity.
- The standard streams `stdin`, `stdout`, and `stderr` are included in the limit on open files. Thus, when overriding `FOPEN_MAX`, set it slightly higher than the actual number of files you intend to open. Some operating systems may open additional files, which are also included in this limit.
- This feature only affects the limit imposed by the toolchain libraries. Your operating system may impose its own limit on how many files a program can open simultaneously, which will not be affected.
- Raising `FOPEN_MAX` also increases the size of certain arrays in the libraries. If your data size is limited, consider using a conservative increment.

Passing Previously Closed FILE to stdio.h Functions

According to the standard, calling any **stdio.h** function on a previously closed `FILE` leads to undefined behavior. For example, the following code is invalid:

```
fclose(file);  
fputs(string, file);
```

The Green Hills **stdio.h** functions detect simple, single-threaded cases like the above and handle them in a bounded manner, even though the behavior is undefined by the standard. In the above example, `fputs` would return `EOF`, indicating failure. However, not all multi-threaded instances of this undefined behavior will be handled in such a bounded manner. If a `FILE` is used in one thread while another thread is simultaneously closing that `FILE`, this may cause unexpected results. The same caveat applies when one thread uses a closed `FILE` while another thread simultaneously opens a new file which then gets associated with the same `FILE`.

Customizing the Runtime-Constraint Handler for Annex K Functions

C11 adds a number of new bounds-checking functions, defined in Annex K. When called, these functions check certain runtime-constraints. If a constraint is violated, the current runtime-constraint handler is called.

Custom runtime-constraint handlers use the following type signature:

```
void custom_handler(const char *restrict msg, void *restrict ptr,
                    errno_t error);
```

The parameter *msg* contains a human-readable string describing the violated constraint. Currently, *ptr* is always `NULL` and *error* is always `EINVAL`, but they may contain more information in the future.

There are two methods available to set a runtime-constraint handler:

1. Set the handler at run-time by calling the function `set_constraint_handler_s()` (defined in **stdlib.h**). For example:

```
/* Annex K declarations must be explicitly requested. */
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>

void custom_handler(const char *restrict msg, void *restrict ptr,
                    errno_t error)
{
    abort();
}

int main(void)
{
    set_constraint_handler_s(&custom_handler);
    /* The rest of the program goes here. */
}
```

2. Customize the behavior of the default runtime-constraint handler. Normally, this function just calls `abort_handler_s()`, which prints a human-readable error message to `stderr` and then calls `abort()`. To customize the behavior of this function, define a function somewhere in your project with the following name and type signature:

```
void __ghs_default_constraint_handler_s(const char *msg, void *ptr,
                                       errno_t error);
```

On stand-alone targets, you can also modify the implementation in **src/libsys/ind_annex_k.c** and then rebuild **libsys.a**.

One reason to consider this second option is that almost every Annex K function links in the default handler. Customizing the default handler allows you to call those functions without linking in `abort_handler_s()` (and with it `fputs()` and `abort()`).

Green Hills Standard C Library Functions

The functions documented in the ISO C99 and later standards are contained in the libraries listed in “Libraries” on page 776. You can find lists of these functions, as well as thread-safety information, in the following sections:

- “Standard C Functions” on page 791
- “Wide Character Functions” on page 810
- “libmath.a Functions” on page 811
- “libind.a Functions” on page 823
- “libstartup.a Functions” on page 823

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 823.

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 786
E	Writes to the global variable <code>errno</code> . Achieves thread safety by per-thread storage of <code>errno</code> .

Code	Thread Safety
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

This section lists functions available in **libansi.a**, **libwchar_u16.a**, **libwchar_s32.a** and the low-level functions in **libsys.a** on which these standard functions depend. The information is laid out in two tables:

- “Standard C Functions and Dependencies” on page 791
- “libsys.a Function Implementation” on page 809

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.

Standard C Functions and Dependencies

This table lists functions in **libansi.a**, **libwchar_u16.a**, and **libwchar_s32.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 809 and “Low-Level System Library libsys.a” on page 829.

Function		Dependency	Thread Safety
void	<code>abort(void);</code>	<code>raise()</code>	Y
void	<code>abort_handler_s(const char *restrict msg, void *restrict ptr, errno_t error);</code>	<code>write()</code> <code>raise()</code>	IE
int	<code>abs(int x);</code>		K
void	<code>*aligned_alloc(size_t alignment, size_t size);</code>	<code>sbrk()</code>	P
char	<code>*asctime(const struct tm *timeptr);</code>	<code>__gh_timezone()</code>	T
char	<code>*asctime_r(const struct tm *timeptr, char *buffer);</code>	<code>__gh_timezone()</code>	Y
errno_t	<code>asctime_s(char *s, rsize_t maxsize, const struct tm *timeptr);</code>	<code>__gh_timezone()</code>	P
int	<code>_assert(const char *problem, const char *filename, int line);</code>	<code>raise()</code> <code>write()</code>	IE
void	<code>assert(int value);</code>	<code>raise()</code> <code>write()</code>	IE
int	<code>at_quick_exit(void (*func) (void));</code>		P
int	<code>atexit(void (*func) (void));</code>		P
int	<code>atoi(const char *nptr);</code>		E
long	<code>atol(const char *nptr);</code>		E
long long	<code>atoll(const char *nptr);</code>		E
int	<code>bcmp(const void *b1, const void *b2, size_t length);</code> This function is deprecated and will be removed in a future release. Use <code>memcmp</code> instead.		K
void	<code>bcopy(const void *from, void *to, size_t n);</code> This function is deprecated and will be removed in a future release. Use <code>memmove</code> instead. Note that the <code>from</code> and <code>to</code> parameters are swapped for <code>memmove</code> compared to <code>bcopy</code> .		K

Function	Dependency	Thread Safety
void <code>*bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));</code>		K
void <code>*bsearch_s(const void *key, const void *base, rsize_t nmemb, rsize_t size, int (*compar)(const void *, const void *, void *), void *context);</code>		P
wint_t <code>btowc(int c);</code>		Y
void <code>bzero(void *pt, size_t n);</code> This function is deprecated and will be removed in a future release. Use <code>memset</code> instead.		K
size_t <code>cl6rtomb(char *restrict s, char16_t c16, mbstate_t *restrict ps);</code>		E
size_t <code>c32rtomb(char *restrict s, char32_t c32, mbstate_t *restrict ps);</code>		E
void <code>*calloc(size_t nmemb, size_t size);</code>	<code>sbrk()</code>	P
void <code>cfree(void *item);</code>	<code>sbrk()</code>	P
void <code>clearerr(FILE *stream);</code>		I
clock_t <code>clock(void);</code>	<code>times()</code>	Y
char <code>*ctime(const time_t *timer);</code>	<code>__gh_timezone()</code> <code>localtime()</code>	T
char <code>*ctime_r(const time_t *timer, char *buffer);</code>	<code>__gh_timezone()</code> <code>localtime()</code>	Y
errno_t <code>ctime_s(char *s, rsize_t maxsize, const time_t *timer);</code>	<code>__gh_timezone()</code> <code>localtime_r()</code>	P
div_t <code>div(int numer, int denom);</code>		K
int <code>eprintf(const char *format, ...);</code> This function is deprecated and will be removed in a future release.	<code>write()</code>	IE
void <code>exit(int status);</code>	<code>_Exit()</code>	P
int <code>fclose(FILE *stream);</code>	<code>close()</code> <code>write()</code>	IE
FILE <code>*fdopen(int fno, const char *mode);</code>	<code>write()</code> <code>close()</code>	IE

Function		Dependency	Thread Safety
int	<code>feof(FILE *stream);</code>		Y
int	<code>ferror(FILE *stream);</code>		Y
int	<code>fflush(FILE *stream);</code>	<code>write()</code>	IE
int	<code>ffs(int i);</code>		K
int	<code>fgetc(FILE *stream);</code>	<code>read()</code>	IE
int	<code>fgetpos(FILE *restrict stream, fpos_t *restrict pos);</code>	<code>lseek()</code>	IE
char	<code>*fgets(char *restrict str, int n, FILE *restrict stream);</code>	<code>read()</code>	IE
wint_t	<code>fgetwc(FILE *stream);</code>	<code>read()</code>	IE
wint_t	<code>fgetwc_unlocked(FILE *stream);</code>	<code>read()</code>	NE
wchar_t	<code>*fgetws(wchar_t *restrict s, int n, FILE *restrict stream);</code>	<code>read()</code>	IE
int	<code>_filbuf(FILE *file);</code>	<code>read()</code>	IE
int	<code>fileno(FILE *stream)</code>		Y
int	<code>_flsbuf(int ch, FILE *file);</code>	<code>write()</code>	IE
FILE	<code>*fopen(const char *restrict filename, const char *restrict mode);</code>	<code>close()</code> <code>open()</code> <code>write()</code>	IE
errno_t	<code>fopen_s(FILE *restrict *restrict streamptr, const char *restrict filename, const char *restrict mode);</code>	<code>close()</code> <code>open()</code> <code>write()</code>	IE
int	<code>fprintf(FILE *restrict stream, const char *restrict format, ...);</code>	<code>write()</code>	IE
int	<code>fprintf_s(FILE *restrict stream, const char *restrict format, ...);</code>	<code>write()</code>	IE
int	<code>fputc(int c, FILE *stream);</code>	<code>write()</code>	IE
int	<code>fputs(const char *restrict s, FILE *restrict stream);</code>	<code>write()</code>	IE
wint_t	<code>fputwc(wchar_t c, FILE *stream);</code>	<code>write()</code>	IE
wint_t	<code>fputwc_unlocked(wchar_t c, FILE *stream);</code>	<code>write()</code>	NE
int	<code>fputws(const wchar_t *restrict s, FILE *restrict stream);</code>	<code>write()</code>	IE

Function		Dependency	Thread Safety
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
errno_t	freopen_s(FILE *restrict *restrict newstreamptr, const char *restrict filename, const char *restrict mode, FILE *restrict stream);	close() open() write()	IE
int	fscanf(FILE *restrict stream, const char *restrict format, ...);	close() read() sbrk() write()	IE
int	fscanf_s(FILE *restrict stream, const char *restrict format, ...);	close() read() sbrk() write()	IE
int	fseek(FILE *stream, long offset, int whence);	lseek() write()	IE
int	fseeko(FILE *stream, off_t offset, int whence); Under INTEGRITY, off_t is a 64-bit signed type. Elsewhere, it is long, so that fseeko() is the same as fseek().	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); Under INTEGRITY, off_t is a 64-bit signed type. Elsewhere, it is long, so that ftello() is the same as ftell().	lseek()	IE
int	fwide(FILE *stream, int mode);		N
int	fwprintf(FILE *restrict stream, const wchar_t *restrict format, ...);	write()	IE
int	fwprintf_s(FILE *restrict stream, const wchar_t *restrict format, ...);	write()	IE

Function		Dependency	Thread Safety
size_t	<code>fwrite(const void *restrict ptr, size_t size, size_t nmemb, FILE *stream);</code>	<code>write()</code>	IE
int	<code>fwscanf(FILE *restrict stream, const wchar_t *restrict format, ...);</code>	<code>close()</code> <code>read()</code> <code>sbrk()</code> <code>write()</code>	IE
int	<code>fwscanf_s(FILE *restrict stream, const wchar_t *restrict format, ...);</code>	<code>close()</code> <code>read()</code> <code>sbrk()</code> <code>write()</code>	IE
int	<code>getc(FILE *stream);</code>	<code>read()</code>	IE
int	<code>getc_unlocked(FILE *stream);</code>	<code>read()</code>	NE
int	<code>getchar(void);</code>	<code>read()</code>	IE
int	<code>getchar_unlocked(void);</code>	<code>read()</code>	NE
char	<code>*getenv(const char *name);</code>	<code>environ</code>	P
errno_t	<code>getenv_s(size_t *restrict len, char *restrict value, rsize_t maxsize, const char *restrict name);</code>	<code>environ</code>	P
char	<code>*gets(char *s);</code>	<code>read()</code>	IE
char	<code>*gets_s(char *s, rsize_t n);</code>	<code>read()</code>	IE
int	<code>getw(FILE *stream);</code> This function is deprecated and will be removed in a future release.	<code>read()</code>	IE
wint_t	<code>getwchar(void);</code>	<code>read()</code>	IE
struct tm	<code>*gmtime_s(const time_t *restrict timer, struct tm *restrict result);</code>	<code>gmtime_r()</code>	P
void	<code>ignore_handler_s(const char *restrict msg, void *restrict ptr, errno_t error);</code>		K
intmax_t	<code>imaxabs(intmax_t);</code>		Y
imaxdiv_t	<code>imaxdiv(intmax_t, intmax_t);</code>		Y
char	<code>*index(const char *str, int ch);</code>		K
int	<code>isalnum(int c);</code>		K
int	<code>isalpha(int c);</code>		K
int	<code>isblank(int c);</code>		K
int	<code>isctrl(int c);</code>		K

Function		Dependency	Thread Safety
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
struct lconv	*localeconv(void);		Y
struct tm	*localtime_s(const time_t *restrict timer, struct tm *restrict result);	localtime_r()	P

Function	Dependency	Thread Safety
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 mbrtoc16(char16_t *restrict pc16, const char *restrict s, size_t n, mbstate_t *restrict ps);		E
size_t mbrtoc32(char32_t *restrict pc32, 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
errno_t memcpy_s(void *restrict s1, rsize_t slmax, const void *restrict s2, rsize_t n);		P
errno_t memmove_s(void *s1, rsize_t slmax, const void *s2, rsize_t n);		P
void *memrchr(const void *s, int c, size_t n);		K
errno_t memset_s(void *s, rsize_t smax, int c, rsize_t n);		P
char *mktemp(char *str); This function is deprecated and will be removed in a future release.	getpid()	P

Function	Dependency	Thread Safety
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, ...);	write()	IE
int printf_s(const char *restrict format, ...);	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);		PE
int puts(const char *s);	write()	IE
int putw(int w, FILE *stream); This function is deprecated and will be removed in a future release.	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 nmem, size_t size, int (*compar)(const void *, const void *));		K
errno_t qsort_s(void *base, rsize_t nmem, rsize_t size, int (*compar)(const void *, const void *, void *), void *context);		P
void quick_exit(int status);	_Exit()	P
int rand(void);		P
int rand_r(unsigned int seedSent[1]);		K
int rand_r2(long seed[2]) This function is deprecated and will be removed in a future release.		K
void *realloc(void *ptr, size_t size);	sbrk()	P

Function		Dependency	Thread Safety
int	<code>remove(const char *filename);</code>	<code>unlink()</code>	Y
void	<code>rewind(FILE *stream);</code>	<code>lseek()</code> <code>write()</code>	IE
char	<code>*rindex(const char *str, int ch);</code>		K
int	<code>scanf(const char *restrict format, ...);</code>	<code>close()</code> <code>read()</code> <code>sbrk()</code> <code>write()</code>	IE
int	<code>scanf_s(const char *restrict format, ...);</code>	<code>close()</code> <code>read()</code> <code>sbrk()</code> <code>write()</code>	IE
See below	<code>set_constraint_handler_s(</code> <code>constraint_handler_t handler);</code> The return type of this function is <code>constraint_handler_t</code> .		P
void	<code>setbuf(FILE *restrict stream, char *restrict buf);</code>	<code>close()</code> <code>write()</code>	IE
int	<code>setenv(const char *envname, const char *envval, int overwrite);</code>		PE
int	<code>setlinebuf(FILE *stream);</code> This function is deprecated and will be removed in a future release.		N
char	<code>*setlocale(int category, const char *locale);</code>		Y
int	<code>setvbuf(FILE *restrict stream, char *restrict buf, int mode, size_t size);</code>	<code>close()</code> <code>write()</code>	IE
int	<code>snprintf(char *restrict s, size_t n, const char *restrict format, ...);</code>		K*
int	<code>snprintf_s(char *restrict s, rsize_t n, const char *restrict format, ...);</code>		P
int	<code>snwprintf_s(wchar_t *restrict s, rsize_t n, const wchar_t *restrict format, ...);</code>		P
int	<code>sprintf(char * restrict s, const char *restrict format, ...);</code>		K*
int	<code>sprintf_s(char *restrict s, rsize_t n, const char *restrict format, ...);</code>		P

Function		Dependency	Thread Safety
void	<code>srand(unsigned int seed);</code>		N
int	<code>sscanf(const char *restrict s, const char *restrict format, ...);</code>		K*
int	<code>sscanf_s(const char *restrict s, const char *restrict format, ...);</code>		P
int	<code>strcasecmp(const char *s1, const char *s2);</code> Ignore the difference between uppercase and lowercase		K
char	<code>*strcat(char *restrict s1, const char *restrict s2);</code>		K
errno_t	<code>strcat_s(char *restrict s1, rsize_t slmax, const char *restrict s2);</code>		P
char	<code>*strchr(const char *s, int c);</code>		K
int	<code>strcmp(const char *s1, const char *s2);</code>		K
errno_t	<code>strcpy_s(char *restrict s1, rsize_t slmax, const char *restrict s2);</code>		P
int	<code>strcoll(const char *s1, const char *s2);</code>		K
char	<code>*strcpy(char *restrict s1, const char *restrict s2);</code>		K
size_t	<code>strcspn(const char *s1, const char *s2);</code>		K
char	<code>*strdup(const char *s);</code>	<code>sbrk()</code>	P
char	<code>*strerror(int errnum);</code>		K
errno_t	<code>strerror_s(char *s, rsize_t maxsize, errno_t errnum);</code>		P
size_t	<code>strerrorlen_s(errno_t errnum);</code>		K
size_t	<code>strftime(char *restrict s, size_t maxsize, const char *restrict format, const struct tm *restrict timeptr);</code>	<code>__gh_timezone()</code>	Y
size_t	<code>strlen(const char *s);</code>		K

Function		Dependency	Thread Safety
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
errno_t	strncat_s(char *restrict s1, rsize_t slmax, const char *restrict s2, rsize_t n);		P
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
errno_t	strncpy_s(char *restrict s1, rsize_t slmax, const char *restrict s2, rsize_t n);		P
char	*strndup(const char *s, size_t n);	sbrk()	P
size_t	strnlen(const char *s, size_t n);		K
size_t	strnlen_s(const char *s, size_t maxsize);		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	strtoumax(const char * restrict nptr, char **restrict endptr int base);		E
char	*strtok(char *restrict s1, const char *restrict s2);		T
char	*strtok_s(char *restrict s1, rsize_t *restrict slmax, const char *restrict s2, char **restrict ptr);		P

Function		Dependency	Thread Safety
char	*strtok_r(char *s1, const char *s2, char **ppLast);		K
long long	strtoll(const char * restrict nptr, char **restrict endptr, int base);		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 nptr, char **restrict endptr, int base);		E
uintmax_t	strtoumax(const char * restrict nptr, char **restrict endptr, int base);		E
size_t	strxfrm(char *restrict s1, const char *restrict s2, size_t n);		K
void	swab(const char *from, char *to, ssize_t nbytes);		K
int	swprintf(wchar_t *restrict s, size_t n, const wchar_t *restrict format, ...);		Y
int	swprintf_s(wchar_t *restrict s, rsize_t n, const wchar_t *restrict format, ...);		P
int	swscanf(const wchar_t *restrict s, const wchar_t *restrict format, ...);		Y
int	swscanf_s(const wchar_t *restrict s, const wchar_t *restrict format, ...);		P
FILE	*tmpfile(void);	close() open() unlink() write()	IE
errno_t	tmpfile_s(FILE *restrict *restrict streamptr);	access() close() open() unlink() write()	IE
char	*tmpnam(char *s);	access()	TPE
errno_t	tmpnam_s(char *s, rsize_t maxsize);	access()	PE
char	*tmpnam_r(char *s);	access()	PE
int	tolower(int c);		K

Function	Dependency	Thread Safety
int toupper(int c);		K
wint_t towctrans(wint_t wc, wctrans_t desc);		Y
wint_t tolower(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 vfprintf_s(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 vfscanf_s(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 vfwprintf_s(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 vfwscanf_s(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 vprintf_s(const char *restrict format, va_list arg);	write()	IE

Function		Dependency	Thread Safety
int	<code>vscanf(const char *restrict format, va_list arg);</code>	<code>close()</code> <code>read()</code> <code>sbrk()</code> <code>write()</code>	IE
int	<code>vscanf_s(const char *restrict format, va_list arg);</code>	<code>close()</code> <code>read()</code> <code>sbrk()</code> <code>write()</code>	IE
int	<code>vsnprintf(char *restrict s, size_t n, const char *restrict format, va_list arg);</code>		K*
int	<code>vsnprintf_s(char *restrict s, rsize_t n, const char *restrict format, va_list arg);</code>		P
int	<code>vsnwprintf_s(wchar_t *restrict s, rsize_t n, const wchar_t *restrict format, va_list arg);</code>		P
int	<code>vsprintf(char *restrict s, const char *restrict format, va_list arg);</code>		K*
int	<code>vsprintf_s(char *restrict s, rsize_t n, const char *restrict format, va_list arg);</code>		P
int	<code>vsscanf(const char *restrict s, const char *restrict format, va_list arg);</code>		K*
int	<code>vsscanf_s(const char *restrict s, const char *restrict format, va_list arg);</code>		P
int	<code>vswprintf(wchar_t *restrict s, size_t, const wchar_t *restrict format, va_list arg);</code>		Y
int	<code>vswprintf_s(wchar_t *restrict s, rsize_t n, const wchar_t *restrict format, va_list arg);</code>		P
int	<code>vswscanf(const wchar_t *restrict s, const wchar_t *restrict format, va_list arg);</code>		Y
int	<code>vswscanf_s(const wchar_t *restrict s, const wchar_t *restrict format, va_list arg);</code>		P
int	<code>vwprintf(const wchar_t *restrict format, va_list arg);</code>	<code>write()</code>	IE

Function		Dependency	Thread Safety
int	<code>vwprintf_s(const wchar_t *restrict format, va_list arg);</code>	<code>write()</code>	IE
int	<code>vscanf(const wchar_t *restrict format, va_list arg);</code>	<code>close()</code> <code>read()</code> <code>sbrk()</code> <code>write()</code>	IE
int	<code>vscanf_s(const wchar_t *restrict format, va_list arg);</code>	<code>close()</code> <code>read()</code> <code>sbrk()</code> <code>write()</code>	IE
size_t	<code>wcrtomb(char *restrict s, wchar_t wc, mbstate_t *restrict ps);</code>		Y
wchar_t	<code>*wcscat(wchar_t *restrict s1, const wchar_t *restrict s2);</code>		Y
errno_t	<code>wcscat_s(wchar_t *restrict s1, rsize_t slmax, const wchar_t *restrict s2);</code>		P
wchar_t	<code>*wcschr(const wchar_t *s, wchar_t wc);</code>		Y
int	<code>wscmp(const wchar_t *s1, const wchar_t *s2);</code>		Y
int	<code>wscoll(const wchar_t *s1, const wchar_t *s2);</code>		Y
wchar_t	<code>*wcscpy(wchar_t *restrict s1, const wchar_t *restrict s2);</code>		Y
errno_t	<code>wcscpy_s(wchar_t *restrict s1, rsize_t slmax, const wchar_t *restrict s2);</code>		P
size_t	<code>wscspn(const wchar_t *s1, const wchar_t *s2);</code>		Y
size_t	<code>wcsftime(wchar_t *restrict s, size_t maxsize, const wchar_t *restrict format, const struct tm *restrict timeptr);</code>	<code>__gh_timezone()</code>	Y
size_t	<code>wcslen(const wchar_t *s);</code>		Y
wchar_t	<code>*wcsncat(wchar_t *restrict s1, const wchar_t *restrict s2, size_t n);</code>		Y
errno_t	<code>wcsncat_s(wchar_t *restrict s1, rsize_t slmax, const wchar_t *restrict s2, rsize_t n);</code>		P

Function		Dependency	Thread Safety
int	<code>wcsncmp(const wchar_t *restrict s1, const wchar_t *restrict s2, size_t n);</code>		Y
wchar_t	<code>*wcsncpy(wchar_t *restrict s1, const wchar_t *restrict s2, size_t n);</code>		Y
errno_t	<code>wcsncpy_s(wchar_t *restrict s1, rsize_t slmax, const wchar_t *restrict s2, rsize_t n);</code>		P
size_t	<code>wcsnlen(const wchar_t *wstr, size_t n);</code>		Y
size_t	<code>wcsnlen_s(const wchar_t *s, size_t maxsize);</code>		Y
wchar_t	<code>*wcpbrk(const wchar_t *s1, const wchar_t *s2);</code>		Y
wchar_t	<code>*wcsrchr(const wchar_t *s, wchar_t wc);</code>		Y
size_t	<code>wcsrtombs(char *restrict dst, const wchar_t **restrict src, size_t len, mbstate_t *restrict ps);</code>		Y
size_t	<code>wcsspn(const wchar_t *s1, const wchar_t *s2);</code>		Y
wchar_t	<code>*wcsstr(const wchar_t *s1, const wchar_t *s2);</code>		Y
intmax_t	<code>wcstoimax(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);</code>		Y
wchar_t	<code>*wcstok(wchar_t *restrict s1, const wchar_t *restrict s2, wchar_t **restrict ptr);</code>		Y
wchar_t	<code>*wcstok_s(wchar_t *restrict s1, rsize_t *restrict slmax, const wchar_t *restrict s2, wchar_t **restrict ptr);</code>		P
long	<code>wcstol(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);</code>		Y

Function		Dependency	Thread Safety
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
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
errno_t	wmemcpy_s(wchar_t *restrict s1, rsize_t slmax, const wchar_t *restrict s2, rsize_t n);		P
wchar_t	*wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);		Y
errno_t	wmemmove_s(wchar_t *s1, rsize_t slmax, const wchar_t *s2, rsize_t n);		P
wchar_t	*wmemset(wchar_t *s, wchar_t c, size_t n);		Y
int	wprintf(const wchar_t *restrict format, ...);	write()	IE

Function		Dependency	Thread Safety
int	wprintf_s(const wchar_t *restrict format, ...);	write()	IE
int	wscanf(const wchar_t *restrict format, ...);	close() read() sbrk() write()	IE
int	wscanf_s(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.

**Note**

Excepting `getenv_s()`, Annex K functions marked as "P" only require a lock to guard accesses to the current runtime-constraint handler pointer. This pointer is shared between threads, in accordance with the C standard. If the current handler is never changed after threads are spawned, then these functions are thread-safe. (Note that Annex K functions are indicated by the suffix "_s")

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
<code>access()</code>	ind_stat.c	MULTI system call
<code>close()</code>	ind_io.c	MULTI system call

Function	Module	Implementation
<code>creat()</code>	ind_io.c	MULTI system call
<code>_Exit()</code>	ind_exit.c	MULTI system call
<code>getpid()</code>	ind_gpid.c	Returns 17
<code>__gh_timezone()</code>	ind_tmzn.c	8 hours West of UTC/GMT
<code>localtime()</code>	ind_loctm.c	US daylight saving rules
<code>localtime_r()</code>	ind_loctm.c	US daylight saving rules
<code>lseek()</code>	ind_io.c	MULTI system call
<code>open()</code>	ind_io.c	MULTI system call
<code>raise()</code>	ind_sgnl.c	ISO C99 compliant
<code>read()</code>	ind_io.c	MULTI system call
<code>sbrk()</code>	ind_heap.c	Allocates from <code>.heap</code>
<code>time()</code>	ind_time.c	MULTI system call
<code>unlink()</code>	ind_io.c	MULTI system call
<code>write()</code>	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 210 and “Kanji Character Support” on page 610). 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
<code>int</code>	<code>mbtowc(wchar_t * restrict pwc, const char *restrict s, size_t n);</code>	Y
<code>int</code>	<code>wctomb(char *s, wchar_t wchar);</code>	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	<code>__fpclassifyd(double x);</code>	Y
int	<code>__fpclassifyf(float x);</code>	Y
int	<code>__fpclassifyl(long double x);</code>	Y
int	<code>__isfinitd(double x);</code>	Y
int	<code>__isfinitef(float x);</code>	Y
int	<code>__isfinitel(long double x);</code>	Y
int	<code>__isinfld(double x);</code>	Y
int	<code>__isinff(float x);</code>	Y
int	<code>__isinfl(long double x);</code>	Y
int	<code>__isnand(double x);</code>	Y
int	<code>__isnanf(float x);</code>	Y
int	<code>__isnandl(long double x);</code>	Y
int	<code>__isnormald(double x);</code>	Y
int	<code>__isnormalf(float x);</code>	Y
int	<code>__isnormall(long double x);</code>	Y
int	<code>__signbitd(double x);</code>	Y
int	<code>__signbitf(float x);</code>	Y
int	<code>__signbitl(long double x);</code>	Y
double	<code>acos(double x);</code>	E
float	<code>acosf(float x);</code>	E
long double	<code>acosl(long double x);</code>	E
double	<code>acosh(double x);</code>	E
float	<code>acoshf(float x);</code>	E

Function		Thread safety
long double	<code>acoshl(long double x);</code>	E
double	<code>asin(double x);</code>	E
float	<code>asinf(float x);</code>	E
long double	<code>asinl(long double x);</code>	E
double	<code>asinh(double x);</code>	Y
float	<code>asinhf(float x);</code>	Y
long double	<code>asinhf(long double x);</code>	Y
double	<code>atan(double x);</code>	Y
float	<code>atanf(float x);</code>	Y
long double	<code>atanl(long double x);</code>	Y
double	<code>atan2(double y, double x);</code>	Y
float	<code>atan2f(float y, float x);</code>	Y
long double	<code>atan2l(long double y, long double x);</code>	Y
double	<code>atanh(double x);</code>	E
float	<code>atanhf(float x);</code>	E
long double	<code>atanhl(long double x);</code>	E
double	<code>cbrt(double x);</code>	Y
float	<code>cbrtf(float x);</code>	Y
long double	<code>cbrtl(long double x);</code>	Y
double	<code>ceil(double x);</code>	Y
float	<code>ceilf(float x);</code>	Y
long double	<code>ceilf(long double x);</code>	Y
double	<code>copysign(double x, double y);</code>	Y
float	<code>copysignf(float x, float y);</code>	Y
long double	<code>copysignl(long double x, long double y);</code>	Y
double	<code>cos(double x);</code>	Y
float	<code>cosf(float x);</code>	Y
long double	<code>cosl(long double x);</code>	Y
double	<code>cosh(double x);</code>	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	erfc1(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	fabs1(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	<code>fmax(double x, double y);</code>	Y
float	<code>fmaxf(float x, float y);</code>	Y
long double	<code>fmaxl(long double x, long double y);</code>	Y
double	<code>fmin(double x, double y);</code>	Y
float	<code>fminf(float x, float y);</code>	Y
long double	<code>fminl(long double x, long double y);</code>	Y
double	<code>fmod(double x, double y);</code>	E
float	<code>fmodf(float x, float y);</code>	E
long double	<code>fmodl(long double x, long double y);</code>	E
double	<code>frexp(double value, int *exp);</code>	Y
float	<code>frexpf(float value, int *exp);</code>	Y
long double	<code>frexpl(long double value, int *exp);</code>	Y
double	<code>gamma(double x);</code>	E
double	<code>hypot(double x, double y);</code>	E
float	<code>hypotf(float x, float y);</code>	E
long double	<code>hypotl(long double x, long double y);</code>	E
int	<code>ilogb(double x);</code>	E
int	<code>ilogbf(float x);</code>	E
int	<code>ilogbl(long double x);</code>	E
int	<code>isfinite(double x);</code>	Y
int	<code>isinf(double x);</code>	Y
int	<code>isnan(double x);</code>	Y
int	<code>isnormal(double x);</code>	Y
double	<code>j0(double x);</code>	Y
double	<code>j1(double x);</code>	Y
double	<code>jn(int n, double x);</code>	Y
double	<code>ldexp(double x, int exp);</code>	E
float	<code>ldexpf(float x, int exp);</code>	E
long double	<code>ldexpl(long double x, int exp);</code>	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
long	lrint(double x);	E
long	lrintf(float x);	E
long	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	tanh1(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	<code>truncl(long double x);</code>	Y
double	<code>y0(double x);</code>	E
double	<code>y1(double x);</code>	E
double	<code>yn(int n, double x);</code>	E

Functions from `fenv.h` in `libmath.a`

Function		Thread safety
int	<code>feclearexcept(int excepts);</code>	N
int	<code>fegetenv(fenv_t *envp);</code>	N
int	<code>fegetexceptflag(fexcept_t *flagp, int excepts);</code>	N
int	<code>fegetround(void);</code>	N
int	<code>feholdexcept(fenv_t *envp);</code>	N
int	<code>feraiseexcept(int excepts);</code>	N
int	<code>fesetenv(const fenv_t *envp);</code>	N
int	<code>fesetexceptflag(const fexcept_t *flagp, int excepts);</code>	N
int	<code>fesetround(int round);</code>	N
int	<code>fetestexcept(int excepts);</code>	N
int	<code>feupdateenv(const fenv_t *envp);</code>	N

Functions from `complex.h` in `libmath.a`

Function		Thread safety
double	<code>cabs(double complex z);</code>	E
float	<code>cabsf(float complex z);</code>	E
long double	<code>cabsl(long double complex z);</code>	E
double complex	<code>cacos(double complex z);</code>	E

Function		Thread safety
float complex	<code>cacosf(float complex z);</code>	E
long double complex	<code>cacosl(long double complex z);</code>	E
double complex	<code>cacosh(double complex z);</code>	E
float complex	<code>cacoshf(float complex z);</code>	E
long double complex	<code>cacoshl(long double complex z);</code>	E
double complex	<code>casin(double complex z);</code>	E
float complex	<code>casinf(float complex z);</code>	E
long double complex	<code>casinl(long double complex z);</code>	E
double complex	<code>casinh(double complex z);</code>	E
float complex	<code>casinhf(float complex z);</code>	E
long double complex	<code>casinhl(long double complex z);</code>	E
double	<code>carg(double complex z);</code>	E
float	<code>cargf(float complex z);</code>	E
long double	<code>cargl(long double complex z);</code>	E
double complex	<code>catan(double complex z);</code>	E
float complex	<code>catanf(float complex z);</code>	E
long double complex	<code>catanl(long double complex z);</code>	E
double complex	<code>catanh(double complex z);</code>	E
float complex	<code>catanhf(float complex z);</code>	E
long double complex	<code>catanh1(long double complex z);</code>	E
double complex	<code>ccos(double complex z);</code>	E

Function		Thread safety
float complex	<code>ccosf(float complex z);</code>	E
long double complex	<code>ccosl(long double complex z);</code>	E
double complex	<code>ccosh(double complex z);</code>	E
float complex	<code>ccoshf(float complex z);</code>	E
long double complex	<code>ccoshl(long double complex z);</code>	E
double complex	<code>cexp(double complex z);</code>	E
float complex	<code>cexpf(float complex z);</code>	E
long double complex	<code>cexpl(long double complex z);</code>	E
double	<code>cimag(double complex z);</code>	E
float	<code>cimagf(float complex z);</code>	E
long double	<code>cimagl(long double complex z);</code>	E
double complex	<code>clog(double complex z);</code>	E
float complex	<code>clogf(float complex z);</code>	E
long double complex	<code>clogl(long double complex z);</code>	E
double complex	<code>conj(double complex z);</code>	E
float complex	<code>conjf(float complex z);</code>	E
long double complex	<code>conjl(long double complex z);</code>	E
double complex	<code>cpow(double complex x, double complex y);</code>	E
float complex	<code>cpowf(float complex x, float complex y);</code>	E
long double complex	<code>cpowl(long double complex x, long double complex y);</code>	E
double complex	<code>cproj(double complex z);</code>	E

Function		Thread safety
float complex	<code>cprojf(float complex z);</code>	E
long double complex	<code>cprojl(long double complex z);</code>	E
double	<code>creal(double complex z);</code>	E
float	<code>crealf(float complex z);</code>	E
long double	<code>creall(long double complex z);</code>	E
double complex	<code>csin(double complex z);</code>	E
float complex	<code>csinf(float complex z);</code>	E
long double complex	<code>csinl(long double complex z);</code>	E
double complex	<code>csinh(double complex z);</code>	E
float complex	<code>csinhf(float complex z);</code>	E
long double complex	<code>csinhl(long double complex z);</code>	E
double complex	<code>csqrt(double complex z);</code>	E
float complex	<code>csqrtf(float complex z);</code>	E
long double complex	<code>csqrtl(long double complex z);</code>	E
double complex	<code>ctan(double complex z);</code>	E
float complex	<code>ctanf(float complex z);</code>	E
long double complex	<code>ctanl(long double complex z);</code>	E
double complex	<code>ctanh(double complex z);</code>	E
float complex	<code>ctanhf(float complex z);</code>	E
long double complex	<code>ctanhl(long double complex z);</code>	E

Functions from stdlib.h in libmath.a

Function		Thread safety
double	<code>atof(const char *nptr);</code>	E
double	<code>strtod(const char *restrict nptr, char **restrict endptr);</code>	E
float	<code>strtof(const char *restrict nptr, char **restrict endptr);</code>	E
long double	<code>strtold(const char *restrict nptr, char **restrict endptr);</code>	E

Functions from wchar.h in libmath.a

Function		Thread safety
double	<code>wctod(const wchar_t *restrict nptr, wchar_t **restrict endptr);</code>	E
long double	<code>wctold(const wchar_t *restrict nptr, wchar_t **restrict endptr);</code>	E
float	<code>wcstof(const wchar_t *restrict nptr, wchar_t **restrict endptr);</code>	

Additional information about wide character functions is provided in “Wide Character Functions” on page 810.

Functions from setjmp.h in libmath.a

Function		Thread safety
void	<code>longjmp(jmp_buf env, int val);</code>	Y
int	<code>setjmp(jmp_buf env);</code>	Y

Functions from time.h in libmath.a

Function		Thread safety
double	<code>difftime(time_t time1, time_t time0);</code>	Y

libind.a Functions

Function		Thread safety
void	<code>_rterr(int num, int linenum, char *str, void *ptr, void **trace, int depth, size_t len);</code>	Y
void	<code>_rterr2(int num, int linenum, char *str);</code>	Y
void	<code>*memmove(void *s1, const void *s2, size_t n);</code>	K



Note

The functions `_rterr()` and `_rterr2()` are used in run-time error checking. For more information about run-time error checking, see “Run-Time Error Checks” on page 135.

libstartup.a Functions

The following table lists the standard C library functions in **libstartup.a**.

Function		Thread safety
void	<code>*memcpy(void * restrict d, const void * restrict s, size_t n);</code>	YK
void	<code>*memset(void *s, int c, size_t n);</code>	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_cnd_timedwait`
 - `__ghs_time64_difftime`
 - `__ghs_time64_gmtime`
 - `__ghs_time64_gmtime_r`
 - `__ghs_time64_localtime`
 - `__ghs_time64_localtime_r`
 - `__ghs_time64_mktime`
 - `__ghs_time64_mtx_timedlock`
 - `__ghs_time64_time`
 - `__ghs_time64_times`
 - `__ghs_time64_timespec_get`
 - `__ghs_time64_thrd_sleep`
- **Safe variants of memory functions (-mem_funcs=safe):**
 - `__ghs_variant1_memfuncs_safe_memcpy`
 - `__ghs_variant1_memfuncs_safe_memset`

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 69).

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.a64** — see “Startup Module crt0.o” on page 827.
 - **libstartup.gpj** — see “Low-Level Startup Library libstartup.a” on page 828.
- **libsys/libsys.gpj** — see “Low-Level System Library libsys.a” on page 829.
- **libboardinit/libboardinit.gpj** — see “Board Initialization Library libboardinit.a” on page 834. 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 825.
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/startup.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
- 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.a64**. 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.a64

crt0.a64 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.a64** assumes that the debug server has already performed some register initialization (such as the stack pointer).

If the system call is not successful, **crt0.a64** 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.a64**

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 455.
- 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. 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 449.

4. 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 830.

ind_mcpy.c, ind_mset.c

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.c

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 ARM64.

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 809.

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.a64, ind_dots.a64

These modules implement low-level system call capability. The routines `__ghs_syscall()` and `__ghs_syscall64()` are called by various system call routines, such as `open()`, `close()`, `read()`, and `write()`. The `__ghs_syscall()` routines transfer 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_condvar.c

This module contains stubs for implementing an interface for the multi-threaded aspects of the `<condition_variable>` header used in C11 or later and C++11 or later.

ind_cov.c

This module contains the initialization function needed for instrumented coverage (see “Enabling Instrumented Coverage or Performance Profiling” on page 47).

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_except.c

This module provides routines that allow C++ exceptions to be used in multiple threads. This module uses the per-thread static library data routines in **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 49).

ind_stackcanary.c, ind_stackcanary2.c

These modules provide support for protection against stack smashing attacks via use of stack canaries (see “Check for Stack Smashing Attacks” on page 94).

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_thrd11.c

This module provides stubs that can be modified to enable the thread, thread-local storage, and synchronization primitives of C11 or later and C++11 or later, as enabled by `<thread>` and `<mutex>`.

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-like implementation, including:

```
int open(const char *filename, int flags, ...);
int creat(const char *filename, int prot);
int close(int fno);
long read(int fno, void *buf, long size);
long write(int fno, const void *buf, long size);
int unlink(const 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 routines `__ghs_syscall()` and `__ghs_syscall64()`, described in “ind_call.a64, ind_dots.a64” on page 830.

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
<code>__ghs_at_exit(struct __GHS_AT_EXIT*)</code>	Registers functions that will be called by <code>_Exit()</code> (similar to the C library <code>atexit()</code>).
<code>void _Exit(int)</code>	Calls clean-up routines registered by the program via <code>__ghs_at_exit()</code> and allows the program to terminate cleanly via the <code>__ghs_syscall(SYSCALL_EXIT, code)</code> system call. Clean-up routines registered by <code>atexit()</code> and C++ global destructors are handled separately in <code>exit()</code>, which finishes by calling <code>_Exit()</code>.

Other Low-Level Functions

The following files contain a basic set of Linux-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	<code>struct tm *gmtime(const time_t *timer);</code>
ind_loctm.c	<code>struct tm *localtime(const time_t *timer);</code>
ind_renm.c	<code>int rename(const char *old, const char *new);</code>
ind_sgnl.c	<code>int raise(int sig);</code> <code>void (*signal(int sig, void (*func)(int)))(int);</code>
ind_stat.c	<code>int access(const char *name, int mode);</code>
ind_syst.c	<code>int system(const char *string);</code>
ind_time.c	<code>time_t time(time_t *tptr);</code> <code>int times (struct tms *buffer);</code>
ind_tmzn.c	<code>void tzset(void);</code> <code>int __gh_timezone(void);</code>
ind_trnc.c	<code>int truncate(const char* path, long length);</code> 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 825), 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 49.

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
<code>atexit()</code>	Processing in the Green Hills <code>_exit()</code> function.
C++ destructors	The Green Hills <code>exit()</code> function.
Multithreading	Might require <code>__gh_lock_init()</code> to be called.

Feature	Startup Code Requirement
PIC and PID	Might require base registers to be set up.
<code>stdin</code> , <code>stdout</code> , and <code>stderr</code>	<code>__gh_iob_init()</code> must be called.

The following options also depend on Green Hills startup code:

Option	Startup Code Requirement
-globalreg	Certain variables (such as those declared with <code>register int glob</code>) 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 16

Mixing Languages and Writing Portable Code

Contents

Mixing Languages	838
Writing Portable Code	843

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 841.

```
int main(void);
void _main(void);

int 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 842). 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 a call 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 838.

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.

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.

Using Assembly Code with INTEGRITY TimeMachine Enabled

If you are employing the separately licensed INTEGRITY TimeMachine feature, there are additional considerations when using assembly code.

Assembly Code Requirements

- Functions must be marked by `.type/.size` directives to allow the assembler to differentiate between code and data.
- Expressions containing the difference of labels defined in the text section are not allowed. (The distance between labels will grow when INTEGRITY TimeMachine instrumentation is added and will shrink again if the instrumentation is stripped.)
- Stores to `volatile __sideeffect` memory may not be used.
- Atomic memory access instructions may not be used.
- Functions may only be entered from the top. As such, you cannot branch into the middle of a function.
- Code in one function may not flow-through to another function.
- Code may not use registers reserved for INTEGRITY TimeMachine.
- Indirect branches may only branch to labels. All such labels must be contained in the current function.

How the Assembler Instruments Code for INTEGRITY TimeMachine

- Function-entry packets are added to the beginning of functions.
- Function-exit packets are added before `return` or tail call instructions.
- Block-entry packets are added at the beginning of functions and when entering basic blocks. Basic blocks are assumed to begin after branches and whenever a label inside a function is encountered.
- Appropriate pre-image instructions are added before a store. Stores are assumed to be either `non-volatile` or `volatile __nosideeffect` (see “Using Volatiles with INTEGRITY TimeMachine Enabled” on page 599).



Note

Assembly code embedded within C/C++ modules should be marked with the `#pragma directives` `ghs safeasm` or `ghs optasm` (see “Green Hills Extension Pragma Directives” on page 752). Otherwise, the linker may replace the INTEGRITY TimeMachine instrumentation with a series of NOPs (see “Function Entry/Exit Logging” on page 133).

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 or later, 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 325.

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 354). In particular, when the compiler reads a memory location, or writes to a memory location, it assumes that the value remains several instructions later. To avoid the (potentially high) speed penalties involved in re-reading memory, the compiler may reuse the value previously read from memory 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, interrupt-driven routines, and when Linux-style signals are enabled.

An example of the potential problems involved with memory optimizations is that many Linux 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.

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 119.

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 determines 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. Note that with interprocedural optimizations and inlining, the largest function may be a function with other functions inlined into it. Because the inliner will refrain from increasing the size and complexity of a function beyond a certain ratio, attending to a function's original size will still be helpful.

Chapter 17

Enhanced asm Macro Facility for C and C++

Contents

Definition of Terms	859
Using and Defining the asm Macro Facility	859
Using asm Macros	861
ARM64 asm Macros	865
Guidelines for Writing asm Macros	866

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 401 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
<code>asm</code> macro	The mechanism by which programs use the enhanced <code>asm</code> facility. An <code>asm</code> 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 <code>asm</code> 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 <code>asm</code> macro. When the modes in the pattern all match those of the use, the corresponding body is expanded.
<code>asm</code> 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 ARM64, see “ARM64 `asm` Macros” on page 865.

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
<code>con</code>	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 <i>identifier(s)</i> that are specified as being of mode <code>lab</code> do not appear as formal parameters in the <code>asm</code> macro definition, unlike the preceding modes. Such identifiers will be expanded to be unique. Example: <pre>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); }</pre>
mem	Matches any allowed machine addressing mode, with the exception of <code>reg</code> and <code>con</code> .
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 <code>treg</code> or <code>ureg</code> .
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 (`%`).

ARM64 asm Macros

The following example demonstrates how to use `asm` macros correctly:

```
__asm int addword10(arg1)
{
%con arg1 %lab lab1 %lab lab2
    ldr        x0, =(arg1 + 10)
%reg arg1
    mov        x0, arg1
    add        x0, x0, #10
%nearmem arg1
    ldr        x0, arg1
    add        x0, x0, #10
%farsprel arg1 %lab lab1 %lab lab2
    ldr        x0, = %SPOFF(arg1)
    ldr        x0, [sp, x0]
    add        x0, x0, #10
%mem arg1 %lab lab1 %lab lab2
    ldr        x0, =arg1
    ldr        x0, [x0]
    add        x0, x0, #10
%error
}
```

```
short x = 30;
short y = 2;
int z = 10;

int func1(void)
{
    int tmp2 = y;
    return addword10(10) + addword10(tmp2) + addword10(0x10000) + addword10(z);
}

int func2(void)
{
    int tmp2 = y;
    return (10+10) + (tmp2+10) + (0x10000+10) + (z+10);
}
```

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 18

GNU Extended Assembly

Contents

Extended GNU Support	870

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.

The compiler supports most common use cases, but not all use cases and constraints allowed by GNU are supported. We recommend the use of intrinsic functions over the use of GNU extended `asm` statements. Note also that the compiler may optimize `asm` statements away unless you use the `volatile` keyword.

Extended GNU Support

The list of target-independent format strings supported by the compiler is:

- `"%%"`: Outputs a single `'%'` into the assembly.
- `"%="`: Outputs a number that is unique to this instantiation of the `asm` statement, which is useful when creating and referring to local labels.

The list of target-independent constraints supported by the compiler is:

- `'r'`: Any integer register.
- `'f'`: Any float register.
- `'m'`: Any memory location.
- `'i'`: Any immediate integer (including symbolic references).
- `'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.

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.

Within assembly text, operands are referred to by number (for example, `%0` and `%1`). The following modifiers may be added before the number to change the output format of the operand:

- `'w'`: The register should be output as a 32-bit GPR (`w0` to `w30`).
- `'x'`: The register should be output as a 64-bit GPR (`x0` to `x30`).

Chapter 19

The ELF File Format

Contents

Formats of Relocatable and Executable Files	874
32-Bit ELF Data Types	875
ELF Headers	876
ELF Identification	879
ELF Sections	881
Symbol Tables	889
String Tables	892
Program Headers	893

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:

Relocatable file	Executable file
ELF Header	ELF Header
Program Header Table (<i>optional</i>)	Program Header Table
Section 1	Segment 1
...	
Section <i>n</i>	Segment 2
...	
...	...
Section Header Table	Section Header Table (<i>optional</i>)

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 879.

`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
- 183 (`EM_AARCH64`): ARM64
- 243 (`EM_RISCV`): RISC-V

`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.

<code>e_entry</code>	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.
<code>e_phoff</code>	Specifies the program header table's file offset in bytes. If the file has no program header table, this field contains the value zero.
<code>e_shoff</code>	Specifies the section header table's file offset in bytes. If the file has no section header table, this field contains the value zero.
<code>e_flags</code>	Specifies processor-specific flags associated with the file. Flag names take the form <code>EF_machine_flag</code> .
<code>e_ehsize</code>	Specifies the ELF header size in bytes.
<code>e_phentsize</code>	Specifies the size in bytes of one entry in the file's program header table; all entries are the same size.
<code>e_phnum</code>	Specifies the number of entries in the program header table. The product of <code>e_phentsize</code> and <code>e_phnum</code> specifies the table's size in bytes. If a file has no program header table, <code>e_phnum</code> contains the value zero.
<code>e_shentsize</code>	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.
<code>e_shnum</code>	Specifies the number of entries in the section header table. The product of <code>e_shentsize</code> and <code>e_shnum</code> specifies the section header table's size in bytes. If a file has no section header table, this field contains the value zero.
<code>e_shstrndx</code>	Specifies the index of the section containing the string table used for section names. By convention, this is the section <code>.shstrtab</code> . If the file has no section name string table, this field contains the value <code>SHN_UNDEF</code> .

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 <code>e_ident[]</code>

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.

<code>sh_name</code>	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.
<code>sh_type</code>	Specifies the section's contents and semantics. For more information, see “Section Attribute Flags” on page 884.
<code>sh_flags</code>	Specifies 1-bit flags that describe miscellaneous attributes of the section.
<code>sh_addr</code>	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.
<code>sh_offset</code>	Specifies the byte offset from the beginning of the file to the first byte in the section.
<code>sh_size</code>	Specifies the section's size in bytes. Unless the section type is <code>SHT_NOBITS</code> , the section occupies <code>sh_size</code> bytes in the file. A section of type <code>SHT_NOBITS</code> may have a non-zero size, but it occupies no space in the file.
<code>sh_link</code>	Contains a section header table index link, whose interpretation depends on the section type.
<code>sh_info</code>	Contains extra information, whose interpretation depends on the section type.
<code>sh_addralign</code>	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 <code>sh_addr</code> must be equal to 0 (zero), modulo the value of <code>sh_addralign</code> . 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.
<code>sh_entsize</code>	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 889.

Name	Value	Meaning
<code>SHN_UNDEF</code>	0	A meaningless section. A symbol “defined” relative to this section is an undefined symbol.
<code>SHN_COMMON</code>	-14	Common, or <code>.bss</code> , symbols are allocated space in this section.
<code>SHN_ABS</code>	-15	Contents of the section are absolute values; they are not affected by relocation.
<code>SHN_GHS_SMALLCOMMON</code>	-256	Not available for all processors. Similar to <code>SHN_COMMON</code> , but for a limited number of small variables allocated to SDA.
<code>SHN_GHS_ZERO_COMMON</code>	-255	Not available for all processors. Similar to <code>SHN_COMMON</code> , but for a limited number of small variables allocated to ZDA.
<code>SHN_GHS_TINYCOMMON</code>	-254	Not available for all processors. Similar to <code>SHN_COMMON</code> , 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
<code>SHT_NULL</code>	0	Marks the section header as inactive. It does not have an associated section. Other members of the section header have undefined values.
<code>SHT_PROGBITS</code>	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 <code>Elf32_Rela</code> 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 <code>SHT_PROGBITS</code> .
SHT_REL	9	Contains relocation entries without explicit addends, such as type <code>Elf32_Rel</code> 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).
<i>other</i>	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

<code>.relname</code> <code>.relaname</code> These sections contain relocation information. Conventionally, <i>name</i> is supplied by the section to which the relocations apply. A relocation section for <code>.text</code> has the name <code>.rel.text</code> or <code>.rela.text</code> . Types: <code>SHT_REL</code> and <code>SHT_RELA</code> Attributes: None
<code>.rodata</code> Read-only data area. Similar to the <code>.data</code> section, but composed of constant data. Type: <code>SHT_PROGBITS</code> Attributes: <code>SHF_ALLOC</code>
<code>.shstrtab</code> Contains section names. Type: <code>SHT_STRTAB</code> Attributes: None
<code>.strtab</code> Contains strings, most commonly the strings that represent the names associated with symbol table entries. Type: <code>SHT_STRTAB</code> Attributes: None
<code>.symtab</code> Contains a symbol table. Type: <code>SHT_SYMTAB</code> Attributes: None
<code>.text</code> Contains the “text”, or executable instructions, of a program. Type: <code>SHT_PROGBITS</code> Attributes: <code>SHF_ALLOC</code> <code>SHF_EXECINSTR</code>

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.

<code>.fixaddr</code> <code>.fixtype</code> 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
<code>.heap</code> Section describing the area of memory for dynamic allocations through <code>malloc</code> and related functions. Type: SHT_NOBITS Attributes: SHF_ALLOC SHF_WRITE
<code>.linfix</code> 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 <code>.linfix</code> section. MULTI merges the information in the <code>.dla</code> file and the <code>.linfix</code> section when debugging the program. The <code>.linfix</code> section is not downloaded to the target. Type: SHT_PROGBITS Attributes: None
<code>.rominfo</code> A linker-created table describing the type of each section (ROM, CROM, ROM_NOCOPY, or UNCOMPRESSED) that gets created as a result of a ROM, CROM, or ROM_NOCOPY attribute, as well as its association with the equivalent RAM section. The <code>.rominfo</code> section is not downloaded to the target. Type: SHT_PROGBITS Attributes: None
<code>.sdbase</code> If not in PID mode, the run-time system initializes the Small Data Area (SDA) base register to be the address of this section (<code>.sdbase</code>). Type: SHT_NULL Attributes: SHF_ALLOC

<code>.secinfo</code> A linker-created table, 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: <code>SHT_PROGBITS</code> Attributes: <code>SHF_ALLOC</code>
<code>.syscall</code> A program code section to support the Green Hills system call mechanism. Type: <code>SHT_PROGBITS</code> Attributes: <code>SHF_EXECINSTR</code> <code>SHF_ALLOC</code>
<code>.stack</code> Section describing the area of memory that the program stack will occupy. Type: <code>SHT_NOBITS</code> Attributes: <code>SHF_ALLOC</code> <code>SHF_WRITE</code>

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 19.1. A Relocation Entry Corresponding to `SHT_REL`

```
typedef struct {  
    Elf32_Addr r_offset;  
    Elf32_Word r_info;  
} Elf32_Rel;
```

Example 19.2. 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: <pre>#define ELF32_ST_BIND(i) ((i)>>4) #define ELF32_ST_TYPE(i) ((i)&0xf) #define ELF32_ST_INFO(b,t) (((b)<<4) + ((t)&0xf))</pre>
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 883.

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.

Name	Value	Attributes
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.
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 876). 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	<i>None</i>
1	name.
7	Variable
11	able
16	able
24	<i>null string</i>

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;
```

<code>p_type</code>	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 894.
<code>p_offset</code>	Offset from the beginning of the file at which the first byte of the segment resides.
<code>p_vaddr</code>	Address at which the first byte of the segment resides in memory.
<code>p_paddr</code>	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.
<code>p_filesz</code>	Number of bytes in the file image of the segment; it may be zero.
<code>p_memsz</code>	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 895.
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

GHS Standard Mode	900

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 13, “Coding Standards” on page 717.

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 multibyte 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	

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	-Opeep
-fschedule-insns	-Opipeline
-fshort-enums	--short_enum
-fstrict-aliasing	-ansi_alias
-fsyntax-only	-syntax
-funroll-loops	-Oloop
-fvolatile	-Onomemory
-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	-OM
-fno-unroll-loops	-Onoloop
-march	-cpu
-mbig -mbig-endian -EB	-bigendian
-mbigswitch	-bigswitch
-mlittle -mlittle-endian -EL	-littleendian
-msda	-sda

**Note**

Some of the above options are only recognized by the driver:

- **-###**
- **-march**
- **-mlittle**
- **-mlittle-endian**
- **-msda**

**Note**

The **-Wall** option enables several warnings which are off by default. This option does not have an equivalent supported option and is deprecated. We recommend **--ghstd** as a more complete solution for detecting issues in source code. Otherwise, these separate options may be used to turn on some warnings which are controlled by **-Wall**: **-Wunknown-pragmas**, **-Wimplicit-int**, **-Wshadow**, **-Wtrigraphs**, **-Wundef**.

Appendix C

Green Hills Project File Format

Contents

Green Hills Project File Syntax	910
Top Project Directives	914
Conditional Control Statements	918
Macro Functions	922
File Types	923

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 and optionally contains macro definitions. 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 an imported environment variable by prefixing a 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
```



Note

Macros can be defined in project files (described below), build configuration (**.gpc**) files (see “Build Configuration Quick Start” on page 948), and customization (**.bod**) files (see “Preprocessor Directives” on page 974). Macros defined in customization files but not in project files will be inherited by project files. Macros defined in configuration files are only used when performing an applicable build.

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 923.

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

In addition to the required and optional lines documented above, Top Project headers must specify the `primaryTarget` Top Project directive, and may specify other Top Project directives. For example:

```
#!gbuild
# My Top Project
# Macros go here (optional)
# Group 1 directives go here (optional)
primaryTarget=arm64_standalone.tgt
[Project]
```

For information about Top Project and Group 1 directives, see “Top Project Directives” on page 914. For information about the `primaryTarget` directive, see “Group 2 Directives” on page 918.

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 (" "). Any backslashes in strings must be escaped with another backslash, otherwise they will be treated as escape characters. The following example shows options for a simple Top Project:

```
#!gbuild
# Header section
primaryTarget=arm64_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 project options section ends at the first occurrence of a line that is not a comment or indented by whitespace.

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 117). 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 923. 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=arm64_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. Unless otherwise stated, 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], remembering that group 1 directives must appear before group 2 directives.
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 488.

defineConfig(*descriptive_name short_name build_configuration_file.gpc*)

Defines a project build configuration. Project build configurations allow you to easily build a single project in different ways. For example, you might want a debug configuration:

```
defineConfig(Debug dbg debug.gpc)
```

which sets options that configure your project for maximum debugging ability, and a release configuration:

```
defineConfig(Release rel release.gpc)
```

which sets options that optimize your project for speed and size.

The configuration's *descriptive_name* should be easy to read and understand. It appears in a drop-down list that lets you choose between build configurations in the Project Manager.

The *short_name* should be easy to type. It is used in conditional control statements to indicate options that only take effect, or files that are only included, when building the *short_name* configuration. For more information, see “Conditional Control Statements” on page 918, and see the description of **config** in “Valid Conditions” on page 919.

The pre-defined macro `__CURRCFG` resolves to the short name of the active build configuration.

build_configuration_file.gpc is the path to a file containing options and macros that only take effect when you build the project in the *descriptive_name* configuration (see “Build Configuration Files” on page 917). These options and macros supplement the options and macros defined elsewhere in your project hierarchy; they do not replace them. The path is resolved relative to the location of the Top Project file.

If you do not want to use a build configuration file to define configuration-specific settings, specify `""` instead of specifying a path to a file.

By default, the first build configuration that you specify is the one that is used when you build your project via the **gbuild** command line utility.

**Note**

The **defineConfig** project directive is only supported in Compiler 2015.1 or newer. Using **defineConfig** will cause a **.gpj** file to be incompatible with earlier versions of the Green Hills toolchain and IDE.

externalDepends=*path_to_external_Top_Project*

Causes **gbuild** to search the Top Project specified by *path_to_external_Top_Project* when determining how to build dependencies. This is useful if programs within your Top Project depend on libraries or objects built by another **.gpj** project tree.

When you build a program, **gbuild** locates the Top Project specified by `externalDepends` and automatically builds the dependencies if they are not current. For more information, see “Implicit Dependency Analysis” on page 497.

To specify that a particular build configuration of another Top Project should be used when building dependencies, enclose that configuration's *descriptive_name* or *short_name* in square brackets after giving the location of the Top Project. For example:

```
externalDepends=${__BSP_DIR}/default.gpj      [Debug]
```

For more information, see the description of `defineConfig` above.



Tip

The `externalDepends` directive may be specified in a Top Project file or in a build configuration file (see “Build Configuration Files” on page 917). Specifying it in a build configuration file is useful if dependencies are only present in one configuration of the project, or if both the project with the dependency and the linked project can be built in multiple configurations. For example, you might want to link from one `Debug` configuration to another `Debug` configuration, and from one `Release` configuration to another `Release` configuration.

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 directive 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 directive. 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 directive 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 directive 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*.

Build Configuration Files

Build configuration files specified as part of a `defineConfig` directive (see “Group 1 Directives” on page 914) should be formatted as follows:

```
#!/gbuild
# Macros for the project build configuration go here (optional)
# externalDepends for the project build configuration go here (optional)
[Build Configuration]
# Options for the project build configuration go here
```

The first line (`#!/gbuild`) and the file type (`[Build Configuration]`) are required.

Macro definitions and `externalDepends` directives are optional. Macro definitions follow the same format used in project file headers. For more information, see “The Header Section” on page 911.

Each line containing an option must begin with a whitespace. Option lines may use macros.

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 F, “Customization Files” on page 955.

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.
- Each statement applies only to the line it is set on.
- 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.

Valid Conditions

The following are valid conditions for 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 956).

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.

config(short_name)

The option or file is included when you build the project in the configuration indicated by *short_name*, which is one of the names defined for the build configuration in the Top Project file. For more information, see “The Header Section” on page 911.

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 957).

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, you must put quotes (") around the string unless the string is a valid C-style identifier.

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 **arm64_standalone.tgt**, then `istarget("arm64_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 956.

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. (Note that arguments to options that accept strings and lists are ignored during the evaluation of this condition).

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 Conditions for Select One Projects

The following are special conditions for conditional control statements. They can only be applied to Select One **.gpj** files. For information about Select One projects, see the documentation about building platform specific programs from the same source files in the *MULTI: Managing Projects and Configuring the IDE* book.

Note that 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. Merged libraries can accommodate multiple files with the same name. Note that **Relocatable Object** files are not supported. For more information, see “Creating a Library from the Compiler Driver” on page 466.
 - 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 282), 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. (To configure the compiler driver to treat files with a **.C** extension as C source files rather than C++ source files, define the following macro in your Top Project file: `GHS_IGNORE_CAPS_FOR_C_FILE_EXT=1.`)
 - **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 429.

- 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.
- Build Configuration Files (**.gpc**):
 - Build Configuration — An options file that only takes effect when you build your project in a particular configuration.
- 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

Secure Coding Practices

Contents

Summary of Available Features	928
Recommended Coding Practices	929
Strategies for Ensuring Control-Flow Integrity	930
Supporting CERT C Rules	932
Green Hills Secure Coding Options Files	944

This appendix provides documentation on building software with attention to safety/security-critical systems.

Summary of Available Features

This section summarizes toolchain features that may be helpful in writing secure code. The features themselves are described in more detail elsewhere.

Static Checks

When applying new static checks to an existing codebase, it is best to start with individual rules that present the highest risk to the project. Existing code may need to be reworked to comply with the checks. Some checks may be too strict to practically be applied to existing code, in which case they should not be enabled.

The following features may assist in detecting problems at compile time:

- Green Hills Standard Mode enables a large selection of helpful diagnostics (see “Green Hills Standard Mode” on page 253).
- DoubleCheck statically analyzes paths through the program to identify bugs beyond those caught by compiler diagnostics (see Chapter 5, “The DoubleCheck Source Analysis Tool” on page 357).
- Many MISRA guidelines can be checked at build time (see “MISRA C 1998” on page 718, “MISRA C 2004” on page 721, and “MISRA C 2012” on page 723).
- Various individual diagnostics can be enabled to check CERT C rules (see “Supporting CERT C Rules” on page 932).
- In projects, a predefined options file may be specified to enable several MISRA and CERT C checks at once, among other checks (see “Green Hills Secure Coding Options Files” on page 944).
- The **gstack** utility program can be used to determine a program's maximum stack usage and thus help to avoid stack overflow bugs (see “The gstack Utility Program” on page 539).
- Diagnostic 2140 can be enabled to detect if any single function has a stack frame of at least 4096 bytes, a typical size of a guard page. Guard pages are used by some operating systems to detect stack overflows at run time. If a

single function has a stack frame this large, it could skip over the guard page, potentially allowing a stack overflow to go undetected.

The option `--quit_after_warnings` will cause a build to fail if any compiler warning is output, preventing failures in these checks from being ignored (see “Quit Building if Warnings are Generated” on page 252).

Run-Time Checks

The following features may be helpful in detecting problems at run time:

- Annex K bounds-checking functions provide functionality similar to preexisting C library functions, but with additional checks (see the C11 standard).
- Several options are provided to help ensure control-flow integrity (see “Strategies for Ensuring Control-Flow Integrity” on page 930).
- Run-time error and memory checks can be helpful when debugging (see “Run-Time Error Checks” on page 135 and “Run-Time Memory Checks” on page 138).

Recommended Coding Practices

Avoid `setjmp()`, `longjmp()`, and Exceptions

The functions `setjmp()` and `longjmp()`, as well as C++ exceptions, are a form of hidden control flow that make it harder to reason about a program's behavior. For example, a statement immediately following a function call might not be executed if the function calls `longjmp()` or throws an exception. Additionally, these features add a possible attack vector, since the `jmp_buf` or internal data structure used to implement exceptions may be corrupted in a buffer overflow. This can cause `longjmp()` or `throw` to branch to a potentially-malicious address.

Avoid Threads

With multiple threads, it can be challenging to reason about all possible thread interactions. This will be worsened by hardware-level optimizations that can cause memory writes in one thread to become visible in a different order in another thread.

Memory barriers implemented to counteract this behavior are difficult to use correctly. Furthermore, timing differences can cause threads to interact differently across runs, which will complicate testing for race conditions.

Limit Dynamic Memory Allocation to System Initialization

If memory is dynamically allocated while the system is running (rather than restricted to initialization) it will be harder to guarantee sufficient memory. One reason is that a call to `free()` or similar may be missed, resulting in a memory leak. Even if there are no memory leaks, some allocation patterns may cause the heap to slowly become fragmented over time, eventually resulting in an allocation failure.

Initialize Locals at Declaration

Local variables should be initialized at their point of declaration, even if it means writing a dummy value that will be overwritten later, such as zero. This simplifies debugging in instances where the later logic to write the variable misses a case. With an initializer at the declaration, such a bug will manifest consistently. With no initializer, the variable will contain an arbitrary value that can vary across runs. This practice can be enforced at compile time by enabling diagnostics 1959, 1968, and 2019.



Note

This advice does not apply to global or function-static variables, because these are already initialized to zero if no other initializer is specified.

Strategies for Ensuring Control-Flow Integrity

Control Flow Integrity (CFI) strategies attempt to protect against attacks intended to cause errant execution jumps. The mitigation strategies described below focus on stack smashing attacks. A stack smashing attack exploits a stack buffer overflow to corrupt data on the stack. Typically, the goal is to overwrite the saved return address, causing the current function to return to a maliciously chosen address instead of its caller. Note that while it is possible to greatly mitigate the danger of attacks on control flow, mitigations cannot substitute for well-written code. The best protection is to write a program that contains no possible buffer overflows.

**Note**

The functions `setjmp()` and `longjmp()` work by saving the program state (including return address) to a `jmp_buf` and restoring it later. If that `jmp_buf` is stored close to a buffer vulnerable to overflow, then an overflow could corrupt the `jmp_buf`, enabling the same type of attack. As such, we recommend against using `setjmp()` and `longjmp()` if CFI checking is desired. This also applies to C++ exceptions, which are implemented using a similar scheme.

Stack Canary

A stack canary is a sentinel value written to the stack at the start of each function and verified at the end. The expectation is that a buffer overflow will change the canary value before the return address, allowing the attack to be detected. For more information, see “Check for Stack Smashing Attacks” on page 94.

Shadow Stack

A shadow stack is a region of memory, separate from the stack, dedicated to storing a redundant copy of the return address for each function currently on the call stack. On entry to a function, the return address is pushed to the shadow stack. On exit, a value is popped from the shadow stack and compared to the return address. The expectation is that a buffer overflow that overwrites the stack will not affect the shadow stack, which will cause unequal values.

Some versions of INTEGRITY provide an integrated shadow stack implementation. Consult your INTEGRITY documentation to see if it has this support. If native support is unavailable, a shadow stack can be implemented by using **-instrument_prologue_epilogue** to insert calls at the start and end of each function to subroutines that push and pop from the shadow stack. For assistance with this process, contact Green Hills Support.

Note that if your program is prepared for **gstack** analysis then you can use the utility to determine the maximum required size of your shadow stack (see “Preparing Your Program for gstack Analysis” on page 540). To do so, invoke **gstack** on your program and use **-constant_stack_usage *N***, where *N* is the size of a pointer (in bytes). This invocation should be separate from the standard stack calculation. The option

-constant_stack_usage N causes **gstack** to treat each frame as being N bytes in size, which is the size of the shadow stack entries.

Forward-Edge CFI Attacks and Indirect Function Call Checks

Forward-edge attacks attempt to subvert the behavior of a program by corrupting and then manipulating the values of function pointers. Attackers may use corrupted function pointers to point to any executable component of a program, including the middle of a function.

The option **-indirect_function_call_checks** instructs the toolchain to create one or more tables that contain descriptors of functions that have had their address taken. The compiler can then perform checks when making indirect function calls to verify that destinations are legal addresses, allowing the attack to be detected. For more information, see “Indirect Function Call Checks” on page 95.

Supporting CERT C Rules

The SEI CERT C Coding Standard is a set of secure coding rules and recommendations for the C programming language. The CERT C standard was created by the Computer Emergency Response Team (CERT) Division at Carnegie Mellon University's Software Engineering Institute (SEI). Details are available in the book *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition)*. For more information, see the SEI CERT C website [<https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard>].

The Green Hills toolchain supports a subset of the CERT C rules. Enforcement of each rule is performed by an option set. Each option set provides one of three levels of coverage:

1. Enforced — Passing the given option(s) will generate an error if the rule is breached.
2. Partially enforced — Passing the given option(s) will generate an error if the rule is breached *in some circumstances*. Rules designated as partially enforced will provide information about the circumstances of error generation.
3. Requires coding practice — Passing the given option(s) will enforce a stricter rule that subsumes the CERT C rule. This will generate some false-positive errors for rule-compliant code.

Note the following caveats:

- Certain listed options are specifically tied to CERT C and will mention a specific rule number when an error is reported, while other options will not.
- Operating system headers or libraries may have built-in CERT C violations. For potential workarounds, see “Using MISRA C 2012 with Operating Systems and Libraries” on page 728 and `#pragma ghs nowarning` in “Green Hills Extension Pragma Directives” on page 752.
- All CERT C rules in the following categories are not supported: Concurrency (CON), Miscellaneous (MSC), POSIX (POS), and Windows (WIN).

- **PRE30-C. Do not create a universal character name through concatenation**
 - Not enforced.
- **PRE31-C. Avoid side effects in arguments to unsafe macros**
 - Not enforced.
- **PRE32-C. Do not use preprocessor directives in invocations of function-like macros**
 - Not enforced.
- **DCL30-C: Declare objects with appropriate storage durations**
 - Partially enforced with `--misra_2012=R18.6`. The option will check that the address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.
- **DCL31-C: Declare identifiers before using them**
 - Enforced with `--misra_2012=R8.1,R17.3`.
- **DCL36-C: Do not declare an identifier with conflicting linkage classifications**
 - Enforced with `--misra_2012=R8.8,R17.3`.
- **DCL37-C: Do not declare or define a reserved identifier**
 - Partially enforced with `--misra_2012=R21.1`. The option will catch `#defines`, but global declarations with underscores or reserved names will not be caught unless multiple definitions are present.
- **DCL38-C: Use the correct syntax when declaring a flexible array member**
 - Enforced with `--diag_error=2130`. The option will give an error when a struct definition has at least two fields and its last field is an array of size 1. (This may be incorrect syntax for a flexible array member.)
- **DCL39-C: Avoid information leakage when passing a structure across a trust boundary**
 - Not enforced.
- **DCL40-C: Do not create incompatible declarations of the same function or object**
 - Enforced with `--misra_2012=R17.3, -argcheck=normal, -globalcheck=normal`.

- **DCL41-C: Do not declare variables inside a switch statement before the first case label**
 - Enforced with **--diag_error=1821**.
- **EXP30-C: Do not depend on the order of evaluation for side effects**
 - Partially enforced with **--misra_2012=R13.2**. The option will not check for side-effects across function boundaries.
- **EXP32-C: Do not access a volatile object through a nonvolatile reference**
 - Enforced with **--misra_2012=R11.8**.



Note

The option **--misra_2012=R11.8** is stricter than required by EXP32-C as it will enable diagnostics if any qualifier is removed from a type (for example, if `const` is removed). This option will also partially enforce EXP40-C.

- **EXP33-C: Do not read uninitialized memory**
 - Partially enforced with **--misra_2012=R9.1, -double_check.level=high [Rules 14 and 17]**. The options can catch many (but not all) cases where uninitialized memory is used.
- **EXP34-C: Do not dereference null pointers**
 - Partially enforced with **-double_check.level=high [Rule 1]**. The option can catch some instances of dereferenced null pointers.
- **EXP35-C: Do not modify objects with temporary lifetime**
 - Enforced with **--diag_error=2137**. The option gives an error when a temporary array is accessed after its sequence point or when a temporary array is modified. This protects against undefined behavior. Temporary arrays come from expressions where the function returns a struct/union containing an array, such as `foo().a` and variants thereof.
- **EXP36-C: Do not cast pointers into more strictly aligned pointer types**
 - Enforced with **--diag_error=2123**. The option gives an error when a pointer is cast into another pointer type with stricter (larger) alignment, unless the casted pointer is `NULL` or from a call to `malloc()`, `realloc()`, `calloc()`, or `aligned_alloc()`. For example, a 4-byte aligned pointer may be cast to a 2-byte aligned pointer type, but attempting the reverse

(2-byte aligned to 4-byte aligned) would be undefined behavior if the original pointer address were not 4-byte aligned. As a corollary, any pointer type may be cast to `void*` or `char*` since these have no alignment requirements, while `void*` and `char*` themselves may only be cast to each other (or with the `NULL` or `malloc()` exceptions).

- **EXP37-C: Call functions with the correct number and type of arguments**
 - Enforced with `--misra_2012=R17.3, -argcheck=normal`.
- **EXP39-C: Do not access a variable through a pointer of an incompatible type**
 - Partially enforced with `--diag_error=1518,1519,1798`. The option will not check for cases where `realloc()` is used to change the type of a pointer.
- **EXP40-C: Do not modify constant objects**
 - Partially enforced with `--misra_2012=R7.4,R11.8, --diag_error=513`. The options cover implicit and explicit casts that remove `const` qualifiers. They will not give an error for C library functions that return a non-`const` pointer to a `const` array argument (such as `strchr()`). Additionally, they will not give an error for unions with overlapping `const` and non-`const` pointer fields, where a `const` pointer may be assigned to the `const` pointer field and written to via the non-`const` pointer field.



Note

The option `--misra_2012=R11.8` is stricter than required by EXP40-C as it will enable diagnostics if any qualifier is removed from a type (for example, if `volatile` is removed). Enabling this option will also enforce EXP32-C.

- **EXP42-C: Do not compare padding data**
 - Enforced with `--diag_error=2126`. The option gives an error when padded structs/unions are passed into `memcmp()`. This applies both within and at the end of a struct and protects against unexpected behavior when ostensibly equal structs fail to match because of unequal, indeterminate padding bytes.
- **EXP43-C: Avoid undefined behavior when using restrict-qualified pointers**

- Requires coding practice: do not use `restrict` keyword (this can be enforced with `--misra_2012=R8.14`). Note that implicit restrict pointers in library interfaces are not checked.
- **EXP44-C: Do not rely on side effects in operands to `sizeof`, `_Alignof`, or `_Generic`**
 - Partially enforced with `--misra_2012=R13.6`, `--diag_error=2141`. No diagnostic is given for `_Generic`. This is because side effects are often used deliberately in `_Generic` statements.
- **EXP45-C: Do not perform assignments in selection statements**
 - Enforced with `--diag_error=187`.
- **EXP46-C: Do not use a bitwise operator with a Boolean-like operand**
 - Enforced with `--diag_error=1892`.
- **EXP47-C: Do not call `va_arg` with an argument of the incorrect type**
 - Partially enforced with `--diag_error=1932,2132`, `-Wformat`. The `--diag_error=2132` option will give an error when `va_arg()` is called with a type in the second argument that is not possible, such as `char`, `short`, or `float`. This protects against undefined behavior. Note that other types of violations will not be detected, such as calling `va_arg()` too many times or calling with an incorrect type that is still promoted. The `-Wformat` option will detect mismatched format strings in `printf()` and `scanf()`-like functions.



Note

The option `--diag_error=1932` will enable diagnostics that prevent any non-literal format string to `printf()` and `scanf()`-like functions. This is stricter than required by EXP47-C, but is recommended to allow for static verification of matching parameters in the variable argument function and format string.

- **INT30-C: Ensure that unsigned integer operations do not wrap**
 - Enforced with `--diag_error=1702`, `-double_check.level=high`, `-double_check.enable=20`. Note that if a multi-dimensional array is accessed with integer arithmetic expressions in the subscripts, the toolchain will consider this arithmetic to fall under the scope of ARR30 instead.



Note

This rule requires explicit checking that unsigned integer operations do not overflow. For more information, see “20: Potential unsigned integer overflow - check operands” on page 382.

- **INT31-C: Ensure that integer conversions do not result in lost or misinterpreted data**
 - Not enforced.
- **INT32-C: Ensure that operations on signed integers do not result in overflow**
 - Enforced with `-double_check.level=high, -double_check.enable=19`.
Note that if a multi-dimensional array is accessed with integer arithmetic expressions in the subscripts, the toolchain will consider this arithmetic to fall under the scope of ARR30 instead.



Note

This rule requires explicit checking that signed integer operations do not overflow. For more information, see “19: Potential signed integer overflow - check or cast operands” on page 381.

- **INT33-C: Ensure that division and remainder operations do not result in divide-by-zero errors**
 - Not enforced.
- **INT34-C: Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand**
 - Not enforced.
- **INT35-C: Use correct integer precisions**
 - Not enforced.
- **INT36-C: Converting a pointer to integer or integer to pointer**
 - Not enforced.
- **FLP30-C: Do not use floating-point variables as loop counters**
 - Enforced with `--misra_2012=R14.1`.
- **FLP32-C: Prevent or detect domain and range errors in math functions**

- Not enforced.
- **FLP34-C: Ensure that floating-point conversions are within range of the new type**
 - Not enforced.
- **FLP36-C: Preserve precision when converting integral values to floating-point type**
 - Not enforced.
- **FLP37-C: Do not use object representations to compare floating-point values**
 - Not enforced.
- **ARR30-C: Do not form or use out-of-bounds pointers or array subscripts**
 - Partially enforced with `--diag_error=175, -double_check.level=high` [Rule 2]. The options can detect some cases where pointers or arrays are accessed out-of-bounds.
- **ARR32-C: Ensure size arguments for variable length arrays are in a valid range**
 - Requires coding practice: do not use variable length arrays (this can be enforced with `--misra_2012=R18.8`).
- **ARR36-C: Do not subtract or compare two pointers that do not refer to the same array**
 - Not enforced.
- **ARR37-C: Do not add or subtract an integer to a pointer to a non-array object**
 - Not enforced.
- **ARR38-C: Guarantee that library functions do not form invalid pointers**
 - Not enforced.
- **ARR39-C: Do not add or subtract a scaled integer to a pointer**
 - Partially enforced with `--diag_error=170`. The option will solely catch cases where adding to or subtracting from a pointer results in an out-of-bounds error for the object.
- **STR30-C: Do not attempt to modify string literals**
 - Enforced with `--misra_2012=R7.4, --diag_error=513`.

- **STR31-C: Guarantee that storage for strings has sufficient space for character data and the null terminator**
 - Partially enforced with **--diag_error=1932,2133,2134, -Wformat** or **-Wall**. The options will give an error when calling `scanf()`, `fscanf()`, and `sscanf()` with a string specifier that has no width, as well as an error when calling `sprintf()`. The use of these options will not detect other types of string buffer overflows.



Note

The option **--diag_error=1932** will enable diagnostics that prevent any non-literal format string to `printf()` and `scanf()`-like functions. This is stricter than required by STR31-C, but is recommended to allow for static verification of matching parameters in the variable argument function and format string.

- **STR32-C: Do not pass a non-null-terminated character sequence to a library function that expects a string**
 - Not enforced.
- **STR34-C: Cast characters to unsigned char before converting to larger integer sizes**
 - Not enforced.
- **STR37-C: Arguments to character-handling functions must be representable as an unsigned char**
 - Not enforced.
- **STR38-C: Do not confuse narrow and wide character strings and functions**
 - Not enforced.
- **MEM30-C: Do not access freed memory**
 - Partially enforced with **-double_check.level=high [Rule 6]**. The option can detect some cases where freed memory is accessed.
- **MEM31-C: Free dynamically allocated memory when no longer needed**
 - Partially enforced with **-double_check.level=high [Rule 4]**. The option can detect some instances of `malloc()` used without `free()`.
- **MEM33-C: Allocate and copy structures containing a flexible array member dynamically**

- Enforced with **--diag_error=2135,2136**. The option gives an error when a struct with a flexible array is dereferenced or allocated non-dynamically. In the former case, this protects against a potential copy of the incorrect size. In the latter case, this protects against unallocated space for the flexible array.
- **MEM34-C: Only free memory allocated dynamically**
 - Partially enforced with **-double_check.level=high [Rule 9]**. The option can detect some cases where freed memory was not allocated dynamically.
- **MEM35-C: Allocate sufficient memory for an object**
 - Partially enforced with **-double_check.level=high [Rule 2]**. The option can detect some cases where insufficient memory is allocated for an object.
- **MEM36-C: Do not modify the alignment of objects by calling realloc()**
 - Not enforced.
- **FIO30-C: Exclude user input from format strings**
 - Enforced with **--diag_error=1932, -Wformat**.
- **FIO32-C: Do not perform operations on devices that are only appropriate for files**
 - Requires coding practice: do not use **stdio.h** (this can be enforced with **--misra_2012=R21.6**).
- **FIO34-C: Distinguish between characters read from a file and EOF or WEOF**
 - Requires coding practice: do not use **stdio.h** (this can be enforced with **--misra_2012=R21.6**).
- **FIO37-C: Do not assume that fgets() or fgetws() returns a nonempty string when successful**
 - Requires coding practice: do not use **stdio.h** (this can be enforced with **--misra_2012=R21.6**).
- **FIO38-C: Do not copy a FILE object**
 - Enforced with **--misra_2012=R22.5**.
- **FIO39-C: Do not alternately input and output from a stream without an intervening flush or positioning call**
 - Requires coding practice: do not use **stdio.h** (this can be enforced with **--misra_2012=R21.6**).

- **FIO40-C: Reset strings on fgets() or fgetws() failure**
 - Requires coding practice: do not use **stdio.h** (this can be enforced with **--misra_2012=R21.6**).
- **FIO41-C: Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects**
 - Requires coding practice: do not use **stdio.h** (this can be enforced with **--misra_2012=R21.6**).
- **FIO42-C: Close files when they are no longer needed**
 - Requires coding practice: do not use **stdio.h** (this can be enforced with **--misra_2012=R21.6**).
- **FIO44-C: Only use values for fsetpos() that are returned from fgetpos()**
 - Requires coding practice: do not use **stdio.h** (this can be enforced with **--misra_2012=R21.6**).
- **FIO45-C: Avoid TOCTOU race conditions while accessing files**
 - Requires coding practice: do not use **stdio.h** (this can be enforced with **--misra_2012=R21.6**).
- **FIO46-C: Do not access a closed file**
 - Requires coding practice: do not use **stdio.h** (this can be enforced with **--misra_2012=R21.6**).
- **FIO47-C: Use valid format strings**
 - Enforced with **-Wformat, --diag_error=181,226,1932**.
- **ENV30-C: Do not modify the object referenced by the return value of certain functions**
 - Requires coding practice: do not call functions that return unmodifiable non-const objects. This can be enforced with **-symbol_blacklist_file**. Add the following functions to a symbol blacklist file: `getenv()`, `setlocale()`, `localeconv()`, `asctime()`, and `strerror()`.
- **ENV31-C: Do not rely on an environment pointer following an operation that may invalidate it**
 - Requires coding practice: do not use environment argument (this can be enforced with **--diag_error=2441**).
- **ENV32-C: All exit handlers must return normally**

- Requires coding practice: do not use exit handlers. This can be enforced with **-symbol_blacklist_file**. Add the following functions to a symbol blacklist file: `atexit()` and `at_quick_exit()`.
- **ENV33-C: Do not call `system()`**
 - Enforced with **-symbol_blacklist_file**. Add the following function to a symbol blacklist file: `system()`.
- **ENV34-C: Do not store pointers returned by certain functions**
 - Requires coding practice: do not call functions that return non-storable pointers. This can be enforced with **-symbol_blacklist_file**. Add the following functions to a symbol blacklist file: `getenv()`, `setlocale()`, `localeconv()`, `asctime()`, and `strerror()`.
- **SIG30-C: Call only asynchronous-safe functions within signal handlers**
 - Requires coding practice: do not use **signal.h** (this can be enforced with **--misra_2012=R21.5**).
- **SIG31-C: Do not access shared objects in signal handlers**
 - Requires coding practice: do not use **signal.h** (this can be enforced with **--misra_2012=R21.5**).
- **SIG34-C: Do not call `signal()` from within interruptible signal handlers**
 - Requires coding practice: do not use **signal.h** (this can be enforced with **--misra_2012=R21.5**).
- **SIG35-C: Do not return from a computational exception signal handler**
 - Requires coding practice: do not use **signal.h** (this can be enforced with **--misra_2012=R21.5**).
- **ERR30-C: Set `errno` to zero before calling a library function known to set `errno`, and check `errno` only after the function returns a value indicating failure**
 - Not enforced.
- **ERR32-C: Do not rely on indeterminate values of `errno`**
 - Requires coding practice: do not use **signal.h** (this can be enforced with **--misra_2012=R21.5**).
- **ERR33-C: Detect and handle standard library errors**
 - Not enforced.

- **ERR34-C: Detect errors when converting a string to a number**

- Enforced with **--diag_error=1932,2127,2128,2129**. The option disallows use of the `ato*()` string-to-number conversion functions, as well as calling `strto*()` with a null pointer as the second argument. To disallow number parsing out of strings using `scanf()`, `fscanf()`, and `sscanf()`, you must also use **-Wformat** or **-Wall**.

**Note**

The option **--diag_error=1932** will enable diagnostics that prevent any non-literal format string to `printf()` and `scanf()`-like functions. This is stricter than required by ERR34-C, but is recommended to allow for static verification of format strings passed to `scanf()`-like functions.

Green Hills Secure Coding Options Files

In project files, the following line may be used to include a set of rules for catching coding errors or security-compromising practices:

```
:optionsFile=${GHS_STANDARDS}/ghsecure_latest.opt
```

These rules include a selection of MISRA and CERT C checks alongside other miscellaneous secure coding options. (MISRA and CERT C checks that are overly strict or that tend to cause false positives are excluded from this file.) The syntax above will provide the latest version of the rules. To use a specific release version of the rules, replace `_latest` with a version number (such as `ghsecure2021_1.opt`).

In general, the **ghsecure*.opt** file should be enabled at the project's top level. For new projects, we recommended selecting a **ghsecure*.opt** file and then adhering to those rules throughout development. For existing projects, enabling the rules may initially lead to a large number of errors/warnings. In this case, we recommend the following procedure:

1. Temporarily disable the rules generating the most diagnostic messages. Following the option file's inclusion, this can be done by adding options such as **--diag_suppress** or **--misra_2012=-RX.Y**.

2. Address the rules generating the fewest diagnostic messages (these are most likely to be true positives).
3. After the project builds without errors, the list of disabled rules can then be reevaluated. Go through each rule and assess whether it is useful and thus desirable to enforce.



Note

The rules include the DoubleCheck static analysis tool, which may impact build times. If necessary, the **-double_check.level** option may be adjusted to a lower setting (see “Enabling DoubleCheck” on page 359). Note that if DoubleCheck is disabled, then secure coding rules that depend on it cannot be checked.

Appendix E

Porting an Existing Project to Use Build Configurations

Contents

Build Configuration Quick Start	948
Setting Configuration-Specific Files and Options Outside of Build Configuration Files	949
Defining Unique Output Directories for Build Configurations	950

Project build configurations allow you to easily build a single project in two or more different ways. For example, you can define a debug configuration and a release configuration with different build options set and/or different project macros defined for each.

Build configurations are defined in the header of the Top Project file. Configuration-specific options and macros are specified in build configuration files (**.gpc**).

Build Configuration Quick Start

Follow these steps to set up different configurations of your project:

1. Determine which project-wide options and macros you want to set in each configuration. Any options and macros that you do not want to be shared by all configurations should be specified in a build configuration file (**.gpc**). These options and macros affect the entire project, but only when it is built in the configuration to which the **.gpc** file applies. For example, setting the **-Odebug** option in **debug.gpc** affects the entire project when it is built in the debug configuration.
2. Use a text editor to create your build configuration files. Format them as specified in “Build Configuration Files” on page 917.
3. Make sure that the options you want to uniquely define for a configuration are set only in the appropriate build configuration file, not in a project file, which applies to all configurations. This is important to keep in mind when porting projects because you may have set options in a build configuration file that were already set in a project file. In this case, remove the options from the project file.
4. Define each of your configurations in the header of your project's Top Project file. To do so, add a **defineConfig** line for each configuration. For example:

```
defineConfig(Debug    dbg debug.gpc)
defineConfig(Release rel release.gpc)
```

For more information about the **defineConfig** directive, see “Group 1 Directives” on page 914. For information about where in the header to specify **defineConfig**—a group 1 directive—see “Top Project Headers” on page 911.

5. Open your project in the Project Manager, select each configuration and build it.

Alternatively, build from the command line by running **gbuild** with the **-cfg** option to specify the configuration you want to build. For example:

```
gbuild -cfg=rel
```

If you do not specify a configuration, the first one defined in your project is built.

Setting Configuration-Specific Files and Options Outside of Build Configuration Files

In addition to setting configuration-specific options and macros in your build configuration files, you can also indicate that any option or file in a project file applies only to one or more configurations.

You can specify that an option is configuration specific when you set it via the Project Manager's option editing dialog. The Project Manager automatically inserts a conditional control statement for the option in the designated project file, indicating that the option should only take effect when you build the specified configuration. If you want to bypass the GUI, open the project file in a text editor and manually insert the conditional control statement, formatting it as described in “Conditional Control Statements” on page 918. Use the **config** condition (see “Valid Conditions” on page 919). For example:

```
{config(dbg)} -Odebug  
{config(rel)} -Osize
```

To indicate that a file is configuration specific, you must use a text editor and manually insert the conditional control statement in the project file. Again, use the **config** condition. For example:

```
{config(dbg)} debug_hooks.c  
{config(rel)} hand_optimized_algorithm.asm
```

Defining Unique Output Directories for Build Configurations

For most projects, it is easiest to start with all configurations building into the output location used by the project before build configurations were introduced. However, sharing one output location for all build configurations has a major drawback: each time you want to switch to a different configuration, you need to rebuild the entire project. Rebuilding is required to ensure the entire project is built according to the newly chosen build configuration.

To eliminate this requirement, use a unique output location for each configuration. To do so, you must update:

- All output directories set in your project.
- Any input directory paths that rely on any of those output directories.

Updating Output Directories

To set output locations based on the configuration, you will need to do one of the following:

- Specify the necessary output location settings in each build configuration file (*.gpc).
- Use configuration-specific macros in output location settings throughout your project files.

The first option usually only makes sense for small projects. If your project already sets output locations in multiple places, you should use the second option.

The best way to set up a unique output location for each configuration is to include the name of the configuration in the name of the output directory. You can then use the `__CURRCFG` macro, which resolves to the short name of the active build configuration, when setting output locations in your files. For example, you could specify the following options in your Top Project:

```
:outputDir=${__CURRCFG}/work
-object_dir=${__CURRCFG}/work
:binDir=${__CURRCFG}/bin
```

This would result in an output directory structure like the following:

```
dbg/  
  work  
  bin  
rel/  
  work  
  bin
```

Output locations are commonly specified using one or more of the following options:

- **-object_dir**
- **:outputDir**
- **:outputDirRelative**
- **:binDir**
- **:binDirRelative**

In addition to updating the commonly used output directory options listed above, update any option in your project that specifies the path to an output location (for example, **-srec=c:\path\to\prog.srec**). You do not need to update options that only specify a filename (for example, **-srec=prog.srec**).



Tip

Once you change the output location specified in your project, the Builder does not remember the old output location. As a result, you should do a clean build *before* changing the output location; otherwise, you will need to manually remove files from the old location(s).

Updating Input Directories

In addition to setting output locations, you must also update any project settings that use files from those locations. Most commonly, this means updating your library `#include` paths (**-L**) to reference the new output locations.

For example, if your Top Project includes:

```
-L.
```

update it to something like:

```
-L${__CURRCFG}/bin
```

Handling Custom File Types

Files built using a custom processing tool defined in a customization (**.bod**) file may need special attention when you modify output directories.

If you are specifying the output location in the processing tool's command definition, use the `$OUTPUTDIR` variable in the command definition, and the Builder will substitute the correct output location for you based on the settings in the project. This allows you to specify an output directory to the tool without needing to set a tool-specific option in your project file.

In addition to updating the output locations for custom processing tools, you may also need to update the project settings that look for these files. Common options that may need updating include:

- **-I**
- **:sourceDir**
- **:sourceDirNonRelative**

For example, if your project uses a tool to convert data files into source code, and that source code's location is specified with:

```
-Igenerated
```

update it to something like:

```
-I${__CURRCFG}/generated
```

Verifying Correctness

To verify that each configuration of your project is built into a unique output location, follow these steps:

1. Ensure that your project's output directories are empty.
2. Build the first configuration of your project. Does it build successfully? Did it build into the location you expected?
3. If you answered yes to both questions, build the next configuration of your project using the **-strict_config** option to **gbuild**. For example:

```
gbuild -cfg=rel -strict_config
```

The **-strict_config** option forces a build error if output from the second build will overwrite previously built output.

Did the entire project build, with no errors, into a new location? Was it the location you expected?

If you defined three or more configurations, repeat step 3 as necessary.

We recommend that you always build with the **-strict_config** option. To make a project act as if you passed this option on the command line with every build, add the **defaultBuildOptions** directive to the header of your Top Project:

```
defaultBuildOptions=-strict_config
```

defaultBuildOptions is a group 1 directive. For information about where in the header to specify group 1 directives, see “Top Project Headers” on page 911.

Helpful Tricks

- If the build output for a single configuration contains more than one subdirectory, we recommend:

Creating a macro in your Top Project for your output base location. Be sure to specify the root directory of a relative path using the `%expand_path()` function. For example:

```
macro OUTPUT_BASE=${%expand_path(.)}/${__CURRCFG}
```

or:

```
macro OUTPUT_BASE=${%expand_path(.)}/output/${__CURRCFG}
```

Then, everywhere you need to specify an output subdirectory, use the macro:

```
-object_dir=${OUTPUT_BASE}/subdir
```

- You can view commands the Builder executes as it builds. This may be useful for determining exactly how various tools are being run. To view Builder commands, do one of the following:

- From the Project Manager, select **Build** → **Advanced Build**, and select **Show tool commands**.
- From the command line, run **gbuild** with the **-commands** option.

Additional Options to Check

The options that most commonly need to be updated are identified in the preceding sections. However, any option that allows you to specify a path may need to be updated when you start using a unique output directory for each build configuration. The following list identifies less common directory options and options that may, but do not commonly, include path information in their arguments. This list is not exhaustive.

Output:

- **-headerfiledir**
- **-list_dir**
- **-double_check.report**
- **-hex**
- **-list**
- **-memory**
- **-o**
- **-srec**

Input:

- **-includefiledir**
- **-startfile_dir**
- **-directive_dir**

Appendix F

Customization Files

Contents

Components of Customization Files	956
Extended Examples	961
Customization File Syntax	967

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 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"
    # This is the default file type for files with the .bin extension.
    extensions={"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 are automatically associated with the custom file type you are defining. Add the `command`, `commandLine` and `outputType` properties to specify how to build the custom file type. See “Customization File Syntax” on page 967 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 967.

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 occurrence 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 967.

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 918.

Extended Examples

The following sections provide additional reference examples to use when creating customization files.

Example F.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 F.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"
    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 969.
- `extensions` — A list of filename extensions to automatically associate with this file type (see “File Types” on page 923).

If more than one file type is defined to use the same filename extension, only the last file type defined is automatically associated with that extension.

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 969.

- `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 969.
- `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. A choke point is when additional jobs may not be dispatched until all currently running jobs finish. For example, linking a program may cause a choke point because it requires all of the program's objects to be built before linking can begin.
- `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 (see above definition).
- `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 <code>Commands</code> definition that matches the <code>command</code> property of the <code>FileType</code> . Resolves to an absolute path if the <code>runInDirectory</code> flag is set.
DISKFILE
Full path to the file being processed.
EDITOR
Executable of the graphical editor defined by the <code>FileType</code> 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 <code>FILENAMEBASE</code> except that it strips away the <code>lib</code> 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 <code>makeDepends</code> section of the <code>COMMAND</code> 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 <code>OUTPUTFILE</code> 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 <code>runInDirectory</code> flag set.
SECOND_PASS_OPTION	Same as the <code>secondPassOption</code> from the <code>FileType</code> 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 <code>siblingType</code> , which are descendents of the parent of the file being processed.
SIBLINGS[<i>type</i>]
Like SIBLINGS, but uses the specified type instead of the <code>siblingType</code> 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:

<code>%define</code> Defines a macro.
<code>%echo</code> <code>%info</code> Outputs a message.
<code>%elif</code> <code>%elsif</code> <code>%elseif</code> <code>%else</code> <code>%endif</code> <code>%if</code> Conditional inclusion directives. <code>%elif</code> and <code>%elsif</code> are equivalent to <code>%elseif</code> .
<code>%error</code> Outputs an error message.
<code>%ifdef</code> Shorthand for <code>%if defined</code> .
<code>%include</code> Includes another file.
<code>%warning</code> 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(MACRO_NAME, STRING, ...)` — Evaluates to true if the value of the macro matches at least one of the strings. Examples:

```
%if streq(__MULTI_HOST__, "linux86")
```

```
%if streq(__MULTI_HOST__, "win32", "win64")
```

Predefined Macros

You can use the following macros in any customization file:

- `__DIR__` — Directory of the file you are editing.
- `__MULTI_HOST__` — The host machine's operating system. Current values are as follows:
 - 64-bit Windows: `win64`
 - 64-bit Linux: `linux86_64`
- `__TOOLS_DIR__` — The MULTI Compiler installation directory.
- `GHS_TOOLS_VERSION` — The MULTI toolchain version in use.
- `GHS_STANDARDS` — The directory containing predefined options files that are designed to aid compliance with secure coding standards. For more information, see “Green Hills Secure Coding Options Files” on page 944.

The syntax for using one of these macros is `${MACRO}` (see examples below).

Example F.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
```

Example F.4. Passing Options Conditioned on Toolchain Version

In a **.gpj** file, the following code checks whether MULTI 7.0 or later is in use. If so, the option **--diagnostic_header_trace** is used.

```
{ isdefined(GHS_TOOLS_VERSION) && GHS_TOOLS_VERSION >= 70 }
--diagnostic_header_trace
```

Note that the `GHS_TOOLS_VERSION` macro is defined in ***install_dir/defaults/bld_rules/ghs_compiler_options.bod***.

Appendix G

The MULTI Eclipse Plug-in

Contents

Prerequisites to Installing MULTI for Eclipse	979
Installing MULTI for Eclipse	980
Creating a Green Hills Project from Within Eclipse	981
Changing Compiler Options	985
Building and Running Your Project	986
Debugging Your Project with the MULTI Debugger	988

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. You can download Eclipse and CDT from the Eclipse Web site [<http://www.eclipse.org/cdt>].
- Java SE 7. You can download Java from Oracle's Java Web site [<https://www.oracle.com/java/index.html>].
- The following combinations of Eclipse, CDT, and Java are supported:
 - Eclipse 4.5.x (Mars), CDT 8.8.x, and Java 7.
 - Eclipse 4.6.x (Neon), CDT 9.2.x, and Java 8.
 - Eclipse 4.7.x (Oxygen), CDT 9.3.x, and Java 8.
- 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 MSYS2, which you can download from the MSYS2 Web site [<https://www.msys2.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.



Note

The instructions in the following sections were written for Eclipse 4.5 and CDT 8.8. They should also apply to any of the supported combinations of versions listed above.

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** until you see the license acceptance page.
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 **Yes** 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. Green Hills project types are listed under "Others". 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 985.

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 981.

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++, linker scripts, and assembly source files. For C++ projects, you can add C, C++, and assembly source files. For C projects, you

can add C and assembly files. You may also add a single linker script to any 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, **.s** file extension for assembly source files, and the **.ld** file extension for linker scripts. 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.

For information about importing an existing file into your project, see the Eclipse documentation.

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 77.



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 981), 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, forward slashes are recommended over backslashes. If there are spaces in the setup script path, you should enclose the setup script path in double quotes, such as `setup="path"`.
 - 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 G.1. Connecting with a Setup Script on page 987.

5. Click **Apply** and then **Run**.

You can see the output of your project in the **Console** view.

Example G.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 Probe'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 986 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 Probe. 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 986. 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, forward slashes are recommended over backslashes. If there

are spaces in the setup script path, you should enclose the setup script path in double quotes, such as `setup="path"`.

- 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 G.1. Connecting with a Setup Script on page 987.

5. Click **Apply** and then **Debug**.

The MULTI Debugger opens and automatically connects to your target. Debug your project as usual.

Appendix H

Third-Party Software Licensing Information

Contents

Edison Group License	992
Other Licenses	992

Edison Group License

```

/*****
*
* Edison Design Group C++ Runtime
*
* Copyright 1992-2018 Edison Design Group Inc.
*
*****/
Redistribution and use in source and binary forms are permitted
provided that the above copyright notice and this paragraph are
duplicated in all source code forms. The name of Edison Design
Group, Inc. may not be used to endorse or promote products derived
from this software without specific prior written permission.
THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
Any use of this software is at the user's own risk.

```



Note

No Edison Group source code is provided as part of the toolchain, only binaries.

Other Licenses

Third-Party Tool	Description	License Type
BSD grep	Stand-alone grep utility used by the Debugger's Search in Files capability.	BSD license For more information, see the bsdgrep.txt file in the copyright subdirectory of your MULTI compiler installation directory.
DES	Software used by the gpatch utility in the MULTI IDE and compiler.	MIT license For more information, see the des.txt file in the copyright subdirectory of your MULTI compiler installation directory.

Third-Party Tool	Description	License Type
libiberty	Software library used for demangling C++ names in the compiler.	GNU Lesser General Public License (LGPL) v2 license For more information, see the demangle_gnu.txt file in the copyright subdirectory of your MULTI compiler installation directory.
PCRE	Software library used by the MULTI IDE and compiler for syntax coloring and running the gcolor utility.	BSD license For more information, see the libpcre.txt file in the copyright subdirectory of your MULTI compiler installation directory.
zlib	Software library used by various installers and the gpatch utility in the MULTI IDE and compiler.	BSD license For more information, see the zlib.txt file in the copyright subdirectory of your MULTI compiler installation directory.

Index

Symbols

'Select One' Project Extension List Builder option, 282
* (pointer) data type
 signedness, 205
*_reserved memory area, 433
-gnu99 Inline Semantics Builder option, 309
-Olimit= Without Debug Information Builder option, 296
64-bit Conversion Warnings Builder option, 258
64-bit integer arguments, 781
? operator, 671
@filename assembler option, 391
@filename driver option, 17

A

-A driver option, 242
.a file extension, 8, 466, 924
.a64 file extension, 8
abort(), C implementation, 626
ABS linker section attribute, 444
absolute assembler expression, 400
absolute linker expression function, 442
Accept GNU __asm__ statements Builder option, 311
-act_like=5.2 driver option, 113
-act_like=latest driver option, 114
Add Extra File to Debug Info Builder option, 299
__ADD_WITH_CARRY64() intrinsic function, 763
Additional Assembler Options Builder option, 225, 390
Additional Intex Options Builder option, 110
Additional Linker Options (among object files) Builder option, 233
Additional Linker Options (before start file) Builder option, 232, 424
Additional Linker Options (beginning of link line) Builder option, 231
Additional Output Files Builder option, 281
Additional System Library Builder option, 106
addr linker expression function, 442
Address Space Layout Randomization Seed Builder option, 314
.ael output file type, 14
alias GNU attribute, 637
.align assembler directive, 406
align linker expression function, 442
ALIGN linker section attribute, 444
Align Standalone Objects Builder option, 293
-align_standalone_objects driver option, 294
Aligned Data Sections Builder option, 97
aligned GNU attribute, 637
alignment
 bitfields, 603
 packing (see structure packing)
 requirements, 845
 type double, 742
alignof linker expression function, 442
-allabsolute linker option, 425
Allocation of Uninitialized Global Variables Builder option, 91, 639, 752
Allow C++ Style Slash Comments in C Builder option, 146
Allow Narrowing Conversions in C++11 Dialects Builder option, 223
Allow String Literals to be Assigned to non-const char Pointers Builder option, 223
-allow_ambiguous_abs linker option, 425
-allow_bad_relocations linker option, 425
-allow_cxx_duplicates driver option, 319
--allow_different_section_types driver option, 317
-allow_linkonce_duplicates driver option, 320
Alternate Assembler Directory Builder option, 285
Alternate Compiler Directory Builder option, 285
Alternate Compression Table Builder option, 323
Alternate Header Name Builder option, 323
Alternate Library Directory Builder option, 285
Alternate Linker Directory Builder option, 286

- alternate program start address
 - specifying, 74
- Alternate Source Name Builder option, 322
- Alternative Tokens Builder option, 209
- alternative_tokens driver option, 210
- always_inline GNU attribute, 638
- ambiguous_absolute_error linker option, 425
- Annotate Call/Return Instructions Builder option, 81
- annotate_call_return driver option, 81
- Annotated C++ Reference Manual, The (ARM), 681
- annotated examples
 - images for linking, converting, 963
- ANSI Aliasing Rules Builder option, 146
- ANSI C
 - limitations, 601
 - required macro names, 734
 - specifying, 577
- ANSI driver option, 142, 577, 596
- ansi driver option, 142, 577, 595
- ansi header file directory, 775
- ansi_alias driver option, 146
- any_output_suffix driver option, 273
- any_p_align linker option, 425
- Append Comment Section with Link-Time Information Builder option, 229
- Append Default Extension to Output Filename Builder option, 276
- append linker option, 425
- Append String to Every Section Name Builder option, 310
- :appendExtension Builder option, 276
- archive driver option, 12, 13, 466
- archiver (see librarian)
- archives (see libraries)
- argcheck driver option, 246
- argument field, 396
- arithmetic
 - machine specific, 852
 - pointer, 849
- ARM compliant C++, 681
- arm driver option, 144, 681
- __ARM_NEON__, 738
- .arm64 file extension, 8
- __ARM64__, 738
- arm64_errata_1183550 driver option, 82
- ARMv8.1, 26
- ARMv8.2, 26
- __ARRAY_OPERATORS, 745
- arrays
 - new and delete, predefined macro, 745
- asarm64 (see assembler)
- .ascii assembler directive, 408
- .asciiz assembler directive, 408
- aslr_seed driver option, 315
- asm driver option, 225, 390
- asm facility, 858
 - asm body, 864
 - asm macro, 859
 - con, 863
 - error, 864
 - examples, 859
 - farmem, 864
 - farsprel, 864
 - lab, 864
 - macro body, 859
 - mem, 864
 - nearmem, 864
 - pattern, 859
 - reg, 864
 - storage mode, 859
 - treg, 863, 864
 - ureg, 864
 - using asm macros, 861
 - writing asm macros, 866
- .asm file extension, 8, 924
- __asm keyword (see asm statements)
- asm keyword (see asm statements)
- asm statements
 - overview, 606
 - #pragma directive for, 749
- Asm Statements Builder option, 255, 595, 607
- asm_errors driver option, 255, 607
- asm_silent driver option, 255, 595, 607
- asm_warnings driver option, 255, 607
- asm3g driver option, 226
- asmcmd driver option, 226, 390
- asmfile_dir driver option, 270
- assembler, 390
 - assignment statements, 396
 - command line options for, 391
 - escape sequences, 394
 - expressions, 396
 - introduction to, xxxi
 - invoking with the driver, 59
 - listings, 415
 - passing options to through compiler driver, 390
 - running with compiler driver, 390
 - syntax, 392
- Assembler Command File Builder option, 226, 390
- assembler directives
 - .align, 406
 - .ascii, 408
 - .asciiz, 408

- .assert, 420
- .byte, 408
- .comm, 416
- .data, 413
- .define, 412
- .double, 408
- .dsect, 416
- .dword, 408
- .eject, 415
- .else, 407
- .elseif, 407
- .end, 416
- .endf ident, 419
- .endif, 407
- .endm, 412
- .endo ident, 419
- .endr, 413
- .equ, 397, 416
- .error, 421
- .exitm, 412
- .export, 416
- .file, 419
- .float, 408
- .fsize, 419
- .gen, 415
- .ghsnote, 463
- .global, 416
- .half, 408
- .ident, 421
- .if, 407
- .import, 416
- .include, 409
- .inspect, 416
- .lcomm, 416
- .list, 415
- .lsbss, 417
- .macro, 412
- .macrolocal, 412
- .maxinstances, 419
- .maxstack, 420
- .need, 421
- .nogen, 415
- .nolist, 415
- .note, 413
- .org, 413
- .previous, 413
- .rept, 413
- .rodata, 413
- .sbss, 417
- .sbttl, 415
- .scall, 420
- .section, 414
- .set, 397, 417
- .size, 420
- .space, 408
- .string, 408
- .struct, 418
- .subtitle, 415
- .text, 414
- .title, 415
- .undef, 412
- .warning, 421
- .weak, 419
- .word, 409
- assembler options
 - @filename, 391
 - cpu=cpu, 391
 - help, 391
 - I, 391
 - list, 392
 - o, 392
 - source, 392
 - V, 392
 - w,
- Assembler Warnings Builder option, 313
- assembler_warnings driver option, 313
- Assembly Files to Pre-Include Builder option, 304
- assembly language, 392
 - assignment statements, 396
 - comments, 396
 - constants for, 393
 - continuation lines, 396
 - escape sequences for, 394
 - expression types, 400
 - expressions, 396
 - identifiers for, 392
 - interface, 850
 - interleaving with source code, 225
 - labels, 401
 - line terminators, 396
 - operator precedence, 398
 - operators, 397
 - source statement syntax, 395
 - type combinations, 401
- .assert assembler directive, 420
- assert(), C implementation, 623
- assertions, static, 772
- assignment statements, assembler, 396
- at GNU attribute, 638
- AT linker section attribute, 444
- __attribute__ keyword, 637
- attributes

- ghs_noprofile, 642
- auto Type Keyword Support Builder option, 222
- auto_large_archive driver option, 318
- __AUTO_TYPE, 745
- auto_type driver option, 222
- :autoInclude driver option, 283
- Automatic Register Allocation Builder option, 290
- automatic two-pass inlining, 331
- Automatic Vector Optimization Builder option, 121, 346
- autoregister driver option, 290
- ax (see librarian)

B

- __BASE__, 743
- Basic block, 47
- __BIG_ENDIAN__, 738
- bigendian driver option, 79
- __bigendian keyword, 598
- bigswitch driver option, 266
- binary files, importing, 230
- binary memory image
 - generating, 228, 513
- binary operators, 397
- Binary Output Directory Relative to This File Builder option, 274
- Binary Output Directory Relative to Top-Level Project
 - Builder option, 274
- :binDir Builder option, 274
- :binDirRelative Builder option, 274
- bitfields, 26, 601, 846
 - alignment and size, 603
 - code efficiency, 602
 - portability, 847
 - signedness of, 205, 601
- .bmon output file type, 14
- .bod files (see customization files (.bod))
- __BOOL, 745
- bool data type, 745
- bool driver option, 213, 745
- Bool Type Support Builder option, 213
- Brief Diagnostic Message Mode Builder option, 254
- brief_diagnostics driver option, 254
- .bsp
 - file extension, 925
- bsp driver option, 36
- .bss program section, 61, 885
- buffering
 - C++ I/O streams, 781
- build configurations, project, 915
- Build Macro to Define Builder option, 284

- Builder, xxxii
- Builder options
 - 'Select One' Project Extension List, 282
 - gnu99 Inline Semantics, 309
 - Olimit= Without Debug Information, 296
 - 64-bit Conversion Warnings, 258
 - Accept GNU __asm__ statements, 311
 - Add Extra File to Debug Info, 299
 - Additional Assembler Options, 225, 390
 - Additional Intex Options, 110
 - Additional Linker Options (among object files), 233
 - Additional Linker Options (before start file), 232, 424
 - Additional Linker Options (beginning of link line), 231
 - Additional Output Files, 281
 - Additional System Library, 106
 - Address Space Layout Randomization Seed, 314
 - Align Standalone Objects, 293
 - Aligned Data Sections, 97
 - Allocation of Uninitialized Global Variables, 91, 639, 752
 - Allow C++ Style Slash Comments in C, 146
 - Allow Narrowing Conversions in C++11 Dialects, 223
 - Allow String Literals to be Assigned to non-const char Pointers, 223
 - Alternate Assembler Directory, 285
 - Alternate Compiler Directory, 285
 - Alternate Compression Table, 323
 - Alternate Header Name, 323
 - Alternate Library Directory, 285
 - Alternate Linker Directory, 286
 - Alternate Source Name, 322
 - Alternative Tokens, 209
 - Annotate Call/Return Instructions, 81
 - ANSI Aliasing Rules, 146
 - Append Comment Section with Link-Time Information, 229
 - Append Default Extension to Output Filename, 276
 - Append String to Every Section Name, 310
 - :appendExtension, 276
 - Asm Statements, 255, 595, 607
 - Assembler Command File, 226, 390
 - Assembler Warnings, 313
 - Assembly Files to Pre-Include, 304
 - auto Type Keyword Support, 222
 - Automatic Register Allocation, 290
 - Automatic Vector Optimization, 121, 346
 - Binary Output Directory Relative to This File, 274
 - Binary Output Directory Relative to Top-Level Project, 274
 - :binDir, 274
 - :binDirRelative, 274

- Bool Type Support, 213
- Brief Diagnostic Message Mode, 254
- Build Macro to Define, 284
- C Japanese Automotive Extensions, 143, 595, 596
- C Language Dialect, 142, 576, 577, 583, 585, 589, 595, 596, 601, 607
- C Standard Include Directories, 306
- C++ Exception Handling, 145, 667, 670
- C++ Inlining Level, 308
- C++ Language Dialect, 143, 145, 666, 667, 669, 672, 676, 678, 679, 681, 682
- C++ Libraries, 145, 667, 668
- C++ Linking Method, 312
- C++ Standard Include Directories, 307
- C/C++ Minimum/Maximum Optimization, 292
- Check for Stack Smashing Attacks, 94
- Checksum, 76, 246
- Code Factoring, 237, 460
- Coding Standard Profile, 253, 718, 729
- Commands to Execute After Associated Command, 279
- Commands to Execute After Associated Command (via Shell), 280
- Commands to Execute Before Associated Command, 276
- Commands to Execute Before Associated Command (via Shell), 278
- Common Subexpression Elimination, 291
- Consistent code without debug information, 296
- Constant Propagation, 291
- Convert C Memset to Memclr, 288
- Convert DWARF Files to MULTI (.dbo) Format, 304
- Create Raw Binary Image, 109
- Create S-Record File Image, 110
- Cross-Reference Information, 302
- Dbo File Version, 322
- Debug Info Linker Extra Arguments, 301
- Debug Information (.dbo) Tracing Diagnostics, 322
- Debugging Information, 303
- Debugging Level, 38, 40, 132, 286, 303, 761
- Deferral of Function Template Parsing, 220
- Define Linker Constant, 240
- Define Preprocessor Symbol, 140
- Definition of Standard Symbols, 305
- Definition of Unsafe Symbols, 305, 734
- Delete Unused Kernel Calls, 109
- Deletion of Unused C++ Virtual Functions, 237
- Deletion of Unused Data, 236
- Deletion of Unused Functions, 60, 235
- Demangling of Names in Linker Messages, 312
- Dependencies Relative to Source Directory, 274
- Dependencies Relative to This File, 274
- Dependencies Relative to Top-Level Project, 275
- Dependent Name Processing, 219, 670, 688
- :depends, 274
- :dependsNonRelative, 275
- :dependsRelative, 274
- Display BootTable, 109
- Display Diagnostic Numbers, 260, 261
- Display End-of-Function Symbols in Map File, 245
- Display includes Preprocessor Directives Listing, 140
- Display Local Symbols in Map File, 245
- Display Section Layout in Map File, 244
- Display Unused Functions in Map File, 244
- Display Version Information, 252
- Distinct C and C++ Functions, 313, 670
- Do not allow types double or long double, 267
- DoubleCheck Config File, 263
- DoubleCheck Errors to Enable, 263
- DoubleCheck Errors to Ignore, 263
- DoubleCheck Level, 262
- DoubleCheck Output that Stops Builds, 264
- DoubleCheck Report File, 262
- Duplicate Whitelist, 235
- Dynamic Download Project, 108
- EAGLE Logging, 133
- EAGLE Mute Signal, 295
- EAGLE Record Signal, 294
- EAGLE Strip Signal, 295
- EDG Front End Options, 321
- Emulate Behaviors of a Specific Compiler Version, 113
- Emulated GNU Version, 309
- Enable Toolchain Versions of Certain Memory Functions, 81
- End Files, 239
- Endianness, 79
- Error on Duplicate Symbols in an Archive, 319
- Exclude Error Message Paths, 254
- Executable Stripping, 228
- Explode Jumps, 289
- Extend Variable Liveness, 297
- Extra Dot in Individual Section Name, 102
- Extract Whole Archive, 234
- :extraOutputFile, 281
- Files Listing External Symbols, 125
- Files to Pre-Include, 304
- Floating Point Associativity, 86
- Floating-Point Functions, 87
- Floating-Point Fused Multiply and Add, 87
- Floating-Point Mode, 83
- Floating-Point Precise Signed Zero, 88
- Floating-Point Rigor, 85
- Follow Section, 317
- Force All Symbols From Library, 241

Force Deletion of Unused C++ Virtual Function Symbol, 316
Force Frame Pointer, 299
Force Undefined Symbol, 241
Full Breakdots, 297
Full POSIX-2001 Compilation Support, 108
Function Entry/Exit History Instruction Threshold, 134
Function Entry/Exit Logging, 133
Functions Without Prototypes, 255, 257
Generate Additional Output, 14, 228
Generate Function Entry/Exit Instrumentation, 294
Generate MULTI and DWARF Information, 303
Generate Target-Walkable Stack, 295
Generated Header File Directory, 108
Generated Header File Name, 110
Global Registers, 71, 92
GNU Compatibility Optimizations, 293
Green Hills Standard Mode, 253
Guiding Declarations of Template Functions, 218
Half-Precision Floating-Point Support, 88
Host and Target Character Encoding, 210, 610, 693
HTML File Compression, 322
Identifier Blacklist, 321
Identifier Definition, 311
Identifier Support, 312
Implicit Floating-Point/SIMD Register Use, 87
Implicit Int Return Type, 257
Implicit Use of 'std' Namespace, 213
Import Symbols, 241
Include Directories, 57, 58, 114
Include File Search Directory, 110
Include libbsp.a, 268
Incorrect Pragma Directives, 256, 748
Indirect Function Call Checks, 95
Individual Function Sections for Default Text Section, 100
Individual Function Sections for Renamed Text Sections, 100
Individual Function Sections for User-Defined Text Sections, 101
Individual Link-Once Sections, 102
Individual Variable Sections for Default Data Sections, 98
Individual Variable Sections for Nameless Variables, 99
Individual Variable Sections for Renamed Data Sections, 98
Individual Variable Sections for User-Defined Data Sections, 99
Inline C Memory Functions, 287
Inline C String Functions, 288
Inline Constant Math Functions, 286
Inline Larger Functions, 123
Inline Specific Functions, 131, 331
Inline Tiny Functions, 286
Input Languages, 272
Instantiate Extern Inline, 214, 670
Instrument Prologue/Epilogue, 96
INTEGRITY Application Stripping, 111
INTEGRITY Posix Mode, 108
INTEGRITY TimeMachine, 132
INTEGRITY TimeMachine Enabled, 297
Interleaved Source and Assembly, 225
Intermediate Output Directory Relative to This File, 273
Intermediate Output Directory Relative to Top-Level Project, 117
Intermodule Inlining, 121, 331
Interprocedural Optimizations, 122, 338, 339
Intex Variable Definition, 111
IP Alias Lib Functions, 130
IP Alias Reads, 129
IP Alias Writes, 129
IP Allocate Variables, 130
IP Constant Globals, 127
IP Constant Propagation, 128
IP Constant Returns, 129
IP Delete Functions, 126
IP Delete Globals, 126
IP Limit Inlining, 128
IP Localize Globals, 127
IP One-Site Inlining, 126
IP Remove Parameters, 128
IP Remove Returns, 129
IP Small Inlining, 128
Kernel Project, 107
Libraries, 59, 116
Library Directories, 59, 115
Line Numbers in Temporary Preprocessor Output, 272
Line Wrap Messages, 254
Link in Minimum Libraries, 315
Link in Standard Libraries, 315
Link Mode, 316
Link-Time Argument Type Checking, 246
Link-Time Constant Propagation, 237
Link-Time Global Variable Type Checking, 249
Link-Time Global Variable Type Qualifiers, 250
Link-Time Ignore Debug References, 238
Link-Time Trivial Function Inlining, 238
Linker Command File, 233
Linker Directive Files with Non-standard Extensions, 231
Linker Directives Directory, 314
Linker Optimizations, 122
Linker Warnings, 230

- Linker-Based Far Call Patching, 91
- Linking Sections with Different Types, 317
- Long Long Support, 206
- longjmp() Does Not Restore Local Vars, 310
- Loop Optimize Specific Functions, 131, 342
- Loop Unrolling, 289, 345
- Malloc Version, 102
- Map File Cross-Referencing, 244
- Map File Generation, 243
- Map File Page Length, 243
- Map File Retention, 243
- Map File Sorting, 243
- Max Depth for Recursive Template Instantiations, 221
- Maximize Optimizations, 124
- Maximum Number of Errors to Display, 251
- Memory Access, 205
- Memory Functions, 79
- Memory Model, 81
- Memory Optimization, 292, 854, 855
- Message Pragma, 305
- Misaligned Memory Access, 208
- MISRA C - Advisory Rules Level, 148
- MISRA C - Enforcement and Run-Time, 148
- MISRA C - Mandatory Rules Level, 147
- MISRA C - Required Rules Level, 148
- MISRA C 1998 Rules - Character Set, 150
- MISRA C 1998 Rules - Comments, 150
- MISRA C 1998 Rules - Constants, 151
- MISRA C 1998 Rules - Control Flow, 154
- MISRA C 1998 Rules - Conversions, 153
- MISRA C 1998 Rules - Declarations and Definitions, 152
- MISRA C 1998 Rules - Environment, 149
- MISRA C 1998 Rules - Expressions, 154
- MISRA C 1998 Rules - Functions, 155
- MISRA C 1998 Rules - Identifiers, 151
- MISRA C 1998 Rules - Initialization, 152
- MISRA C 1998 Rules - Operators, 153
- MISRA C 1998 Rules - Pointers and Arrays, 157
- MISRA C 1998 Rules - Preprocessor, 156
- MISRA C 1998 Rules - Standard Library, 158
- MISRA C 1998 Rules - Structs and Unions, 157
- MISRA C 1998 Rules - Types, 151
- MISRA C 2004 Rules - 1 Environment, 159
- MISRA C 2004 Rules - 10 Arithmetic Type Conversions, 164
- MISRA C 2004 Rules - 11 Pointer Type Conversions, 165
- MISRA C 2004 Rules - 12 Expressions, 166
- MISRA C 2004 Rules - 13 Control Statement Expressions, 167
- MISRA C 2004 Rules - 14 Control Flow, 168
- MISRA C 2004 Rules - 15 Switch Statements, 169
- MISRA C 2004 Rules - 16 Functions, 170
- MISRA C 2004 Rules - 17 Pointers and Arrays, 171
- MISRA C 2004 Rules - 18 Structs and Unions, 171
- MISRA C 2004 Rules - 19 Preprocessing Directives, 172
- MISRA C 2004 Rules - 2 Language Extensions, 160
- MISRA C 2004 Rules - 20 Standard Libraries, 174
- MISRA C 2004 Rules - 21 Run-time Failures, 175
- MISRA C 2004 Rules - 3 Documentation, 160
- MISRA C 2004 Rules - 4 Character Sets, 161
- MISRA C 2004 Rules - 5 Identifiers, 161
- MISRA C 2004 Rules - 6 Types, 162
- MISRA C 2004 Rules - 7 Constants, 162
- MISRA C 2004 Rules - 8 Declarations and Definitions, 163
- MISRA C 2004 Rules - 9 Initialization, 164
- MISRA C 2012 Guidelines - 1 Environment, 177
- MISRA C 2012 Guidelines - 10 The Essential Type Model, 185
- MISRA C 2012 Guidelines - 11 Pointer Type Conversions, 186
- MISRA C 2012 Guidelines - 12 Expressions, 187
- MISRA C 2012 Guidelines - 13 Side Effects, 188
- MISRA C 2012 Guidelines - 14 Control Statement Expressions, 189
- MISRA C 2012 Guidelines - 15 Control Flow, 190
- MISRA C 2012 Guidelines - 16 Switch Statements, 191
- MISRA C 2012 Guidelines - 17 Functions, 192
- MISRA C 2012 Guidelines - 18 Pointers and Arrays, 194
- MISRA C 2012 Guidelines - 19 Overlapping Storage, 195
- MISRA C 2012 Guidelines - 2 Unused Code, 178
- MISRA C 2012 Guidelines - 20 Preprocessing Directives, 195
- MISRA C 2012 Guidelines - 21 Standard Libraries, 197
- MISRA C 2012 Guidelines - 22 Resources, 200
- MISRA C 2012 Guidelines - 3 Comments, 179
- MISRA C 2012 Guidelines - 4 Character Sets and Lexical Conventions, 179
- MISRA C 2012 Guidelines - 5 Identifiers, 180
- MISRA C 2012 Guidelines - 6 Types, 181
- MISRA C 2012 Guidelines - 7 Literals and Constants, 181
- MISRA C 2012 Guidelines - 8 Declarations and Definitions, 182
- MISRA C 2012 Guidelines - 9 Initialization, 184
- MISRA C 2012 Guidelines - Directives, 201
- MISRA C 2012 Guidelines - Stricter R8.7, 203
- MISRA C 2012 Guidelines - Stricter R8.9, 203
- Monotonic Block Order, 318
- Multi-Core Configuration File, 300

Multi-Core Core Count, 301
Multiply-Defined Symbols, 240
Namespace Support, 213, 688
NaN Compares as Unordered, 86
New-Style Cast Support, 215
:nodepends, 275
Non-Standard Output Suffix, 273
:noSelfDepend, 275
Number of Arenas for Malloc, 105
Number of Callers to Track with Run-Time Memory Checks, 139
Object File Output Directory, 113
Optimization Strategy, 119, 124, 286, 761
Optimize All Appropriate Loops, 292, 341, 855
Optimize Specific Functions for Size, 131
Optimize Specific Functions for Speed, 130
Options to Apply to Project, 283
OS Directory, 106
Output Archives in Large Format, 318
Output File Size Analysis, 245
Output File Type, 227
Output Filename, 74, 116
Output Filename for Generic Types, 275
:outputDir, 117
:outputDirRelative, 273
:outputName, 275
Packing (Maximum Structure Alignment), 32, 33, 207, 760
Parse Templates in Generic Form, 219, 220
Pass Through Arguments, 282
:passThrough, 282
Pattern of Files to Pull Into Project, 283
Peephole Optimization, 293
Physical Address Offset From Virtual Address, 317
Pipeline and Peephole Scope, 123
Pipeline Instruction Scheduling, 293
Placement of Zero-Initialized Data, 92
POSIX DLL, 107
:postexec, 279
:postexecSafe, 280
:postexecShell, 280
:postexecShellSafe, 281
:preexec, 277
:preexecSafe, 277
:preexecShell, 278
:preexecShellSafe, 278
Prepend String to Every Section Name, 309
Preprocess Assembly Files, 224, 390
Preprocess Linker Directives Files, 229
Preprocess Special Assembly Files, 225
printf and scanf Argument Type Checking, 257
Profiling - Block Coverage, 134
Profiling - Target-Based Timing, 50, 135
Program is Single-Threaded, 105
Quit Building if Intex Warnings are Generated, 111
Quit Building if Warnings are Generated, 252
Range-Based For Loop Support, 221
Raw Import Files, 230
Record Pathname (Not linked), 242
Redirect Error Output to File, 252
Redirect Error Output to Overwriting File, 320
Reference, 282
Register Allocation by Coloring, 289
Relocatable Module, 268
Relocatable Module Base Image, 268
Remarks, 251
Rename Section, 97
Report an Error When Multiple Libraries on a Link Line Contain a Definition of the Same Symbol, 234
restrict Keyword Support, 214
Retain Comments During Preprocessing, 305
Retain Symbols for Unused Statics, 209
RTOS Include Directories, 307
RTOS Library Directories, 115
Run-Time Error Checks, 51, 135, 753
Run-Time Memory Checks, 53, 55, 138
Run-Time Type Information Support, 216
Safe Commands to Execute After Associated Command, 280
Safe Commands to Execute After Associated Command (via Shell), 281
Safe Commands to Execute Before Associated Command, 277
Safe Commands to Execute Before Associated Command (via Shell), 278
Scan source files to augment native debug info, 299
Search for DBA, 298
Search Path for .dbo Files, 298
Section Overlap Checking, 246
:select, 282
Self-Dependency, 275
Set Linker Message to Error, 261
Set Linker Message to Silent, 261
Set Linker Message to Warning, 261
Set Maximum DBO Not Found Warnings, 300
Set Message to Error, 260
Set Message to Remark, 260
Set Message to Silent, 260
Set Message to Warning, 260
SHA-1, 111
Shadow Declarations, 257
Shared Libraries, 52, 106

- Show Header Trace for Diagnostics, 254
 - Signedness of Bitfields, 204, 595, 601, 695
 - Signedness of Char Type, 204, 595, 694
 - Signedness of Pointers, 205
 - Simplify C Print Functions, 288
 - Size of `time_t`, 207
 - Source Directories Relative to This File, 117
 - Source Directories Relative to Top-Level Project, 273
 - Source Listing Generation, 59, 224, 225
 - Source Listing Generation Output Directory, 224
 - Source Root, 114
 - `:sourceDir`, 117
 - `:sourceDirNonRelative`, 273
 - Stack Limit Checking, 93
 - Standard Include Directories, 306
 - Start Address Symbol, 74, 229, 429
 - Start File Directory, 239
 - Start Files, 239
 - Straight-line Spectre Assembler Mitigation, 83
 - Straight-line Spectre Compiler Mitigation, 82
 - Straight-line Spectre Linker Mitigation Check, 83
 - Support `__noinline` Keyword for Templates, 214
 - Support floating point types in `scanf`, 314
 - Support for C Type Information in Assembly Files, 226
 - Support for Constructors/Destructors, 215
 - Support for Implicit Extern C Type Conversion, 313
 - Support for Implicit Typenames, 219
 - Support for Old-Style Specializations, 218, 219
 - Support for Very Large Switch Statements, 266
 - Support Variable Length Arrays, 209
 - Suppress Dependencies, 275
 - Switch Tables, 266
 - Symbol Blacklist, 320
 - Symbols Referenced Externally, 124
 - System Include Directories, 307
 - Tail Calls, 291
 - Target Processor, 265
 - Temporary Assembly File Directory, 270
 - Temporary Assembly Files, 270
 - Temporary Output, 270
 - Temporary Output Directory, 269
 - Temporary Preprocessor Output, 271
 - Temporary Preprocessor Output Directory, 271
 - Thread Safe Initializations, 222
 - Thread-Local Storage, 147
 - Treatment of RTTI as `const`, 217
 - Treatment of Virtual Tables as `'const'`, 218
 - Trigraphs, 258
 - Undefine Preprocessor Symbol, 140
 - Undefined Preprocessor Symbols, 258
 - Undefined Symbols, 240
 - Uniquely Allocate All Strings, 208, 694
 - Unknown Pragma Directives, 256, 748
 - Unroll Loops More, 124
 - Use Smallest Type Possible for Enum, 27, 206, 595, 604, 695
 - User-Defined Literals, 222
 - Virtual Function Definition, 217
 - Wait for Debug Translation Before Exiting, 300
 - Warn for all floating point types, 266
 - Warn for types `double` or `long double`, 267
 - Warnings, 251, 614
 - WChar Size, 310
 - When Rejecting Duplicate Symbols in An Archive, Certain Symbols Generated by the C++ Front End in COMMON Sections Are Ignored, 319
 - When Rejecting Duplicate Symbols in An Archive, Symbols in `.ghs.linkonce` Sections Are Ignored, 320
 - When Rejecting Duplicate Symbols, Allow Certain Symbols Added by the compiler, 234
 - When Rejecting Duplicate Symbols, Allow Symbols in `.ghs.linkonce` sections, 235
 - Workaround ARM64 Erratum 1183550 (DMB), 82
 - Workaround ARM64 Erratum 1183550 (NOP), 82
 - Write Version Info to `.comment` Section, 252
 - X-Switches, 321
 - building and debugging, different hosts for, 42
 - built-in functions (see *intrinsics*)
 - `__builtin_constant_p()` intrinsic function, 770
 - `__builtin_frame_address()` intrinsic function, 762
 - `__builtin_return_address()` intrinsic function, 762
 - byte addressable, 844, 848
 - `.byte` assembler directive, 408
 - byte order (see *endianness*)
 - `__bytereversed` keyword, 597
- ## C
- C (see *C language*)
 - extensions, 577
 - implementation-defined features
 - `abort()`, 626
 - arrays, 620
 - `assert()`, 623
 - bit-fields, 621
 - `calloc()`, 626
 - character set, execution, 633
 - characters, 616
 - environment, 614
 - `errno()`, 629
 - `exit()`, 626
 - files, 626

- floating point, 619
- getenv(), 615
- identifiers, 616
- #include directive, 623
- integers, 618
- isalnum(), 628
- isalpha(), 629
- iscntrl(), 629
- isdigit(), 629
- islower(), 629
- isprint(), 629
- isupper(), 629
- locale, 628
- malloc(), 626
- NULL, 623
- overview, 613
- pointers, 620
- registers, 620
- signals, 615
- sterror(), 629
- streams, 626
- strftime(), 633
- structures, 621
- tmpfile(), 625
- translation, 614
- unions, 621
- volatile accesses, 622
- undefined behavior
 - expressions, 613
- c driver option, 11, 13
- C driver option, 240, 305
- .c file extension, 8, 924
- .C file extension, 8, 924
- c HTML compiler option, 323
- C HTML compiler options, 323
- C Japanese Automotive Extensions Builder option, 143, 595, 596
- C language
 - and memory-mapped I/O, 74
 - ANSI dialect, 577, 601 (see ANSI C)
 - asm statements, 606
 - __bigendian keyword, 598
 - bitfields, 601, 603
 - building an executable, 6
 - __bypreversed keyword, 597
 - compiler driver, 5
 - compiler limitations, 611
 - const keyword, 600
 - enum data type, 604
 - extended characters, 608
 - GNU dialect, 588, 589, 591
 - Green Hills implementation of, 576
 - Japanese Automotive C, 595
 - __littleendian keyword, 598
 - MISRA C 1998, 149, 718
 - MISRA C 2004, 721
 - MISRA C 2012, 723
 - __packed keyword, 598
 - preprocessing, 607
 - reentrant functions in run-time libraries, 783
 - run-time libraries, 774
 - sequence related undefined behavior checking, 596
 - specifying a dialect of, 576
 - standard libraries, 774
 - <stdarg.h>, 605
 - struct data type, 600
 - type qualifiers, 597
 - union data type, 600
 - variable arguments, 605
 - volatile keyword, 598
- C Language Dialect Builder option, 142, 576, 577, 583, 585, 589, 595, 596, 601, 607
- C Standard Include Directories Builder option, 306
- C++ (see C++ language)
- C++ Exception Handling Builder option, 145, 667, 670
- C++ Inlining Level Builder option, 308
- C++ language
 - alignment and size limitations, 846
 - ARM compliant, 681
 - building an executable, 6
 - C main(), with, 839
 - compiler driver, 5
 - data types, 845
 - decode utility, 692
 - Embedded dialect, 679
 - Extended Embedded dialect, 681
 - extensions, 670
 - GNU dialect, 681
 - implementation-defined features, 693
 - in C programs, 841
 - inlining, 334
 - International Standard, 669
 - linkage, 691
 - namespaces, 688
 - post processing, 691
 - predefined macros for, 737
 - required macro names, 734
 - specifying a dialect of, 666
 - Standard, 669
 - (see also Standard C++)
 - standard libraries, 774
 - templates, 218, 219, 220, 685, 686

- C++ Language Dialect Builder option, 143, 145, 666, 667, 669, 672, 676, 678, 679, 681, 682
- C++ Libraries Builder option, 145, 667, 668
- C++ library, specifying, 667
- C++ Linking Method Builder option, 312
- C++ Programming Language, The, 671
- C++ Standard Include Directories Builder option, 307
 - C++11 driver option, 144, 672
 - c++11 driver option, 144, 672
 - C++14 driver option, 144, 676
 - c++14 driver option, 144, 676
 - C++17 driver option, 144, 678
 - c++17 driver option, 144, 678
 - c_and_cpp_functions_are_distinct driver option, 313
 - c_include_directory driver option, 306
 - __c_plusplus, 737
- C/C++ minimum/maximum optimization, 354
- C/C++ Minimum/Maximum Optimization Builder option, 292
 - C11 driver option, 142, 585
 - c11 driver option, 142, 585
 - C17 driver option, 142, 585
 - c17 driver option, 142, 585
 - C18 driver option, 143, 585
 - c18 driver option, 143, 585
 - C99 driver option, 142, 583
 - c99 driver option, 142, 583
 - call_shared driver option, 106
- calling conventions, xxxi, 29
- Calling printf from assembly code, 782
- calloc(), C implementation, 626
- .cc file extension, 8, 924
- ccarm64 compiler driver, 5
- cerr, 839
- char data type
 - signedness, 204, 739, 746
 - size, 738
 - variable arguments, 605
- __CHAR_BIT, 738
- __Char_Is_Signed__, 739, 746
- __Char_Is_Unsigned__, 739, 746
- __CHAR_UNSIGNED__, 739
- character
 - encoding, 26
- character constants, 394
- character set dependencies, 849
- check driver option, 138
- Check for Stack Smashing Attacks Builder option, 94
- check_straightline_spectre_mitigation driver option, 83
- check=all driver option, 136
- check=alloc driver option, 53, 138
- check=assignbound driver option, 136
- check=bounds driver option, 137
- check=memory driver option, 53, 138
- check=nilderef driver option, 137
- check=nomemory driver option, 138
- check=none driver option, 136
- check=switch driver option, 137
- check=watch driver option, 138
- check=zerodivide driver option, 137
- Checksum Builder option, 76, 246
- checksum driver option, 246
- Choke points, 969
- cin, 839
- classes, 846
 - anonymous, 671
 - portability, 847
 - template, 685
- CLEAR
 - linker section attribute, 444
- __CLZ32() intrinsic function, 763
- __CLZ64() intrinsic function, 763
- Code Factoring Builder option, 237, 460
- code factoring optimization
 - controlling the scope of, 461
 - incremental linking, 461
 - invoking, 461
 - overview, 460
 - partial, 462
- codefactor driver option, 237, 460
- Coding Standard Profile Builder option, 253, 718, 729
- coding standard profiles
 - creating, 729
 - introduction, 728
 - specifying, 729
 - syntax, 729
- coding_standard driver option, 253
- __COFF__, 743
- collating sequence, 849
- .comm assembler directive, 416
- COMMAND context sensitive variable, customization file, 970
- command line
 - for assembler, 391
 - for gdump utility, 402
- command line function calls, 45
- command line options, gpatch, 526
- command line utilities (see utility programs)
- commandcommand_name.gpj conditional control statement, 919
- commands
 - sourceroot, 42

- Commands to Execute After Associated Command (via Shell) Builder option, 280
- Commands to Execute After Associated Command Builder option, 279
- Commands to Execute Before Associated Command (via Shell) Builder option, 278
- Commands to Execute Before Associated Command Builder option, 276
- comment .gpj conditional control statement, 919
- comments, in assembler source, 396
- Common Subexpression Elimination Builder option, 291
- common subexpression elimination optimization, 352
- commons driver option, 91
- compiler driver, 4
 - (see also driver options)
 - building a C executable with, 6
 - building a C++ executable with, 6
 - controlling the assembler with, 59
 - controlling the linker with, 59
 - creating libraries with, 11
 - file extensions recognized by, 8
 - for C source files, 5
 - for C++ source files, 5
 - generating assembly files with, 11
 - generating debugging information with, 38
 - generating object files with, 11
 - input file types, 7
 - introduction to, xxx
 - linker directives files, passing to, 10
 - optimization (see optimizations)
 - optimizing your code with, 119
 - output file types, 10
 - passing multiple files to, 9
 - renaming output of, 74
 - running assembler with, 390
 - setting optimization options for, 119
 - specifying a target for, 35
 - syntax, 5
 - updating libraries with, 12
 - using a driver options file with, 17
 - using files with, 5
 - using options with, 5
- compiler drivers
 - ccarm64, 5
 - cxarm64, 5
- .con file extension, 925
- conditional assembly directives, 407
- conditional control statements
 - optgroup=option_group, 921
- conditional control statements, for .gpj files, 918, 919, 921
- config .gpj conditional control statement, 919
- configurations, project build, 915
- Consistent code without debug information Builder option, 296
 - consistentcode driver option, 296
- const GNU attribute, 639
- const keyword, 600
- constant data in ROM, 451
- Constant Propagation Builder option, 291
- constant propagation optimization, 353
- constants, in assembler, 393
- constructors, 691, 839, 841
- continuation lines, in assembler, 396
- conventions
 - typographical, xxvii
- Convert C Memset to Memclr Builder option, 288
- Convert DWARF Files to MULTI (.dbo) Format Builder option, 304
- converting debugging information (see translating, debugging information)
- copyright banner, generating, 252
- __COUNTER__, 746
- cout, 839
 - coverage=count driver option, 134
 - coverage=count64 driver option, 134
 - coverage=flag driver option, 134
 - coverage=none driver option, 134
- __cplusplus, 735
- .cpp file extension, 8, 924
- cppfile_dir driver option, 271
- cpu driver option, 38
 - cpu=cpu assembler option, 391
 - cpu=cpu driver option, 36, 265
- CRC, 76, 246, 479
- Create Raw Binary Image Builder option, 109
- Create S-Record File Image Builder option, 110
- CROM linker section attribute, 446, 449
- cross compilers, xxx
- Cross-Reference Information Builder option, 302
- crt0.a64, 827
- crt0.o startup module, 451, 827
- Cryptographic Extensions, 26
- customization files (.bod), 956
 - CommandOptions
 - definitions, 960
 - syntax, 972
 - Commands
 - definitions, 959
 - syntax, 972
 - components, 956
 - context sensitive variables
 - COMMAND, 970

DISKFILE, 970
 EDITOR, 970
 FILENAME, 970
 FILENAMEBASE, 970
 FILENAMELIBBASE, 970
 FILETYPEOPTIONS, 970
 INPUTDIR, 970
 INPUTFILE, 970
 ITEMID, 971
 KEEPOBJECTS, 971
 MAKE_DEPEND_OPTIONS, 971
 OBJECTBASE, 971
 OBJECTS, 971
 OBJECTSUFFIX, 971
 OPTIONS, 971
 OUTPUTDIR, 971
 OUTPUTFILE, 971
 OUTPUTNAMEBASE, 971
 PARENTID, 971
 RUNDIR, 971
 SECOND_PASS_OPTION, 971
 SIBLINGS, 972
 SIBLINGS[type], 972
 syntax, 969
 TOPPROJECT, 972
 examples, 961
 FileTypes
 definitions, 957
 syntax, 967
 including additional, 976
 predefined macro syntax, 975
 preprocessor directives
 %define, 974
 %echo, 974
 %elif, 974
 %else, 974
 %elseif, 974
 %elsif, 974
 %endif, 974
 %error, 974
 %if, 974
 %ifdef, 974
 %include, 974
 %info, 974
 syntax, 974
 %warning, 974
 preprocessor function syntax, 975
 customization Top Project directive, 918
 customizing
 run-time environment, 73
 cxarm64 compiler driver, 5

.cxx file extension, 8, 924

D

-D driver option, 140
 -d librarian command, 467
 -D linker option, 426
 .d output file type, 14, 24
 .data assembler directive, 413
 data initialization assembler directives, 408
 .data program section, 61, 885
 data structures
 thread-specific library, 786
 data types
 size, 845
 -data_delete driver option, 236
 __DATE__, 735
 .dba output file type, 15
 -dbg_source_root driver option, 114
 -dblink.argsdriver option, 301
 Dbo File Version Builder option, 322
 .dbo output file type, 15
 -dbo_trace driver option, 322
 --dbo_version driver option, 322
 -dbopath driver option, 298
 dead code elimination optimization, 355
 Debug Info Linker Extra Arguments Builder option, 301
 Debug Information (.dbo) Tracing Diagnostics Builder
 option, 322
 Debug Translator
 remapping host paths, 42
 source scanning feature, 42
 translating debugging information, 40, 41
 debugger
 displaying memory-mapped I/O, 75
 debugging
 GNU-compiled applications, 40, 44
 host, different than building host, 42
 memory allocation errors, 51
 programs compiled with third-party tools, 40, 41, 42, 44
 limitations, 43
 source level, 855
 symbolic, 419
 debugging information
 controlling level of, 303
 files containing, 40
 generated by third-party tools, 40, 41, 44
 limitations to, 43
 generating, 38
 generating DWARF, 303
 maintaining, 39

- obtaining profiling, 46
- target-based timing profiling, 49
- Debugging Information Builder option, 303
- Debugging Level
 - MULTI, 45
- Debugging Level Builder option, 38, 40, 132, 286, 303, 761
- decode utility, 692
- decrypt_key_file command line option, 526
- defaultBuildOptions Top Project directive, 915
- DEFAULTS linker directive, 431
- defer_parse_function_templates driver option, 220
- Deferral of Function Template Parsing Builder option, 220
- .define assembler directive, 412
- Define Linker Constant Builder option, 240
- %define preprocessor directive, customization file, 974
- Define Preprocessor Symbol Builder option, 140
- defineConfig Top Project directive, 915
- Definition of Standard Symbols Builder option, 305
- Definition of Unsafe Symbols Builder option, 305, 734
- delete
 - array, predefined macro, 745
- delete driver option, 235
- Delete Unused Kernel Calls Builder option, 109
- deleting
 - unused functions, 459
- Deletion of Unused C++ Virtual Functions Builder option, 237
- Deletion of Unused Data Builder option, 236
- Deletion of Unused Functions Builder option, 60, 235
- demangler, 692
- Demangling of Names in Linker Messages Builder option, 312
- .dep output file type, 15, 24
- dep_name driver option, 219, 670, 688
- dependencies on other Top Projects, 916
- Dependencies Relative to Source Directory Builder option, 274
- Dependencies Relative to This File Builder option, 274
- Dependencies Relative to Top-Level Project Builder option, 275
- dependency information, generating, 23
- Dependent Name Processing Builder option, 219, 670, 688
- :depends Builder option, 274
- :dependsNonRelative Builder option, 275
- :dependsRelative Builder option, 274
- deprecated GNU attribute, 639
- deprecated_nonconst_string_literal_conv driver option, 223
- derefed DoubleCheck property type, 364
- destructors, 691, 839, 841
- __DI() intrinsic function, 763
- diag_error driver option, 260
- diag_remark driver option, 260
- diag_suppress driver option, 260
- diag_warning driver option, 260
- diagnostic_header_trace driver option, 254
- __DIR() intrinsic function, 763
- directive_dir driver option, 314
- directories
 - library, 775
- disable_ctors_dtors driver option, 215
- disable_noinline driver option, 214
- discard_zero_initializers driver option, 92
- DISKFILE context sensitive variable, customization file, 970
- Display BootTable Builder option, 109
- Display Diagnostic Numbers Builder option, 260, 261
- Display End-of-Function Symbols in Map File Builder option, 245
- Display includes Preprocessor Directives Listing Builder option, 140
- Display Local Symbols in Map File Builder option, 245
- Display Section Layout in Map File Builder option, 244
- Display Unused Functions in Map File Builder option, 244
- Display Version Information Builder option, 252
- display_boot intex option, 109
- display_diag_number driver option, 261, 756
- display_toc librarian option, 469
- Distinct C and C++ Functions Builder option, 313, 670
- division
 - operators, 853
- .dla output file type, 15
- .dlo output file type, 15
- dM driver option, 13
- __DMB_OPT(); intrinsic function, 768
- .dnm output file type, 15
- Do not allow types double or long double Builder option, 267
- do_not_wait_for_dblink driver option, 300
- document set, xxvi
- dotciscxx driver option, 9
- .double assembler directive, 408
- double_check.config driver option, 263
- double_check.enable driver option, 263
- double_check.ignore driver option, 263
- double_check.level=high driver option, 262, 359
- double_check.level=low driver option, 262, 359
- double_check.level=medium driver option, 262, 359
- double_check.level=none driver option, 262, 359
- double_check.report driver option, 262, 366
- double_check.stop_build=errors driver option, 264, 367
- double_check.stop_build=off driver option, 264, 367

- double_check.stop_build=warnings driver option, 264, 367
 - __DOUBLE_HL, 740
 - DoubleClick
 - controlling output, 367
 - custom functions, using, 361
 - error and warning messages, 367, 368, 369
 - function properties, specifying
 - with #pragma directives, 362
 - with configuration files, 361
 - overview, 358
 - property types
 - derefed, 364
 - frees, 364
 - malloc_size, 363
 - malloced, 363
 - needs_null_check, 363
 - no_return, 365
 - no_return_when_non_zero, 364
 - no_return_when_zero, 364
 - uninit, 365
 - unsaved, 365
 - specifying report files, 360
 - using, 359
 - viewing reports, 366
 - DoubleClick Config File Builder option, 263
 - DoubleClick Errors to Enable Builder option, 263
 - DoubleClick Errors to Ignore Builder option, 263
 - DoubleClick Level Builder option, 262
 - DoubleClick Output that Stops Builds Builder option, 264
 - DoubleClick Report File Builder option, 262
 - # driver option, 16
 - driver option, 297
 - .dsect assembler directive, 416
 - __DSP_OPT(); intrinsic function, 768
 - dual_debug|dwarf2 driver option, 303
 - Duplicate Whitelist Builder option, 235
 - duplicate_whitelist driver option, 235
 - DWARF debugging information
 - converting, 40, 41, 42
 - locating, 42
 - dwarf2dbo driver option, 304
 - dwarf2dbo executable, 41, 42
 - .dword assembler directive, 408
 - dyload driver option, 268
 - Dynamic Download Project Builder option, 108
 - dynamic initialization, 691
 - dynamic intex option, 108
- E**
- e driver option, 144, 679, 737
 - E driver option, 13, 608
 - e driver options, 229
 - (see also -entry)
 - e_ehsize ELF header field, 878
 - e_entry ELF header field, 878
 - e_flags ELF header field, 878
 - e_ident ELF header field, 876
 - e_machine ELF header field, 877
 - e_phentsize ELF header field, 878, 893
 - e_phnum ELF header field, 878, 893
 - e_phoff ELF header field, 878
 - e_shentsize ELF header field, 878, 881
 - e_shnum ELF header field, 878, 881
 - e_shoff ELF header field, 878, 881
 - e_shstrndx ELF header field, 878, 892
 - e_type ELF header field, 876
 - e_version ELF header field, 877
 - EAGLE Logging Builder option, 133
 - EAGLE Mute Signal Builder option, 295
 - EAGLE Record Signal Builder option, 294
 - EAGLE Strip Signal Builder option, 295
 - earlyabsolute linker option, 426
 - EC++ (see Embedded C++)
 - %echo preprocessor directive, customization file, 974
 - Eclipse integration (see MULTI for Eclipse)
 - ecxx header file directory, 775
 - EDG Front End Options Builder option, 321
 - __EDG__, 737
 - __EDG_IMPLICIT_USING_STD, 745
 - __EDG_RUNTIME_USES_NAMESPACES, 745
 - EDITOR context sensitive variable, customization file, 970
 - ee driver option, 145, 681, 737
 - EEC++ (see Extended Embedded C++)
 - eecxx header file directory, 775
 - eel driver option, 145, 737, 777
 - eele driver option, 145, 737, 777
 - __EI() intrinsic function, 763
 - EI_CLASS ELF identification index, 879, 880
 - EI_DATA ELF identification index, 879, 880
 - EI_MAGn ELF identification index, 879
 - EI_NIDENT ELF identification index, 879, 880
 - EI_PAD ELF identification index, 879, 880
 - EI_VERSION ELF identification index, 879, 880
 - __EIR() intrinsic function, 763
 - .eject assembler directive, 415
 - el driver option, 145, 737, 777
 - ele driver option, 145, 737, 777
 - ELF file format
 - data types, 875
 - header fields, 876
 - identification indexes, 879

- introduction, 874
- object and executable file formats, 874, 888
- optional header output, 893
- program attribute flags, 895
- program section names, 885
- program sections, 885
- program types, 894
- relocation directories, 888
- relocation fields, 889
- section attribute flags, 884
- section header fields, 881
- section types, 883
- special section indexes, 883
- string tables, 892
- symbol bindings, 890
- symbol table entries, 889
- symbol types, 891
- symbol values, 891
- ELF files
 - printing with gdump utility, 402
- `__ELF__`, 744
- `%elif` preprocessor directive, customization file, 974
- `.else` assembler directive, 407
- `%else` preprocessor directive, customization file, 974
- `.elseif` assembler directive, 407
- `%elseif` preprocessor directive, customization file, 974
- `%elsif` preprocessor directive, customization file, 974
- `elxr` (see linker)
- `-elxr_diag_error` driver option, 261
- `-elxr_diag_suppress` driver option, 261
- `-elxr_diag_warning` driver option, 261
- Embedded C++
 - features of, 679
 - predefined macro, 737
- embedded development
 - and memory-mapped I/O, 74
 - and multiple-section programs, 62
 - and ROM, 451
 - symbolic memory-mapped I/O, 74
- `__EMBEDDED_CXX`, 737
- `__EMBEDDED_CXX_HEADERS`, 737
- Emulate Behaviors of a Specific Compiler Version Builder option, 113
- Emulated GNU Version Builder option, 309
- Enable Toolchain Versions of Certain Memory Functions Builder option, 81
- `--enable_ctors_dtors` driver option, 215
- `-enable_eagle_log` driver option, 133
- `--enable_noinline` driver option, 214
- `.end` assembler directive, 416
- End Files Builder option, 239
- `endaddr` linker expression function, 442
- `.endf` ident assembler directive, 419
- `-endfile` driver option, 239
- endianness
 - portability problems, 845
 - predefined macros for, 738
 - specifying, 79, 509, 533
- Endianness Builder option, 79
- `.endif` assembler directive, 407
- `%endif` preprocessor directive, customization file, 974
- `.endm` assembler directive, 412
- `.endo` ident assembler directive, 419
- `.endr` assembler directive, 413
- `-entry` driver option, 229
 - (see also `-e`)
- `-entry_exit_history_inst_threshold` driver option, 134
- enum data type, 604
 - size, 206
- `.env` output file type, 15
- `.equ` assembler directive, 397, 416
- `-errmax` driver option, 251
- `errno()`, C implementation, 629
- `.error` assembler directive, 421
- error linker expression function, 442
- error messages
 - in assembler listing, 420, 421
 - treating warnings as, 252
- Error on Duplicate Symbols in an Archive Builder option, 319
- `%error` preprocessor directive, customization file, 974
- `-error_basename` driver option, 254
- escape sequences, in assembler, 394
- evaluation order, 852
- `__EXCEPTION_HANDLING`, 745
- exceptions, 681
 - predefined macro, 745
- `__EXCEPTIONS`, 745
- `--exceptions` driver option, 145, 670
- Exclude Error Message Paths Builder option, 254
- executable files, 227
 - ELF format, 874, 888
- Executable Stripping Builder option, 228
- `exit()`, C implementation, 626
- `.exitm` assembler directive, 412
- `%expand_path(relative_path)` .gpj macro function, 922
- Explode Jumps Builder option, 289
- `.export` assembler directive, 416
- expressions
 - assembler, 396, 400
- expressions, assembler, 396, 400
- Extend Variable Liveness Builder option, 297

- extend_liveness driver option, 297
 - extended characters, 608, 610
 - Extended Embedded C++
 - predefined macro, 737
 - __EXTENDED_EMBEDDED_CXX, 737
 - __EXTENDED_EMBEDDED_CXX_HEADERS, 737
 - extern storage class specifier
 - ``C", 671
 - ``C++", 671
 - syntax, 691
 - use of, 691, 840
 - external driver option, 125
 - external_file driver option, 125
 - externalDepends Top Project directive, 916
 - externally_visible GNU attribute, 640
 - Extra Dot in Individual Section Name Builder option, 102
 - extra_filename driver option, 299
 - Extract Whole Archive Builder option, 234
 - extractall
 - linker option, 458
 - extractall driver option, 241
 - extractall linker option, 426
 - extractweak linker option, 426
 - extractweaklinker option, 458
 - :extraOutputFile Builder option, 281
- ## F
- far function calls, 72
 - __farcall, 73
 - farcallpatch driver option, 92
 - fast driver option, 130
 - fenv.h
 - functions, 818
 - ffunctions driver option, 87
 - fgnu89-inline driver option, 309
 - fhard driver option, 84
 - __Field_Is_Signed__, 739
 - __Field_Is_Unsigned__, 739
 - .file assembler directive, 419
 - file extension
 - .bsp, 925
 - .int, 925
 - file extensions, 8
 - file inclusion directives, 409
 - file types
 - Project Manager, 923
 - setting manually, 9
 - __FILE__, 735
 - FILENAME context sensitive variable, customization file, 970
 - FILENAMEBASE context sensitive variable, customization file, 970
 - FILENAMESLIBBASE context sensitive variable, customization file, 970
 - Files Listing External Symbols Builder option, 125
 - Files to Pre-Include Builder option, 304
 - filetype.suffix driver option, 9
 - filetypefiletype .gpj conditional control statement, 919
 - FILETYPEOPTIONS context sensitive variable, customization file, 970
 - FILL linker section attribute, 444
 - final GNU attribute, 640
 - final linker expression function, 442
 - .fixaddr program section, 452, 887
 - .fixtype program section, 452, 887
 - .float assembler directive, 408
 - float data type
 - variable arguments, 605
 - float_associativity driver option, 86
 - float_scanf driver option, 314
 - <float.h> header file, 844
 - floating point
 - half-precision type, enabling, 88
 - language-independent library, 823
 - predefined macros for, 740
 - range, 850
 - Floating Point Associativity Builder option, 86
 - floating-point
 - values, 26
 - Floating-Point Functions Builder option, 87
 - Floating-Point Fused Multiply and Add Builder option, 87
 - Floating-Point Mode Builder option, 83
 - Floating-Point Precise Signed Zero Builder option, 88
 - Floating-Point Rigor Builder option, 85
 - fNaN_cmp_unordered driver option, 86
 - fno-gnu89-inline driver option, 309
 - fno_NaN_cmp_unordered driver option, 86
 - fnodouble driver option, 267
 - fnofunctions driver option, 87
 - fnone driver option, 84
 - fnonodouble driver option, 267
 - Follow Section Builder option, 317
 - follow_section driver option, 317
 - Force All Symbols From Library Builder option, 241
 - Force Deletion of Unused C++ Virtual Function Symbol Builder option, 316
 - Force Frame Pointer Builder option, 299
 - Force Undefined Symbol Builder option, 241
 - __force_stores() intrinsic function, 767
 - force_uvfd driver option, 316
 - force_vtbl driver option, 217

- format GNU attribute, 640
 - format_arg GNU attribute, 642
 - free() standard library function
 - errors related to, 51
 - frees DoubleCheck property type, 364
 - friend classes, 670
 - frigor=accurate driver option, 85
 - frigor=default driver option, 85
 - frigor=fast driver option, 85
 - .fsize assembler directive, 419
 - fsoft driver option, 84
 - Full Breakdots Builder option, 297
 - Full POSIX-2001 Compilation Support Builder option, 108
 - full_breakdots driver option, 297
 - full_debug_info driver option, 303
 - __FULL_DIR__, 743
 - __FULL_FILE__, 743
 - __FUNCPTR_BIT, 739
 - function declaration, far function call, 73
 - function entry/exit history, 45
 - Function Entry/Exit History Instruction Threshold Builder option, 134
 - Function Entry/Exit Logging Builder option, 133
 - function properties, specifying for DoubleCheck, 361, 362
 - __FUNCTION__, 743
 - functions
 - aligning, 748
 - calling conventions, 847
 - compiler generated, 687
 - deleting unused, 459
 - inline, 688, 749
 - keeping unused, 459
 - prototype, C versus. C++, 842
 - pure virtual, 688
 - return values, 848
 - variable argument, 605
 - Functions Without Prototypes Builder option, 255, 257
 - fused_madd driver option, 87
- G**
- G driver option, 38, 40, 45, 132, 779
 - g driver option, 132
 - g++ driver option, 144, 682
 - ga driver option, 299
 - gaddr2line utility, 475
 - gasmlist utility, 476
 - gbin2c utility, 478
 - gbincmp utility, 486
 - gbuild utility, 488
 - Implicit Dependency Analysis (IDA), 497
 - using with interprocedural optimizations, 339
 - gbuildDir Top Project directive, 917
 - gcc driver option, 143, 589
 - gcolor utility, 499
 - gcompare utility, 502
 - gdump utility, 402, 505
 - .gen assembler directive, 415
 - gen_entry_exit_history driver option, 294
 - gen_timemachine driver option, 297
 - Generate Additional Output Builder option, 14, 228
 - Generate Function Entry/Exit Instrumentation Builder option, 294
 - Generate MULTI and DWARF Information Builder option, 303
 - Generate Target-Walkable Stack Builder option, 295
 - Generated Header File Directory Builder option, 108
 - Generated Header File Name Builder option, 110
 - generatemonolithdeletions intex option, 109
 - __get_stack_pointer(void) intrinsic, 762
 - getenv(), C implementation, 615
 - gfile utility, 511
 - gfunsize utility, 511
 - ghexfile utility (see gsrec utility)
 - ghide utility, 512
 - __ghs__, 737
 - __ghs_alignment, 742
 - __ghs_asm, 737
 - __ghs_getdbits() intrinsic function, 771
 - __ghs_getfbits() intrinsic function, 771
 - __GHS_Inline_Memory_Functions, 744
 - __GHS_Inline_String_Functions, 744
 - __ghs_max_pack_value, 742
 - __GHS_NOCOMMONS__, 742
 - ghs_noprofile attribute, 642
 - __ghs_noprofile_func() intrinsic function, 770
 - __GHS_Optimize_Debug__, 744
 - __GHS_Optimize_General__, 744
 - __GHS_Optimize_Inline, 744
 - __GHS_Optimize_Max_Debug__, 744
 - __GHS_Optimize_More_Debug__, 744
 - __GHS_Optimize_None__, 744
 - __GHS_Optimize_Size__, 744
 - __GHS_Optimize_Speed__, 744
 - __ghs_packing, 743
 - __GHS_REVISION_DATE, 737
 - __GHS_REVISION_VALUE, 737
 - __ghs_setdbits() intrinsic function, 771
 - __ghs_setfbits() intrinsic function, 771
 - __GHS_VERSION_NUMBER, 738
 - __ghsalwaysimport
 - Linker Symbols, 458

- `__ghsautoimport`
 - Linker Symbols, 458
- `__ghsbegin` symbol, 455
- `__ghsend` symbol, 455
- `.ghsinfo` program section, 463
- `-ghsmc_core_count` driver option, 301
- `-ghsmc_file` driver option, 300
- `.ghsnote` assembler directive, 463
- `__ghssize` symbol, 455
- `__ghstable_tablename`, `__ghsentry_tablename_entryname`
 - Linker generated tables, 457
- `--ghstd=2010` driver option, 253
- `--ghstd=last` driver option, 253
- `--ghstd=none` driver option, 253
- `-glimits` driver option, 296
- `.global` assembler directive, 416
- global objects, 692
- Global Registers Builder option, 71, 92
- `-globalcheck_qualifiers` driver option, 250
- `-globalcheck=full` driver option, 249
- `-globalcheck=none` driver option, 249
- `-globalcheck=normal` driver option, 249
- `-globalreg` driver option, 71, 93, 742
- `__GlobalRegisters`, 742
- `gmemfile`
 - invoking from the driver, 228
- `gmemfile` utility, 513
- `.gmon` output file type, 14
- `gnm` utility, 520, 692
- GNU attributes
 - `alias`, 637
 - `aligned`, 637
 - `always_inline`, 638
 - `at`, 638
 - `const`, 639
 - `deprecated`, 639
 - `externally_visible`, 640
 - `final`, 640
 - `format`, 640
 - `format_arg`, 642
 - `naked`, 642
 - `no_variadic_floats`, 642
 - `noinline`, 643
 - `nonnull`, 643
 - `noreturn`, 643
 - `nothrow`, 644
 - overview of, 637
 - `packed`, 644
 - `pure`, 644
 - `section`, 644
 - `sentinel`, 645
 - `transparent_union`, 650
 - `unused`, 651
 - `used`, 651
 - `warn_unused_result`, 651
 - `weak`, 652
- GNU C extensions, 589, 591
- GNU C++ extensions, 682
- GNU Compatibility Optimizations Builder option, 293
- GNU compiler
 - using with MULTI Debugger, 40, 44
- GNU Extended Assembly, 870
- GNU Options, supported, 906
- `__gnu_asm`, 738
- `--gnu_asm` driver option, 311
- `--gnu_version=30300` driver option, 309
- `--gnu_version=40300` driver option, 309
- `-gnu99` driver option, 143
- `__GNUC__`, 736
- `__GNUC_GNU_INLINE__`, 736
- `__GNUC_STDC_INLINE__`, 736
- `gpatch` utility program, 526
- `.gpc` file extension, 925
- `.gpj` file extension, 923
- `.gpj` syntax, advanced, 922
 - conditional control statements
 - `commandcommand_name`, 919
 - `comment`, 919
 - `config`, 919
 - `expressions`, 920
 - `filetypefiletype`, 919
 - `isdefined`, 920
 - `istarget`, 920
 - `isundefined`, 920
 - `nobuild`, 920
 - `optional`, 921
 - overview of, 918
 - `pass`, 921
 - `rebuild`, 921
 - `selectone_optional`, 922
 - `selectone_required`, 922
 - `streq`, 920
 - `strprefix`, 920
 - links to other projects, 916
- macro functions
 - `%expand_path(relative_path)`, 922
 - `%option_value(option_name)`, 922
- project build configurations, 915
- Top Project directives
 - customization, 918
 - `defaultBuildOptions`, 915
 - `defineConfig`, 915

- externalDepends, 916
- gbuildDir, 917
- import, 917
- macro, 917
- overview, 914
- primaryTarget, 918
- Top Project headers, 911
- Green Hills Standard Mode Builder option, 253
- grouping program variables, 62
- gs driver option, 132
- gsize driver option, 245
- gsize utility, 531
- gsrec utility
 - examples, 538, 539
 - invoking from the driver, 228
 - options, 533
 - overview, 532
- gstack
 - @filename, 555
 - annotating recursion for, 550
 - caveats, 560
 - introduction, 539
 - options, 556
 - preparing your program for, 540
 - recursive clusters, 544
 - report format, 553
 - specifying functions with non-unique names, 559
 - specifying interrupts for, 552
 - using a gstack options file with, 555
- gstrip utility, 561
- gtws driver option, 296
- Guidelines For The Use Of The C Language In Critical Systems, Motor Industry Software Reliability Association, March 2013, 723
- Guidelines For the Use Of The C Language In Critical Systems, Motor Industry Software Reliability Association, October 2004, 721
- Guidelines For the Use Of The C Language In Vehicle Based Software, Motor Industry Software Reliability Association, April 1998, 718
- Guiding Declarations of Template Functions Builder option, 218
- guiding_decls driver option, 218
- gversion utility, 563
- gwhat utility, 564

H

- H driver option, 141
- .h file extension, 924
- .H file extension, 924

- h HTML compiler option, 323
- .half assembler directive, 408
- Half-Precision Floating-Point Support Builder option, 88
- half_precision_type driver option, 88
- header file directories
 - ansi, 775
 - C, 775
 - C++, 775
 - ecxx, 775
 - eecxx, 775
 - include/arm64, 775
 - scxx, 775
- header files, 840
 - modifying searches, 58
 - printing included, 23
- headerfiledir intex option, 108
- headerfilename intex option, 110
- headers, deprecated, 698
- .heap program section, 452, 887
- help assembler option, 391
- help command line option, 526
- help driver option, 16
- help linker option, 426
- Help linker option, 426
- hex driver option, 228
- HEX386 format (see Intel hexadecimal formats)
- HEX86 format (see Intel hexadecimal formats)
- .hh file extension, 924
- HLF_DECIMAL_DIG, 741
- HLF_DIG, 741
- HLF_EPSILON, 741
- HLF_HAS_SUBNORM, 741
- HLF_MANT_DIG, 741
- HLF_MAX, 741
- HLF_MAX_10_EXP, 741
- HLF_MAX_EXP, 741
- HLF_MIN, 741
- HLF_MIN_10_EXP, 741
- HLF_MIN_EXP, 741
- HLF_TRUE_MIN, 741
- Host and Target Character Encoding Builder option, 210, 610, 693
- hosts, different building and debugging, 42
- HTML compiler options
 - c, 323
 - C, 323
 - h, 323
 - o, 322
 - p, 323
- HTML File Compression Builder option, 322

I

- I assembler option, 391
- I driver option, 57, 114
- .i file extension, 8
- .i files, 607
- I- driver option, 58
- I/O
 - mixing languages, 839
- I/O, memory-mapped, 74
- .ident assembler directive, 421
- ident driver option, 311
- Identifier Blacklist Builder option, 321
- Identifier Definition Builder option, 311
- Identifier Support Builder option, 312
- identifier_blacklist_file driver option, 321
- identifiers, 392
 - in assembler, 392
- identoutput driver option, 312
- .idep output file type, 15
- __ieeeFloat__, 740
- .if assembler directive, 407
- %if preprocessor directive, customization file, 974
- %ifdef preprocessor directive, customization file, 974
- ignore_debug_references driver option, 238
- .ii output file type, 15
- iind_stackcanary.c, ind_stackcanary2.c run-time code, 831
- illegal assumptions
 - implied register usage, 853
 - memory allocation, 854
- images for linking, converting
 - customization files, 963
- Implicit Dependency Analysis (IDA)
 - gbuild utility, 497
- Implicit Floating-Point/SIMD Register Use Builder option, 87
- Implicit Int Return Type Builder option, 257
- Implicit Use of 'std' Namespace Builder option, 213
- implicit_extern_c_type_conversion driver option, 313
- implicit_float_reg driver option, 87
- implicit_typename driver option, 219
- .import assembler directive, 416
- Import Symbols Builder option, 241
- import Top Project directive, 917
- .include assembler directive, 409
- #include directive, 115
- Include Directories Builder option, 57, 58, 114
- include driver option, 304
- Include File Search Directory Builder option, 110
- include files
 - directive for, 409
 - directory for assembler, 391
 - list of directories searched for, 115
- Include libbsp.a Builder option, 268
- %include preprocessor directive, customization file, 974
- include/arm64 header file directory, 775
- includefiledir intex option, 110
- Incorrect Pragma Directives Builder option, 256, 748
- incorrect_pragma_errors driver option, 256, 748
- incorrect_pragma_silent driver option, 256, 748
- incorrect_pragma_warnings driver option, 256
- ind_call.a64 run-time code, 830
- ind_condvar.c run-time code, 830
- ind_cov.c run-time code, 830
- ind_crt0.c startup code, 828
- ind_crt1.c run-time code, 830
- ind_dots.a64 run-time code, 830
- ind_errn.c run-time code, 831
- ind_except.c run-time code, 831
- ind_exit.c run-time code, 833
- ind_gpid.c run-time code, 831
- ind_heap.c run-time code, 832
- ind_io.c run-time code, 833
- ind_job.c run-time code, 833
- ind_lock.c run-time code, 831
- ind_manprf.c run-time code, 831
- ind_mcpy.c startup code, 829
- ind_mset.c startup code, 829
- ind_reset.c run-time code, 830
- ind_reset.c startup code, 829
- ind_stackcheck.c run-time code, 831
- ind_thrd.c run-time code, 832
- ind_thrd11.c run-time code, 832
- Indirect Function Call Checks Builder option, 95
- indirect_function_call_checks driver option, 96
- Individual Function Sections for Default Text Section Builder option, 100
- Individual Function Sections for Renamed Text Sections Builder option, 100
- Individual Function Sections for User-Defined Text Sections Builder option, 101
- Individual Link-Once Sections Builder option, 102
- Individual Variable Sections for Default Data Sections Builder option, 98
- Individual Variable Sections for Nameless Variables Builder option, 99
- Individual Variable Sections for Renamed Data Sections Builder option, 98
- Individual Variable Sections for User-Defined Data Sections Builder option, 99
- individual_attribute_data_sections driver option, 99
- individual_attribute_function_sections driver option, 101

- individual_data_sections driver option, 98
- individual_data_sections_for_nameless_vars driver option, 99
- individual_function_sections driver option, 100
- individual_linkonce_sections driver option, 102
- individual_pragma_data_sections driver option, 98
- individual_pragma_function_sections driver option, 101
- individual_section_name_extra_dot driver option, 102
- .inf output file type, 15
- %info preprocessor directive, customization file, 974
- initialized data, in ROM, 451
- Inline C Memory Functions Builder option, 287
- Inline C String Functions Builder option, 288
- Inline Constant Math Functions Builder option, 286
- Inline Larger Functions Builder option, 123
- Inline Specific Functions Builder option, 131, 331
- Inline Tiny Functions Builder option, 286
- inline_tiny_functions driver option, 286
- inline_trivial driver option, 238
- inlining
 - advantages of, 327
 - automatic two-pass, 331
 - C++, 334
 - description, 326
 - limitations of, 329
 - manual, 329
 - __noinline, 334
 - specific functions, 331
 - template function, 334
 - two-pass, 331
- inlining driver option, 308
- inlining optimizations, 326
- inlining_unless_debug driver option, 308
- Input Languages Builder option, 272
- INPUTDIR context sensitive variable, customization file, 970
- INPUTFILE context sensitive variable, customization file, 970
- .inspect assembler directive, 416
- installing
 - patches, 526
- Instantiate Extern Inline Builder option, 214, 670
- instantiate_extern_inline driver option, 214
- Instrument Prologue/Epilogue Builder option, 96
- instrument_prologue_epilogue driver option, 96
- .int
 - file extension, 925
- int data type
 - size, 739, 746
- __INT_BIT, 739
- __Int_Is_32, 746
- __Int_Is_64, 746
- __INTEGRITY, 746
- INTEGRITY Application Stripping Builder option, 111
- INTEGRITY Posix Mode Builder option, 108
- INTEGRITY TimeMachine Builder option, 132
- INTEGRITY TimeMachine Enabled Builder option, 297
- integrity_posix driver option, 108
- :integrity_posix_mode=noposix driver option, 108
- :integrity_posix_mode=single driver option, 108
- :integrity_posix_mode=system driver option, 108
- Intel hexadecimal formats
 - generating, 228, 532
- Interleaved Source and Assembly Builder option, 225
- Intermediate Output Directory Relative to This File Builder option, 273
- Intermediate Output Directory Relative to Top-Level Project Builder option, 117
- Intermodule Inlining Builder option, 121, 331
- interprocedural optimizations, 335
 - using gbuild, 339
 - whole program optimizations, 338
- Interprocedural Optimizations Builder option, 122, 338, 339
- interrupt
 - routine and optimizations, 598
- intex options
 - display_boot, 109
 - dynamic, 108
 - generatemonolithdeletions, 109
 - headerfiledir, 108
 - headerfilename, 110
 - includefiledir, 110
 - intex_var, 111
 - intexoption, 110
 - memory, 109
 - memory=, 109
 - no_display_boot, 109
 - no_dynamic, 109
 - no_quit_after_intex_warnings, 111
 - no_strip_application, 111
 - quit_after_intex_warnings, 111
 - sha1, 112
 - strip_application, 111
- Intex Variable Definition Builder option, 111
- intex_var intex option, 111
- intexoption intex option, 110
- intrinsic functions
 - __ADD_WITH_CARRY64(), 763
 - __builtin_constant_p(), 770
 - __builtin_frame_address(), 762
 - __builtin_return_address(), 762
 - __CLZ32(), 763

__CLZ64(), 763
 __DI(), 763
 __DIR(), 763
 __DMB_OPT(), 768
 __DSP_OPT(), 768
 __EI(), 763
 __EIR(), 763
 __force_stores(), 767
 __ghs_getdbits(), 771
 __ghs_getfbits(), 771
 __ghs_noprofile_func(), 770
 __ghs_setdbits(), 771
 __ghs_setfbits(), 771
 __ISB(), 767
 __LDXR16(), 764
 __LDXR32(), 764
 __LDXR64(), 764
 __LDXR8(), 764
 __memory_changed(), 767
 __MRS(), 769
 __MSR(), 769
 __MSR_IMM(), 769
 __MULSH(), 763
 __MULSH64(), 764
 __MULUH(), 763
 __MULUH64(), 764
 __NOP(), 768
 __REV16(), 766
 __REV32(), 766
 __REV64(), 766
 __REVSH(), 766
 __RIR(), 763
 __schedule_barrier(), 767
 __SEV(), 768
 __STXR16(), 764
 __STXR32(), 764
 __STXR64(), 764
 __STXR8(), 764
 __SYS(), 770
 __SYS_XZR(), 770
 TYP __builtin_speculation_safe_load, 771
 __WFE(), 768
 __WFI(), 769
 __YIELD(), 769
 intrinsics, 762
 __get_stack_pointer(void), 762
 IP Alias Lib Functions Builder option, 130
 IP Alias Reads Builder option, 129
 IP Alias Writes Builder option, 129
 IP Allocate Variables Builder option, 130
 IP Constant Globals Builder option, 127

IP Constant Propagation Builder option, 128
 IP Constant Returns Builder option, 129
 IP Delete Functions Builder option, 126
 IP Delete Globals Builder option, 126
 IP Limit Inlining Builder option, 128
 IP Localize Globals Builder option, 127
 IP One-Site Inlining Builder option, 126
 IP Remove Parameters Builder option, 128
 IP Remove Returns Builder option, 129
 IP Small Inlining Builder option, 128
 .ipf output file type, 15
 .irc file extension, 925
 -irel driver option, 268
 isalnum(), C implementation, 628
 isalpha(), C implementation, 629
 __ISB() intrinsic function, 767
 iscntrl(), C implementation, 629
 isdefined .gpj conditional control statement, 920
 isdefined linker expression function, 442
 isdigit(), C implementation, 629
 islower(), C implementation, 629
 ISO/IEC 14882:2003, 669
 isprint(), C implementation, 629
 istarget .gpj conditional control statement, 920
 isundefined .gpj conditional control statement, 920
 isupper(), C implementation, 629
 isweak linker expression function, 442
 ITEMID context sensitive variable, customization file, 971

J

Japanese Automotive C, 595
 builder and driver options for, 595, 596
 -japanese_automotive_c driver option, 143, 595, 596
 __Japanese_Automotive_C__, 736

K

Kanji, 610
 -kanji driver option, 212
 -keep linker option, 426
 -keep_from linker option, 426
 --keep_static_symbols driver option, 209
 -keepasmfiles driver option, 271
 -keepcppfiles driver option, 271
 -keepcppfileswithlines driver option, 272
 -keepmap driver option, 243
 KEEPOBJECTS context sensitive variable, customization file, 971
 -keeptempfiles driver option, 270
 -kernel driver option, 107
 Kernel Project Builder option, 107

-kernel=kernel_checked driver option, 107
-kernel=kernel_opt driver option, 107
-kernel=kernel_vel driver option, 107
-kernel=kernel_vel_checked driver option, 107
-kernel=kernel_vel_opt driver option, 107
-key command line option, 527
Koenig lookup (see argument-dependent lookup)

L

-L driver option, 59, 115
-l driver option, 59, 116
labels
 assembly, 395, 401
-language driver option, 9, 272, 776
 __LANGUAGE_ASM__, 736
 __LANGUAGE_C__, 736
 __LANGUAGE_CXX__, 736, 737
language-independent library, 823
-large_archive driver option, 319
-large_memory_model driver option, 82
-layout=ram
 locating program sections in ROM and RAM, 69
-layout=romcopy
 locating program sections in ROM and RAM, 70, 828
-layout=romrun
 locating program sections in ROM and RAM, 70
.lcomm assembler directive, 416
.lcp output file type, 15
.ld file extension, 9, 925
 (see also linker directives files (.ld))
 __LDXR16() intrinsic function, 764
 __LDXR32() intrinsic function, 764
 __LDXR64() intrinsic function, 764
 __LDXR8() intrinsic function, 764
less buffered I/O, 779
lib8bit library, 778
libansi library, 778
libarch library, 778
-libbsp driver option, 268
libece library, 777
libecnoe library, 777
libedge library, 777
libedgnoe library, 777
libeece library, 777
libeecnoe library, 777
libind library, 778
libind.a
 functions, 823
 language-independent library, 823
libmath library, 778

libmath.a
 functions, 811, 818, 822, 823
libmulti library, 779
libmulti.a file, 45
librarian
 command modifiers
 c, 468
 C, 468
 D, 468
 e, 468
 m, 469
 M, 469
 n, 469
 o, 469
 s, 469
 S, 469
 u, 469
 v, 469
 commands
 -d, 467
 -p, 468
 -r, 468
 -t, 468
 -x, 468
 examples, 470
 introduction to, xxxi
 options
 -display_toc, 469
 -pecoff, 470
 -stderr=errfile, 470
 -tmp=dir, 470
 overview, 466
 syntax, 467
 table of contents, creating and updating, 471
libraries
 creating, 466
 creating and using, examples, 470
 deleting files from, 467
 EC++, 737
 incorporating changes, 825
 initialization, 838
 lib8bit, 778
 libansi, 778
 libarch, 778
 libece, 777
 libecnoe, 777
 libedge, 777
 libedgnoe, 777
 libeece, 777
 libeecnoe, 777
 libind, 778

- libind.a, 753
- libmath, 778
- libmulti, 779
- libsce, 777
- libsnoe, 777
- libsedge, 777
- libsedgnoe, 777
- libstartup, 778
- libsys, 778
- libutf8, 778
- libwc, 778
- libwchar, 778
- merging, 466
- searching, 115
- Libraries Builder option, 59, 116
- library directories, 776
- Library Directories Builder option, 59, 115
- libsce library, 777
- libsnoe library, 777
- libsedge library, 777
- libsedgnoe library, 777
- libstartup library, 778
- libstartup.a library, 451, 828
- libstartup.a
 - functions, 823
- libsys library, 778
- libsys.a library, 451, 829
- libsys.a
 - system calls, 809
- libutf8 library, 778
- libwc library, 778
- libwchar library, 778
- limitations
 - alignment and size, 846
 - C compiler, 611
 - memory optimization, 854
- <limits.h> header file, 844
- Line Numbers in Temporary Preprocessor Output Builder option, 272
- line terminators
 - in assembler source statements, 396
- Line Wrap Messages Builder option, 254
- __LINE__, 735
- .linfix program section, 887
- Link in Minimum Libraries Builder option, 315
- Link in Standard Libraries Builder option, 315
- Link Mode Builder option, 316
- link_allow_duplicate_ghs_syms driver option, 234
- link_allow_duplicate_linkonce_syms driver option, 235
- link_constprop driver option, 237
- link_filter driver option, 312
- link-once templates, 685
- link_output_mode_linksafe driver option, 317
- link_output_mode_reuse driver option, 316
- link_output_mode_safe driver option, 316
- link_output_mode_unlink driver option, 317
- link_reject_duplicate_lib_syms driver option, 234
- Link-Time Argument Type Checking Builder option, 246
- Link-Time Constant Propagation Builder option, 237
- Link-Time Global Variable Type Checking Builder option, 249
- Link-Time Global Variable Type Qualifiers Builder option, 250
- Link-Time Ignore Debug References Builder option, 238
- Link-Time Trivial Function Inlining Builder option, 238
- linkage, 691
- linker
 - advanced features, 460
 - compressed ROM, 446, 449
 - ELF optional header output, 893
 - introduction to, xxxi
 - invoking with the driver, 59
 - memory maps, 432
 - (see also linker directives files (.ld))
 - overview, 424
 - section maps, 434
- Linker Command File Builder option, 233
- Linker Directive Files with Non-standard Extensions Builder option, 231
- Linker Directives Directory Builder option, 314
- linker directives files, 60
- linker directives files (.ld)
 - associating with a Project Manager project, 60
 - Builder and driver options for, 231, 232, 233, 314
 - customizing run-time sections, 451
 - defaults
 - standalone_ram.ld, 69
 - standalone_romcopy.ld, 70
 - standalone_romrun.ld, 70
 - DEFAULTS directive, 431
 - directory, specifying, 314
 - locating program sections in ROM and RAM, 69
 - maximizing speed, 451
 - MEMORY directive, 432
 - memory maps, 432
 - minimizing RAM usage, 451
 - OPTION directive, 430
 - overview, 429
 - ROM and CROM, 449
 - section attributes, 443
 - section maps, 434
 - SECTIONS directive, 434

- sections, including and renaming, 437
- sections, reserved memory area, 433
- specifying options after, 231, 232
- specifying options before, 233
- using, 10
- linker expression functions
 - absolute, 442
 - addr, 442
 - align, 442
 - alignof, 442
 - endaddr, 442
 - error, 442
 - final, 442
 - isdefined, 442
 - isweak, 442
 - max, 443
 - memaddr, 442
 - min, 443
 - pack_or_align, 443
 - provide, 443
 - sizeof, 443
- Linker generated tables
 - __ghstable_tablename, __ghsentry_tablename_entryname, 457
- linker
 - program entry point, 429
- Linker Optimizations Builder option, 122
- linker options
 - allabsolute, 425
 - allow_ambiguous_abs, 425
 - allow_bad_relocations, 425
 - ambiguous_absolute_error, 425
 - any_p_align, 425
 - append, 425
 - D, 426
 - earlyabsolute, 426
 - extractall, 426, 458
 - extractweak, 426, 458
 - help, 426
 - Help, 426
 - keep, 426
 - keep_from, 426
 - mapfile_type, 426
 - max_p_align=n, 425
 - min_p_align=n, 425
 - no_append, 426
 - no_crom, 427, 450
 - no_map_renaming, 427
 - no_p_align, 425
 - no_uncompressed_copy, 427
 - old_section_wildcard_matching, 427
 - only_data_absolute, 427
 - prog2, 427
 - retainlocals, 427
 - T, 427
 - unalign_debug_sections, 428
 - underscore, 428
 - unweak, 428
 - v, 428
 - V, 428
 - vimport, 428
 - w, 428
 - wrapsym, 428
- linker raw files, 924
- linker section attributes
 - ABS, 444
 - ALIGN, 444
 - AT, 444
 - CLEAR, 444
 - CROM, 446, 449
 - FILL, 444
 - LOAD, 444
 - MAX_ENDADDRESS, 444
 - MAX_SIZE, 444
 - MIN_ENDADDRESS, 444
 - MIN_SIZE, 445
 - MONOTONIC, 445
 - NOCHECKSUM, 445
 - NOCLEAR, 444
 - NOLOAD, 445
 - NOMONOTONIC, 445
 - PAD, 445
 - RANDOMIZE, 445
 - ROM, 446, 449
 - ROM_NOCOPY, 446
 - SHFLAGS, 446
- Linker Symbols
 - __ghsalwaysimport, 458
 - __ghsautoimport, 458
- Linker Warnings Builder option, 230
- Linker-Based Far Call Patching Builder option, 91
- linker_link driver option, 312
- linker_warnings driver option, 230
- linking projects, 916
- Linking Sections with Different Types Builder option, 317
- __linkonce keyword, 582, 685
- linkonce keyword, 582, 685
- .list assembler directive, 415
- list assembler option, 392
- list command line option, 527
- list driver option, 59, 224
- list_dir driver option, 224

--list_macros driver option, 13
listing format directives, 415
little endian mode, 26
__LITTLE_ENDIAN__, 738
-littleendian driver option, 79
__littleendian keyword, 598
__LL_BIT, 746
__LL_Is_64, 746
__LLONG_BIT, 739
-lmulti driver option, 45
-lnk driver option, 232
.lnk file extension, 925
-lnk0 driver option, 231
-lnkcmd driver option, 233
LOAD linker section attribute, 444
local scalar variables, 289
-locals_unchanged_by_longjmp driver option, 311
-locatedprogram driver option, 227
long data type
 size, 739, 746
long long data type, 206
 size, 739, 746
Long Long Support Builder option, 206
__LONG_BIT, 739
__Long_Is_32, 746
__Long_Is_64, 746
--long_long driver option, 206
longjmp() Does Not Restore Local Vars Builder option, 310
loop invariant analysis optimization, 343
loop merging optimization, 344
loop motion optimization, 343
loop optimizations, 340
 automatic vector optimization, 346
 manual vectorization, 346
 vectorization, 346
Loop Optimize Specific Functions Builder option, 131, 342
Loop Unrolling Builder option, 289, 345
loop unrolling optimization, 344
.lsbss assembler directive, 417
.lst output file type, 15

M

-Ma driver option, 243
machine-specific arithmetic, 852
macro
 definition directives, 409
macro assembler, 390
.macro assembler directive, 412
:macro driver option, 284
macro functions, for .gpj files, 922

macro Top Project directive, 917
.macrolocal assembler directive, 412
macros
 defining, 409
__main(), 839, 841, 842
main()
 mixing languages, 839
MAKE_DEPEND_OPTIONS context sensitive variable,
 customization file, 971
--make_depends driver option, 24
makefiles
 and custom run-time environments, 826
 dependency information, 23
 generating, 530
 incremental builds with, 19
 individual compiler passes, 21
 interprocedural optimizations with, 21
 macros for, 19
 multiple languages and, 20
 using, 18
Malloc Version Builder option, 102
malloc()
 errors related to, 51
 memory allocation, 782
malloc(), C implementation, 626
-malloc_num_arenas driver option, 105
malloc_size DoubleCheck property type, 363
-malloc_version=arena driver option, 103
-malloc_version=legacy driver option, 102
-malloc_version=memcheck driver option, 103
-malloc_version=realtime driver option, 103
-malloc_version=smallblock driver option, 103
malloced DoubleCheck property type, 363
-Man driver option, 243
 (see also -Mna)
manual inlining, 329
-map driver option, 243
Map File Cross-Referencing Builder option, 244
Map File Generation Builder option, 243
Map File Page Length Builder option, 243
Map File Retention Builder option, 243
Map File Sorting Builder option, 243
.map output file type, 15
-map_eofn_symbols driver option, 245
-mapfile_type linker option, 426
-maplines driver option, 243
mapping host paths, 42
math.h
 functions, 811
Max Depth for Recursive Template Instantiations Builder
 option, 221

- max linker expression function, 443
- MAX_ENDADDRESS linker section attribute, 444
- max_inlining driver option, 308
- max_inlining_unless_debug driver option, 308
- max_p_align=n linker option, 425
- MAX_SIZE linker section attribute, 444
- Maximize Optimizations Builder option, 124
- Maximum Number of Errors to Display Builder option, 251
- .maxinstances assembler directive, 419
- .maxstack assembler directive, 420
- .mbs file extension, 924
- MD driver option, 24
- .mem output file type, 16
- mem_funcs=fast driver option, 79
- mem_funcs=safe driver option, 80
- memaddr linker expression function, 442
- memcheck_num_callers driver option, 139
- memory, 26
 - alignment, 601
 - defining, 432
 - leaks, finding, 52
 - optimization
 - restrictions, 854
 - problems with size, 856
- Memory Access Builder option, 205
- memory allocation
 - debugging, 51
 - disabling, 54
 - enabling, 54
 - error checking, 52, 53, 54
 - malloc(), 782
- Memory Allocations window
 - enabling error-checking from, 54
- memory driver option, 228
- Memory Functions Builder option, 79
- memory_intex option, 109
- MEMORY linker directive, 432
- memory maps, 432
 - (see also linker directives files (.ld))
- Memory Model Builder option, 81
- memory optimization, 354
- Memory Optimization Builder option, 292, 854, 855
- memory_access=default driver option, 205
- memory_access=efficient driver option, 205
- memory_access=safe driver option, 205
- __memory_changed() intrinsic function, 767
- memory-mapped I/O, 74
- memory= intex option, 109
- merge_archive driver option, 13, 466
- Message Pragma Builder option, 305
- MF driver option, 24
- min linker expression function, 443
- MIN_ENDADDRESS linker section attribute, 444
- min_p_align=n linker option, 425
- MIN_SIZE linker section attribute, 445
- minlib driver option, 316
- misalign_access driver option, 208
- misaligned loads and stores, handling, 34
- Misaligned Memory Access Builder option, 208
- MISRA C
 - builder and driver options for, 148, 150, 151, 152, 153, 154, 155, 156, 157, 158
 - #pragma directives, 759
 - predefined macros for, 741
 - switch statements, 726
- MISRA C - Advisory Rules Level Builder option, 148
- MISRA C - Enforcement and Run-Time Builder option, 148
- MISRA C - Mandatory Rules Level Builder option, 147
- MISRA C - Required Rules Level Builder option, 148
- MISRA C 1998, 718
 - builder and driver options for, 149
 - introduction to, 149
- MISRA C 1998 Rules - Character Set Builder option, 150
- MISRA C 1998 Rules - Comments Builder option, 150
- MISRA C 1998 Rules - Constants Builder option, 151
- MISRA C 1998 Rules - Control Flow Builder option, 154
- MISRA C 1998 Rules - Conversions Builder option, 153
- MISRA C 1998 Rules - Declarations and Definitions Builder option, 152
- MISRA C 1998 Rules - Environment Builder option, 149
- MISRA C 1998 Rules - Expressions Builder option, 154
- MISRA C 1998 Rules - Functions Builder option, 155
- MISRA C 1998 Rules - Identifiers Builder option, 151
- MISRA C 1998 Rules - Initialization Builder option, 152
- MISRA C 1998 Rules - Operators Builder option, 153
- MISRA C 1998 Rules - Pointers and Arrays Builder option, 157
- MISRA C 1998 Rules - Preprocessor Builder option, 156
- MISRA C 1998 Rules - Standard Library Builder option, 158
- MISRA C 1998 Rules - Structs and Unions Builder option, 157
- MISRA C 1998 Rules - Types Builder option, 151
- MISRA C 2004
 - builder and driver options for, 721
 - introduction to, 721
- MISRA C 2004 Rules - 1 Environment Builder option, 159
- MISRA C 2004 Rules - 10 Arithmetic Type Conversions Builder option, 164
- MISRA C 2004 Rules - 11 Pointer Type Conversions Builder option, 165
- MISRA C 2004 Rules - 12 Expressions Builder option, 166

- MISRA C 2004 Rules - 13 Control Statement Expressions Builder option, 167
- MISRA C 2004 Rules - 14 Control Flow Builder option, 168
- MISRA C 2004 Rules - 15 Switch Statements Builder option, 169
- MISRA C 2004 Rules - 16 Functions Builder option, 170
- MISRA C 2004 Rules - 17 Pointers and Arrays Builder option, 171
- MISRA C 2004 Rules - 18 Structs and Unions Builder option, 171
- MISRA C 2004 Rules - 19 Preprocessing Directives Builder option, 172
- MISRA C 2004 Rules - 2 Language Extensions Builder option, 160
- MISRA C 2004 Rules - 20 Standard Libraries Builder option, 174
- MISRA C 2004 Rules - 21 Run-time Failures Builder option, 175
- MISRA C 2004 Rules - 3 Documentation Builder option, 160
- MISRA C 2004 Rules - 4 Character Sets Builder option, 161
- MISRA C 2004 Rules - 5 Identifiers Builder option, 161
- MISRA C 2004 Rules - 6 Types Builder option, 162
- MISRA C 2004 Rules - 7 Constants Builder option, 162
- MISRA C 2004 Rules - 8 Declarations and Definitions Builder option, 163
- MISRA C 2004 Rules - 9 Initialization Builder option, 164
- MISRA C 2012
 - builder and driver options for, 723
 - introduction to, 723
- MISRA C 2012 Guidelines - 1 Environment Builder option, 177
- MISRA C 2012 Guidelines - 10 The Essential Type Model Builder option, 185
- MISRA C 2012 Guidelines - 11 Pointer Type Conversions Builder option, 186
- MISRA C 2012 Guidelines - 12 Expressions Builder option, 187
- MISRA C 2012 Guidelines - 13 Side Effects Builder option, 188
- MISRA C 2012 Guidelines - 14 Control Statement Expressions Builder option, 189
- MISRA C 2012 Guidelines - 15 Control Flow Builder option, 190
- MISRA C 2012 Guidelines - 16 Switch Statements Builder option, 191
- MISRA C 2012 Guidelines - 17 Functions Builder option, 192
- MISRA C 2012 Guidelines - 18 Pointers and Arrays Builder option, 194
- MISRA C 2012 Guidelines - 19 Overlapping Storage Builder option, 195
- MISRA C 2012 Guidelines - 2 Unused Code Builder option, 178
- MISRA C 2012 Guidelines - 20 Preprocessing Directives Builder option, 195
- MISRA C 2012 Guidelines - 21 Standard Libraries Builder option, 197
- MISRA C 2012 Guidelines - 22 Resources Builder option, 200
- MISRA C 2012 Guidelines - 3 Comments Builder option, 179
- MISRA C 2012 Guidelines - 4 Character Sets and Lexical Conventions Builder option, 179
- MISRA C 2012 Guidelines - 5 Identifiers Builder option, 180
- MISRA C 2012 Guidelines - 6 Types Builder option, 181
- MISRA C 2012 Guidelines - 7 Literals and Constants Builder option, 181
- MISRA C 2012 Guidelines - 8 Declarations and Definitions Builder option, 182
- MISRA C 2012 Guidelines - 9 Initialization Builder option, 184
- MISRA C 2012 Guidelines - Directives Builder option, 201
- MISRA C 2012 Guidelines - Stricter R8.7 Builder option, 203
- MISRA C 2012 Guidelines - Stricter R8.9 Builder option, 203
- misra_2004=1.1 driver option, 159
- misra_2004=1.2 driver option, 159
- misra_2004=1.3 driver option, 159
- misra_2004=1.4 driver option, 159
- misra_2004=1.5 driver option, 159
- misra_2004=10.1 driver option, 164
- misra_2004=10.2 driver option, 164
- misra_2004=10.3 driver option, 165
- misra_2004=10.4 driver option, 165
- misra_2004=10.5 driver option, 165
- misra_2004=10.6 driver option, 165
- misra_2004=11.1 driver option, 165
- misra_2004=11.2 driver option, 165
- misra_2004=11.3 driver option, 165
- misra_2004=11.4 driver option, 165
- misra_2004=11.5 driver option, 165
- misra_2004=12.1 driver option, 166
- misra_2004=12.10 driver option, 167
- misra_2004=12.11 driver option, 167
- misra_2004=12.12 driver option, 167
- misra_2004=12.13 driver option, 167
- misra_2004=12.2 driver option, 166
- misra_2004=12.3 driver option, 166

--misra_2004=7.1 driver option, 162	--misra_2012=R11.6 driver option, 187
--misra_2004=8.1 driver option, 163	--misra_2012=R11.7 driver option, 187
--misra_2004=8.10 driver option, 163	--misra_2012=R11.8 driver option, 187
--misra_2004=8.11 driver option, 164	--misra_2012=R11.9 driver option, 187
--misra_2004=8.12 driver option, 164	--misra_2012=R12.1 driver option, 188
--misra_2004=8.2 driver option, 163	--misra_2012=R12.2 driver option, 188
--misra_2004=8.3 driver option, 163	--misra_2012=R12.3 driver option, 188
--misra_2004=8.4 driver option, 163	--misra_2012=R12.4 driver option, 188
--misra_2004=8.5 driver option, 163	--misra_2012=R12.5 driver option, 188
--misra_2004=8.6 driver option, 163	--misra_2012=R13.1 driver option, 188
--misra_2004=8.7 driver option, 163	--misra_2012=R13.2 driver option, 189
--misra_2004=8.8 driver option, 163	--misra_2012=R13.3 driver option, 189
--misra_2004=8.9 driver option, 163	--misra_2012=R13.4 driver option, 189
--misra_2004=9.1 driver option, 164	--misra_2012=R13.5 driver option, 189
--misra_2004=9.2 driver option, 164	--misra_2012=R13.6 driver option, 189
--misra_2004=9.3 driver option, 164	--misra_2012=R14.1 driver option, 190
--misra_2012_stricter_R8_7 driver option, 203	--misra_2012=R14.2 driver option, 190
--misra_2012_stricter_R8_9 driver option, 204	--misra_2012=R14.3 driver option, 190
--misra_2012=D1.1 driver option, 201	--misra_2012=R14.4 driver option, 190
--misra_2012=D2.1 driver option, 201	--misra_2012=R15.1 driver option, 190
--misra_2012=D3.1 driver option, 201	--misra_2012=R15.2 driver option, 190
--misra_2012=D4.1 driver option, 202	--misra_2012=R15.3 driver option, 191
--misra_2012=D4.10 driver option, 203	--misra_2012=R15.4 driver option, 191
--misra_2012=D4.11 driver option, 203	--misra_2012=R15.5 driver option, 191
--misra_2012=D4.12 driver option, 203	--misra_2012=R15.6 driver option, 191
--misra_2012=D4.13 driver option, 203	--misra_2012=R15.7 driver option, 191
--misra_2012=D4.14 driver option, 203	--misra_2012=R16.1 driver option, 191
--misra_2012=D4.2 driver option, 202	--misra_2012=R16.2 driver option, 192
--misra_2012=D4.3 driver option, 202	--misra_2012=R16.3 driver option, 192
--misra_2012=D4.4 driver option, 202	--misra_2012=R16.4 driver option, 192
--misra_2012=D4.5 driver option, 202	--misra_2012=R16.5 driver option, 192
--misra_2012=D4.6 driver option, 202	--misra_2012=R16.6 driver option, 192
--misra_2012=D4.7 driver option, 202	--misra_2012=R16.7 driver option, 192
--misra_2012=D4.8 driver option, 202	--misra_2012=R17.1 driver option, 192
--misra_2012=D4.9 driver option, 202	--misra_2012=R17.2 driver option, 193
--misra_2012=R1.1 driver option, 177	--misra_2012=R17.3 driver option, 193
--misra_2012=R1.2 driver option, 177	--misra_2012=R17.4 driver option, 193
--misra_2012=R1.3 driver option, 177	--misra_2012=R17.5 driver option, 193
--misra_2012=R10.1 driver option, 185	--misra_2012=R17.6 driver option, 193
--misra_2012=R10.2 driver option, 185	--misra_2012=R17.7 driver option, 193
--misra_2012=R10.3 driver option, 185	--misra_2012=R17.8 driver option, 194
--misra_2012=R10.4 driver option, 185	--misra_2012=R18.1 driver option, 194
--misra_2012=R10.5 driver option, 185	--misra_2012=R18.2 driver option, 194
--misra_2012=R10.6 driver option, 186	--misra_2012=R18.3 driver option, 194
--misra_2012=R10.7 driver option, 186	--misra_2012=R18.4 driver option, 194
--misra_2012=R10.8 driver option, 186	--misra_2012=R18.5 driver option, 194
--misra_2012=R11.1 driver option, 186	--misra_2012=R18.6 driver option, 194
--misra_2012=R11.2 driver option, 186	--misra_2012=R18.7 driver option, 195
--misra_2012=R11.3 driver option, 187	--misra_2012=R18.8 driver option, 195
--misra_2012=R11.4 driver option, 187	--misra_2012=R19.1 driver option, 195
--misra_2012=R11.5 driver option, 187	--misra_2012=R19.2 driver option, 195

--misra_2012=R2.1 driver option, 178
--misra_2012=R2.2 driver option, 178
--misra_2012=R2.3 driver option, 178
--misra_2012=R2.4 driver option, 178
--misra_2012=R2.5 driver option, 178
--misra_2012=R2.6 driver option, 178
--misra_2012=R2.7 driver option, 179
--misra_2012=R20.1 driver option, 195
--misra_2012=R20.10 driver option, 196
--misra_2012=R20.11 driver option, 197
--misra_2012=R20.12 driver option, 197
--misra_2012=R20.13 driver option, 197
--misra_2012=R20.14 driver option, 197
--misra_2012=R20.2 driver option, 195
--misra_2012=R20.3 driver option, 196
--misra_2012=R20.4 driver option, 196
--misra_2012=R20.5 driver option, 196
--misra_2012=R20.6 driver option, 196
--misra_2012=R20.7 driver option, 196
--misra_2012=R20.8 driver option, 196
--misra_2012=R20.9 driver option, 196
--misra_2012=R21.1 driver option, 197
--misra_2012=R21.10 driver option, 198
--misra_2012=R21.11 driver option, 198
--misra_2012=R21.12 driver option, 198
--misra_2012=R21.13 driver option, 199
--misra_2012=R21.14 driver option, 199
--misra_2012=R21.15 driver option, 199
--misra_2012=R21.16 driver option, 199
--misra_2012=R21.17 driver option, 199
--misra_2012=R21.18 driver option, 199
--misra_2012=R21.19 driver option, 199
--misra_2012=R21.2 driver option, 197
--misra_2012=R21.20 driver option, 199
--misra_2012=R21.3 driver option, 197
--misra_2012=R21.4 driver option, 198
--misra_2012=R21.5 driver option, 198
--misra_2012=R21.6 driver option, 198
--misra_2012=R21.7 driver option, 198
--misra_2012=R21.8 driver option, 198
--misra_2012=R21.9 driver option, 198
--misra_2012=R22.1 driver option, 200
--misra_2012=R22.10 driver option, 201
--misra_2012=R22.2 driver option, 200
--misra_2012=R22.3 driver option, 200
--misra_2012=R22.4 driver option, 200
--misra_2012=R22.5 driver option, 201
--misra_2012=R22.6 driver option, 201
--misra_2012=R22.7 driver option, 201
--misra_2012=R22.8 driver option, 201
--misra_2012=R22.9 driver option, 201

--misra_2012=R3.1 driver option, 179
--misra_2012=R3.2 driver option, 179
--misra_2012=R4.1 driver option, 179
--misra_2012=R4.2 driver option, 179
--misra_2012=R5.1 driver option, 180
--misra_2012=R5.2 driver option, 180
--misra_2012=R5.3 driver option, 180
--misra_2012=R5.4 driver option, 180
--misra_2012=R5.5 driver option, 180
--misra_2012=R5.6 driver option, 180
--misra_2012=R5.7 driver option, 180
--misra_2012=R5.8 driver option, 180
--misra_2012=R5.9 driver option, 181
--misra_2012=R6.1 driver option, 181
--misra_2012=R6.2 driver option, 181
--misra_2012=R7.1 driver option, 181
--misra_2012=R7.2 driver option, 181
--misra_2012=R7.3 driver option, 181
--misra_2012=R7.4 driver option, 182
--misra_2012=R8.1 driver option, 182
--misra_2012=R8.10 driver option, 183
--misra_2012=R8.11 driver option, 183
--misra_2012=R8.12 driver option, 183
--misra_2012=R8.13 driver option, 183
--misra_2012=R8.14 driver option, 184
--misra_2012=R8.2 driver option, 182
--misra_2012=R8.3 driver option, 182
--misra_2012=R8.4 driver option, 182
--misra_2012=R8.5 driver option, 182
--misra_2012=R8.6 driver option, 183
--misra_2012=R8.7 driver option, 183
--misra_2012=R8.8 driver option, 183
--misra_2012=R8.9 driver option, 183
--misra_2012=R9.1 driver option, 184
--misra_2012=R9.2 driver option, 184
--misra_2012=R9.3 driver option, 184
--misra_2012=R9.4 driver option, 184
--misra_2012=R9.5 driver option, 185
--misra_adv=error driver option, 148
--misra_adv=silent driver option, 148
--misra_adv=warn driver option, 148
__MISRA_i, 742
--misra_mand=error driver option, 147
--misra_mand=silent driver option, 147
--misra_mand=warn driver option, 147
--misra_req=error driver option, 148
--misra_req=silent driver option, 148
--misra_req=warn driver option, 148
--misra_runtime driver option, 148
mixed language executable, 838
-Ml driver option, 245

- MMD driver option, 24
- Mn driver option, 243
- Mna driver option, 243
 - (see also -Man)
- .mon output file type, 14
- Monotonic Block Order Builder option, 318
- MONOTONIC linker section attribute, 445
- monotonic_block_order driver option, 318
- __MRS() intrinsic function, 769
- Ms driver option, 244
- __MSR() intrinsic function, 769
- __MSR_IMM() intrinsic function, 769
- __msw, 746
- Mu driver option, 244
- __MULSH() intrinsic function, 763
- __MULSH64() intrinsic function, 764
- MULTI
 - document set, xxvi
- MULTI Debugging Level, 45
- MULTI for Eclipse
 - building projects, 986
 - compiler options, changing, 985
 - connecting with setup scripts, 987
 - creating projects in, 981
 - debugging projects, 988
 - installing, 980
 - INTEGRITY projects in, 982
 - overview, 978
 - prerequisites for installing, 979
 - running projects, 986
 - source files, adding to project, 984
- Multi-Core Configuration File Builder option, 300
- Multi-Core Core Count Builder option, 301
- multibyte characters, 609
- multiple driver option, 240
- multiple-section programs, 62
- Multiply-Defined Symbols Builder option, 240
- __MULUH() intrinsic function, 763
- __MULUH64() intrinsic function, 764
- munch driver option, 312
- mute_eagle_signal driver option, 295
- mute_entry_exit_history driver option, 133
- Mx driver option, 244

N

- naked GNU attribute, 642
- name lookup
 - template instantiation, 688
- name mangling, 691, 692
- named labels, 401

- namespace support
 - argument dependent lookup, 690
 - dependent name lookup, 688
 - lookup using the reference context, 689
- Namespace Support Builder option, 213, 688
- namespaces, 688
 - predefined macro for, 745
 - std, 745
 - template instantiation, and, 688
 - __NAMESPACES, 745
- namespaces driver option, 213
- NaN Compares as Unordered Builder option, 86
- narrowing_conversions driver option, 223
- near and far function calls, 72
- near function calls, 72
- __nearcall, 73
- .need assembler directive, 421
- needs_null_check DoubleCheck property type, 363
- NEON coprocessor, 26, 84
- new
 - array, predefined macro, 745
- New-Style Cast Support Builder option, 215
- new_style_casts driver option, 215
- nm utility, 692
- no_additional_output driver option, 228
- no_align_standalone_objects driver option, 294
- no_allow_cxx_duplicates driver option, 319
- no_allow_different_section_types driver option, 317
- no_allow_linkonce_duplicates driver option, 320
- no_alternative_tokens driver option, 210
- no_annotate_call_return driver option, 81
- no_ansi_alias driver option, 146
- no_any_output_suffix driver option, 273
- no_append linker option, 426
- no_arm64_errata_1183550 driver option, 82
- no_assembler_warnings driver option, 314
- no_auto_type driver option, 222
- no_bool driver option, 213
- no_brief_diagnostics driver option, 254
- no_c_and_cpp_functions_are_distinct driver option, 313
- no_check_straightline_spectre_mitigation driver option, 83
- no_codefactor driver option, 237
- no_comments driver option, 305
- no_commons driver option, 62, 65, 67, 91
- no_crom linker option, 427, 450
- no_data_delete driver option, 236
- no_debug driver option, 132
- no_defer_parse_function_templates driver option, 220
- no_delete driver option, 235
- no_dep_name driver option, 219

--no_deprecated_nonconst_string_literal_conv driver option, 223
--no_diagnostic_header_trace driver option, 255
--no_discard_zero_initializers driver option, 92
--no_display_boot_intex option, 109
--no_display_diag_number driver option, 261
--no_dual_debug driver option, 303
--no_dwarf2dbo driver option, 304
--no_dynamic_intex option, 109
--no_error_basename driver option, 254
--no_exceptions driver option, 146
--no_extend_liveness driver option, 297
--no_float_associativity driver option, 86
--no_float_scanf driver option, 314
--no_full_breakdots driver option, 297
--no_full_debug_info driver option, 303
--no_fused_madd driver option, 87
--no_gen_entry_exit_history driver option, 294
--no_globalcheck_qualifiers driver option, 250
--no_gnu_asm driver option, 311
--no_gsize driver option, 245
--no_guiding_decls driver option, 218
--no_half_precision_type driver option, 88
--no_ignore_debug_references driver option, 238
--no_implicit_extern_c_type_conversion driver option, 313
--no_implicit_float_reg driver option, 87
--no_implicit_typename driver option, 219
--no_indirect_function_call_checks driver option, 96
--no_individual_attribute_data_sections driver option, 99
--no_individual_attribute_function_sections driver option, 101
--no_individual_data_sections driver option, 98
--no_individual_data_sections_for_nameless_vars driver option, 100
--no_individual_function_sections driver option, 100
--no_individual_linkonce_sections driver option, 102
--no_individual_pragma_data_sections driver option, 98
--no_individual_pragma_function_sections driver option, 101
--no_individual_section_name_extra_dot driver option, 102
--no_inline_tiny_functions driver option, 286
--no_inline_trivial driver option, 238
--no_inlining driver option, 308
--no_instantiate_extern_inline driver option, 215
--no_instrument_prologue_epilogue driver option, 96
--no_integrity_posix driver option, 108
--no_irel driver option, 268
--no_japanese_automotive_c driver option, 143
--no_keep_static_symbols driver option, 209
--no_kernel driver option, 107
--no_large_archive driver option, 319
--no_libbsp driver option, 268
--no_link_allow_duplicate_ghs_syms driver option, 235
--no_link_allow_duplicate_linkonce_syms driver option, 235
--no_link_constprop driver option, 237
--no_link_filter driver option, 312
--no_link_reject_duplicate_lib_syms driver option, 234
--no_linker_warnings driver option, 230
--no_list driver option, 224
--no_list_macros driver option, 13
--no_locals_unchanged_by_longjmp driver option, 311
--no_long_long driver option, 206
--no_make_depends driver option, 24
--no_map_eofn_symbols driver option, 245
--no_map_renaming linker option, 427
--no_misalign_access driver option, 208
--no_misra_2012_stricter_R8_7 driver option, 203
--no_misra_2012_stricter_R8_9 driver option, 204
--no_misra_runtime driver option, 148
--no_multiple driver option, 240
--no_namespaces driver option, 213, 688
--no_narrowing_conversions driver option, 223
--no_new_style_casts driver option, 215
--no_old_specializations driver option, 219
--no_p_align linker option, 425
--no_parse_templates driver option, 220
--no_posix_dll driver option, 107
--no_precise_signed_zero driver option, 88
--no_preprocess_assembly_files driver option, 224
--no_preprocess_linker_directive driver option, 229
--no_preprocess_special_assembly_files driver option, 225
--no_quit_after_intex_warnings_intex option, 111
--no_quit_after_warnings driver option, 252
--no_range_based_for driver option, 221
--no_readonly_typeinfo driver option, 217
--no_readonly_virtual_tables driver option, 218
--no_reject_duplicates driver option, 319
--no_remarks driver option, 251
--no_restrict driver option, 214
no_return DoubleCheck property type, 365
no_return_when_non_zero DoubleCheck property type, 364
no_return_when_zero DoubleCheck property type, 364
--no_rtti driver option, 216
--no_scan_source driver option, 299
--no_search_for_dba driver option, 298
--no_short_enum driver option, 206, 595
--no_single_threaded driver option, 105
--no_slash_comment driver option, 146
--no_split_data_sections_by_alignment driver option, 97
--no_stack_check driver option, 93
--no_stack_protector driver option, 94
--no_straightline_spectre_assembler_mitigation driver option, 83

- no_straightline_spectre_mitigation driver option, 82
 - no_strip_application index option, 111
 - no_switch_table driver option, 266
 - no_thread_local_storage driver option, 147
 - no_thread_safe_initializations driver option, 222
 - no_timer_profile driver option, 135
 - no_toolchain_mem_funcs driver option, 81
 - no_trace_includes driver option, 141
 - no_uncompressed_copy linker option, 427
 - no_undefined driver option, 240
 - no_unique_strings driver option, 209
 - no_unsafe_predefines driver option, 305
 - no_user_defined_literals driver option, 222
 - no_using_std driver option, 213
 - no_uvfd driver option, 237
 - no_variadic_floats GNU attribute, 642
 - no_version driver option, 252
 - no_version_info driver option, 252
 - no_vla driver option, 209
 - no-whole-archive driver option, 234
 - no_wrap_diagnostics driver option, 254
 - noasm3g driver option, 226
 - noautoregister driver option, 290
 - nobigswitch driver option, 266
 - nobuild .gpj conditional control statement, 920
 - nochecksum driver option, 246
 - NOCHECKSUM linker section attribute, 445
 - NOCLEAR
 - linker section attribute, 444
 - noconsistentcode driver option, 296
 - nocpp driver option, 312
 - :nodepends Builder option, 275
 - noentry driver option, 229
 - nofarcallpatch driver option, 92
 - __NoFloat__, 740
 - noga driver option, 299
 - .nogen assembler directive, 415
 - noglimits driver option, 296
 - nogtws driver option, 296
 - noidentoutput driver option, 312
 - __noinline, 334
 - noinline GNU attribute, 643
 - nokeepasmfiles driver option, 271
 - nokeepcppfiles driver option, 272
 - nokeepcppfileswithlines driver option, 272
 - nokeepmap driver option, 243
 - nokeeptempfiles driver option, 270
 - .nolist assembler directive, 415
 - NOLOAD linker section attribute, 445
 - nomap driver option, 243
 - nominlib driver option, 316
 - NOMONOTONIC linker section attribute, 445
 - nomonotonic_block_order driver option, 318
 - non-local static objects, 691
 - non-portable features, 843
 - non_shared driver option, 106
 - Non-Standard Output Suffix Builder option, 273
 - nonnull GNU attribute, 643
 - nooverlap driver option, 246
 - nooverload driver option, 290
 - __NOP(); intrinsic function, 768
 - nopasssource driver option, 225
 - noreturn GNU attribute, 643
 - :noSelfdepend Builder option, 275
 - nostartfiles driver option, 239
 - nostddef driver option, 305
 - nostdinc driver option, 306
 - nostdlib driver option, 315, 778
 - nostrip driver option, 228
 - .note assembler directive, 413
 - nothrow GNU attribute, 644
 - nouninstall command line option, 527
 - NULL pointer
 - size of, 849
 - Number of Arenas for Malloc Builder option, 105
 - Number of Callers to Track with Run-Time Memory Checks
 - Builder option, 139
 - numeric constants, 393
- ## O
- o assembler option, 392
 - o driver option, 12, 74, 117
 - O driver option, 119
 - .o file extension, 8, 924
 - o HTML compiler option, 322
 - OB driver option, 124
 - Object File Output Directory Builder option, 113
 - object files
 - ELF format, 874, 888
 - object_dir driver option, 113
 - OBJECTBASE context sensitive variable, customization file, 971
 - OBJECTS context sensitive variable, customization file, 971
 - OBJECTSUFFIX context sensitive variable, customization file, 971
 - obtaining profiling information
 - debugging information, 46
 - Oconstprop driver option, 291
 - Ocse driver option, 291
 - Odebug driver option, 119
 - Oexplodejumps driver option, 289

- Ogeneral driver option, 119
- OI driver option, 121, 131, 331
- Oinline_constant_math driver option, 287
- Ointerproc driver option, 123, 338, 340
- Oip_analysis_only driver option, 123
- Oipaliaslibfuncs driver option, 130
- Oipaliasreads driver option, 129
- Oipaliaswrites driver option, 129
- Oipallocvars driver option, 130
- Oipconstantreturns driver option, 129
- Oipconstglobals driver option, 127
- Oipconstprop driver option, 128
- Oipdeletefunctions driver option, 126
- Oipdeleteglobals driver option, 126
- Oiplimitinlining driver option, 128
- Oiplocalizeglobals driver option, 127
- Oiponesiteinlining driver option, 126
- Oipremoveparams driver option, 128
- Oipremovereturns driver option, 129
- Oipsmallinlining driver option, 128
- OL driver option, 131, 292, 341, 342, 855
- old_section_wildcard_matching linker option, 427
- old_specializations driver option, 219
- Olimit=peephole driver option, 123
- Olimit=pipeline driver option, 123
- Olink driver option, 122
- OM driver option, 292, 598, 854, 855
- Omax driver option, 124
- Omaxdebug driver option, 119
- Omemclr driver option, 288
- Omemfuncs driver option, 287
- Ominmax driver option, 292
- Omoredebug driver option, 119
- only_data_absolute linker option, 427
- __ONLY_STANDARD_KEYWORDS_IN_C, 742
- Ono_inline_constant_math driver option, 287
- Onobig driver option, 124
- Onoconstprop driver option, 291
- Onocse driver option, 291
- Onoexplodejumps driver option, 289
- Onoinline driver option, 121, 331
- Onoipa driver option, 123, 339
- Onoipaliaslibfuncs driver option, 130
- Onoipaliasreads driver option, 129
- Onoipaliaswrites driver option, 129
- Onoipallocvars driver option, 130
- Onoipconstantreturns driver option, 129
- Onoipconstglobals driver option, 127
- Onoipconstprop driver option, 128
- Onoipdeletefunctions driver option, 126
- Onoipdeleteglobals driver option, 127
- Onoiplimitinlining driver option, 128
- Onoiplocalizeglobals driver option, 127
- Onoiponesiteinlining driver option, 126
- Onoipremoveparams driver option, 128
- Onoipremovereturns driver option, 129
- Onoipsmallinlining driver option, 128
- Onolink driver option, 122
- Onoloop driver option, 292, 341
- Onomax driver option, 124
- Onomemclr driver option, 288
- Onomemfuncs driver option, 287
- Onomemory driver option, 292
- Onominmax driver option, 292
- Onone driver option, 120
- Onopeephole driver option, 293
- Onopipeline driver option, 293
- Onoprintfuncs driver option, 288
- Onostrfuncs driver option, 288
- Onotailrecursion driver option, 291
- Onounroll driver option, 289, 345
- Onounrollmore driver option, 124
- Onovector driver option, 121
- Opeep driver option, 293
- operating system dependencies, 850
- operator field, 395
- operator
 - delete[], 745
 - new[], 745
- operators
 - integral expression, 397
 - precedence, 398
 - type combinations and, 401
- Opipeline driver option, 293
- Oprintfuncs driver option, 289
- optgroup=option_group .gpj conditional control statement, 921
- Optimization Strategy Builder option, 119, 124, 286, 761
- optimizations
 - automatic inlining, 328
 - automatic two-pass inlining, 331
 - automatic vector optimization, 346
 - Builder and driver options for, 120, 121, 124, 130, 131
 - C/C++ minimum/maximum, 354
 - code factoring, 460
 - common subexpression elimination, 352
 - constant propagation, 353
 - dead code elimination, 355
 - general, 326, 350
 - inlining, 326
 - inlining advantages, 327
 - inlining specific functions, 331

- loop, 340
 - loop invariant analysis, 343
 - loop merging, 344
 - loop motion , 343
 - loop unrolling, 344
 - manual inlining, 329
 - manual vectorization, 346
 - memory, 354
 - partial redundancy elimination optimization, 352
 - peephole, 351
 - pipeline instruction scheduling, 351
 - portability and, 853
 - static address elimination, 355
 - strength reduction, 342
 - tail calls, 353
 - two-pass inlining, 331
 - varying inlining thresholds, 331
 - Optimize All Appropriate Loops Builder option, 292, 341, 855
 - Optimize Specific Functions for Size Builder option, 131
 - Optimize Specific Functions for Speed Builder option, 130
 - optimizing compilers, xxx
 - option driver option, 321
 - OPTION linker directive, 430
 - %option_value(option_name) .gpj macro function, 922
 - optional .gpj conditional control statement, 921
 - options (see driver options)
 - OPTIONS context sensitive variable, customization file, 971
 - Options to Apply to Project Builder option, 283
 - :optionsFile driver option, 284
 - order of evaluation, 852
 - .org assembler directive, 413
 - OS Directory Builder option, 106
 - os_dir driver option, 106
 - Osize driver option, 120
 - Ospeed driver option, 120
 - Ostrfuncs driver option, 288
 - Otailrecursion driver option, 291
 - Ounroll driver option, 289
 - Ounrollmore driver option, 124, 345
 - Output Archives in Large Format Builder option, 318
 - Output File Size Analysis Builder option, 245
 - Output File Type Builder option, 227
 - output file types, 14
 - .ael, 14
 - .bmon.out, 14
 - .d, 14, 24
 - .dba, 15
 - .dbo, 15
 - .dep, 15, 24
 - .dla, 15
 - .dlo, 15
 - .dnm, 15
 - .env, 15
 - .gmon.out, 14
 - .idep, 15
 - .ii, 15
 - .inf, 15
 - .ipf, 15
 - .lcp, 15
 - .lst, 15
 - .map, 15
 - .mem, 16
 - .mon.out, 14
 - .run, 16
 - .ti, 15
 - .time, 16
 - Output Filename Builder option, 74, 116
 - Output Filename for Generic Types Builder option, 275
 - :outputDir Builder option, 117
 - OUTPUTDIR context sensitive variable, customization file, 971
 - :outputDirRelative Builder option, 273
 - OUTPUTFILE context sensitive variable, customization file, 971
 - :outputName Builder option, 275
 - OUTPUTNAMEBASE context sensitive variable, customization file, 971
 - OV driver option, 121, 346
 - overlap driver option, 246
 - overlap_warn driver option, 246
 - overload driver option, 290
 - Owholeprogram driver option, 122, 338, 340
- ## P
- P driver option, 13, 608
 - p HTML compiler option, 323
 - p librarian command, 468
 - p_align ELF program header field, 894
 - p_filesz ELF program header field, 893
 - p_flags ELF program header field, 894, 895
 - p_memsz ELF program header field, 893
 - p_offset ELF program header field, 893
 - p_paddr ELF program header field, 893
 - p_type ELF program header field, 893
 - p_vaddr ELF program header field, 893
 - pack driver option, 207
 - pack_or_align linker expression function, 443
 - pack=n, 743
 - __packed keyword, 30, 34, 598
 - packing, 30 (see structure packing)

- Packing (Maximum Structure Alignment) Builder option, 32, 33, 207, 760
- PAD linker section attribute, 445
- paddr_offset driver option, 317
- PARENTID context sensitive variable, customization file, 971
- Parse Templates in Generic Form Builder option, 219, 220
- parse_templates driver option, 220
- partial redundancy elimination optimization, 352
- pass .gpj conditional control statement, 921
- Pass Through Arguments Builder option, 282
- passsource driver option, 225
- :passThrough Builder option, 282
- patches, installing, 526
- Pattern of Files to Pull Into Project Builder option, 283
- .pdf file extension, 925
- pecoff librarian option, 470
- peephole optimization, 351
- Peephole Optimization Builder option, 293
- pending_instantiations driver option, 221
- Physical Address Offset From Virtual Address Builder option, 317
- Pipeline and Peephole Scope Builder option, 123
- Pipeline Instruction Scheduling Builder option, 293
- pipeline instruction scheduling optimization, 351
- Placement of Zero-Initialized Data Builder option, 92
- plug-in, Eclipse (see MULTI for Eclipse)
- pointer
 - information, 739
- pointer arithmetic, 849
- pointer type
 - size, 747
- POSIX DLL Builder option, 107
- posix_dll driver option, 107
- post processing, 691
- :postexec Builder option, 279
- :postexecSafe Builder option, 280
- :postexecShell Builder option, 280
- :postexecShellSafe Builder option, 281
- #pragma __printf_args, 750
- #pragma __scanf_args, 750
- _Pragma, 581, 761
- #pragma alignfunc (n), 748
- #pragma alignvar (n), 748
- #pragma asm, 749
- #pragma can_instantiate fn, 686, 749
- #pragma directives
 - #pragma ghs section, 62
- #pragma do_not_instantiate fn, 686, 749
- #pragma endasm, 749
- #pragma ghs alias newsym oldsym, 752
- #pragma ghs check=(check-list), 753
- #pragma ghs end_externally_visible, 758
- #pragma ghs endnocoverage, 758
- #pragma ghs endnoinline, 334, 759
- #pragma ghs endnomisra, 759
- #pragma ghs endnostrongfptr, 759
- #pragma ghs endnowarning, 756
- #pragma ghs extra_stack, 754
- #pragma ghs max_instances, 755
- #pragma ghs max_stack, 755
- #pragma ghs nofloat interrupt, 755
- #pragma ghs nowarning, 756
- #pragma ghs optasm, 758
- #pragma ghs Ostring, 757
- #pragma ghs reference sym, 757
- #pragma ghs revertoptions, 757
- #pragma ghs safeasm, 758
- #pragma ghs section, 758
- #pragma ghs start_externally_visible, 758
- #pragma ghs startnocoverage, 758
- #pragma ghs startnoinline, 334, 759
- #pragma ghs startnomisra, 759
- #pragma ghs startnostrongfptr, 759
- #pragma ghs static_call, 759
- #pragma ghs struct_min_alignment(), 760
- #pragma ghs struct_min_alignment(n), 760
- #pragma ghs unroll, 760
- #pragma ghs vectorize[=n], 346
- #pragma ghs ZO, 757
- #pragma ident ``string'', 749
- #pragma inline function-list, 749
- #pragma instantiate fn, 686, 749
- #pragma message "string">, 749
- #pragma message ("string"), 749
- #pragma once, 749
- #pragma pack (), 750
- #pragma pack (n), 30, 32, 750
- #pragma pack (pop {, name} {, n}), 750
- #pragma pack (push {, name} {, n}), 750
- #pragma unknown_control_flow (function-list), 750
- #pragma weak foo, 751
- #pragma weak foo = bar, 752
- pragma_message driver option, 306
- pragma_message_silent driver option, 306
- pragma_message_unknown driver option, 306
- .prc file extension, 925
- precise_signed_zero driver option, 88
- predefined macro names
 - __ARM_NEON__, 738
 - __ARM64__, 738
 - __ARRAY_OPERATORS, 745

__AUTO_TYPE, 745	__gnu_asm, 738
__BASE__, 743	__GNUC__, 736
__BIG_ENDIAN__, 738	__GNUC_GNU_INLINE__, 736
__BOOL, 745	__GNUC_STDC_INLINE__, 736
__c_plusplus, 737	HLF_DECIMAL_DIG, 741
__CHAR_BIT, 738	HLF_DIG, 741
__Char_Is_Signed__, 739, 746	HLF_EPSILON, 741
__Char_Is_Unsigned__, 739, 746	HLF_HAS_SUBNORM, 741
__CHAR_UNSIGNED__, 739	HLF_MANT_DIG, 741
__COFF__, 743	HLF_MAX, 741
__COUNTER__, 746	HLF_MAX_10_EXP, 741
__cplusplus, 735	HLF_MAX_EXP, 741
__DATE__, 735	HLF_MIN, 741
__DOUBLE_HL__, 740	HLF_MIN_10_EXP, 741
__EDG__, 737	HLF_MIN_EXP, 741
__EDG_IMPLICIT_USING_STD, 745	HLF_TRUE_MIN, 741
__EDG_RUNTIME_USES_NAMESPACES, 745	__IeeeFloat__, 740
__ELF__, 744	__INT_BIT, 739
__EMBEDDED_CXX, 737	__Int_Is_32, 746
__EMBEDDED_CXX_HEADERS, 737	__Int_Is_64, 746
__EXCEPTION_HANDLING, 745	__INTEGRITY, 746
__EXCEPTIONS, 745	introduction to, 734
__EXTENDED_EMBEDDED_CXX, 737	__Japanese_Automotive_C__, 736
__EXTENDED_EMBEDDED_CXX_HEADERS, 737	__LANGUAGE_ASM__, 736
__Field_Is_Signed__, 739	__LANGUAGE_C__, 736
__Field_Is_Unsigned__, 739	__LANGUAGE_CXX__, 736, 737
__FILE__, 735	__LINE__, 735
__FULL_DIR__, 743	__LITTLE_ENDIAN__, 738
__FULL_FILE__, 743	__LL_BIT, 746
__FUNCPTR_BIT, 739	__LL_Is_64, 746
__FUNCTION__, 743	__LLONG_BIT, 739
__ghs__, 737	__LONG_BIT, 739
__ghs_alignment, 742	__Long_Is_32, 746
__ghs_asm, 737	__Long_Is_64, 746
__GHS_Inline_Memory_Functions, 744	__MISRA_i, 742
__GHS_Inline_String_Functions, 744	__msw, 746
__ghs_max_pack_value, 742	__NAMESPACES, 745
__GHS_NOCOMMONS__, 742	__NoFloat__, 740
__GHS_Optimize_Debug__, 744	__ONLY_STANDARD_KEYWORDS_IN_C, 742
__GHS_Optimize_General__, 744	__PRETTY_FUNCTION__, 743
__GHS_Optimize_Inline, 744	__PTR_BIT, 739
__GHS_Optimize_Max_Debug__, 744	__Ptr_Is_32, 747
__GHS_Optimize_More_Debug__, 744	__Ptr_Is_64, 747
__GHS_Optimize_None__, 744	__Ptr_Is_Signed__, 739
__GHS_Optimize_Size__, 744	__Ptr_Is_Unsigned__, 739
__GHS_Optimize_Speed__, 744	__RANGE_BASED_FOR, 745
__ghs_packing, 743	__REG_BIT, 739
__GHS_REVISION_DATE, 737	__Reg_Is_32, 747
__GHS_REVISION_VALUE, 737	__Reg_Is_64, 747
__GHS_VERSION_NUMBER, 738	__RTTI, 745
__GlobalRegisters, 742	__SHRT_BIT, 739

- `__SIGNED_CHARS__`, 739
- `__SoftwareDouble__`, 740
- `__SoftwareFloat__`, 740
- `__STANDARD_CXX`, 737
- `__STANDARD_CXX_HEADERS`, 737
- `__STDC__`, 735
- `__STDC_HOSTED__`, 736
- `__STDC_VERSION__`, 735
- `__STRICT_ANSI__`, 736
- `__TIME__`, 736
- `__TIMESTAMP__`, 743
- `__unix_asm`, 738
- `__WCHAR_BIT`, 739
- `__WChar_Is_Int__`, 747
- `__WChar_Is_Long__`, 747
- `__WChar_Is_Short__`, 747
- `__WChar_Is_Signed__`, 740
- `__WChar_Is_Unsigned__`, 740
- `__WCHAR_T`, 745
- `:preexec` Builder option, 277
- `:preexecSafe` Builder option, 277
- `:preexecShell` Builder option, 278
- `:preexecShellSafe` Builder option, 278
- `--preinclude_asm` driver option, 304
- Prepend String to Every Section Name Builder option, 309
- Preprocess Assembly Files Builder option, 224, 390
- Preprocess Linker Directives Files Builder option, 229
- Preprocess Special Assembly Files Builder option, 225
- `-preprocess_assembly_files` driver option, 224, 390
- `--preprocess_linker_directive` driver option, 229
- `--preprocess_linker_directive_full` driver option, 229
- `-preprocess_special_assembly_files` driver option, 225
- preprocessor
 - Builder and driver options for, 140, 141
 - overview, 607
 - saving output from, 607
- `__PRETTY_FUNCTION__`, 743
- `.previous` assembler directive, 413
- primaryTarget Top Project directive, 918
- printf and scanf Argument Type Checking Builder option, 257
- printf function, 781, 782
 - C++ I/O streams, buffering, 781
- printf() function
 - less buffered I/O, 779
- processing input files, using scripts
 - customization files, 961
- processor characteristics, 26
- processor supported, 37
- profiles, coding standard (see coding standard profiles)
- profiling
 - Builder and driver options for, 135
 - with protrans, 567
 - timing, 135
- Profiling - Block Coverage Builder option, 134
- Profiling - Target-Based Timing Builder option, 50, 135
- `-prog2` linker option, 427
- program entry point, 429
- program headers, 893
- Program is Single-Threaded Builder option, 105
- program sections
 - begin, end, and size symbols for, 455
 - `.bss`, 61, 885
 - `.data`, 61, 885
 - ELF indexes, 883
 - ELF program types, 894
 - ELF section attribute flags, 884
 - ELF section types, 883
 - `.fixaddr`, 452, 887
 - `.fixtype`, 452, 887
 - header conditions, 881
 - `.heap`, 452, 887
 - `.linfix`, 887
 - names, 885
 - `.rel`, 886
 - `.rela`, 886
 - `.rodata`, 61, 453, 886
 - `.ROM`, 453
 - `.rominfo`, 887
 - `.sdabase`, 453, 887
 - `.secinfo`, 454, 888
 - `.shstrtab`, 886
 - `.stack`, 455, 888
 - `.strtab`, 886
 - `.symtab`, 886
 - `.syscall`, 455, 830, 888
 - `.text`, 61, 886
- program variables, grouping, 62
- project build configurations, 915
- project files
 - types of, 923
- Project Manager
 - file types, 923
- `--prototype_errors` driver option, 255
- `--prototype_silent` driver option, 255
- `--prototype_warnings` driver option, 255
- prototypes, 29
- protrans utility, 567
- provide linker expression function, 443
- PT_DYNAMIC ELF program type, 894
- PT_HIPROC ELF program type, 895
- PT_INTERP ELF program type, 894

PT_LOAD ELF program type, 894
PT_LOPROC ELF program type, 895
PT_NOTE ELF program type, 894
PT_NULL ELF program type, 894
PT_PHDR ELF program type, 895
PT_SHLIB ELF program type, 894
__PTR_BIT, 739
__Ptr_Is_32, 747
__Ptr_Is_64, 747
__Ptr_Is_Signed__, 739
__Ptr_Is_Unsigned__, 739
pure
 GNU attribute, 644
pure GNU attribute, 644
putenv(), 809

Q

-Q driver option, 13
-Qfinal driver option, 23
-Qn driver option, 229
-Qnormalize driver option, 22
-quiet_auto_large_archive driver option, 319
Quit Building if Intex Warnings are Generated Builder option, 111
Quit Building if Warnings are Generated Builder option, 252
--quit_after_intex_warnings intex option, 111
--quit_after_warnings driver option, 252
quoted string assembler expression, 400
-Qy driver option, 229

R

-r librarian command, 468
r_addend ELF relocation field, 889
r_info ELF relocation field, 889
r_offset ELF relocation field, 889
RAM, placing variables in, 62
RANDOMIZE linker section attribute, 445
Range-Based For Loop Support Builder option, 221
__RANGE_BASED_FOR, 745
--range_based_for driver option, 221
Raw Import Files Builder option, 230
-rawimport driver option, 230
.rc file extension, 925
--readonly_typeinfo driver option, 217
--readonly_virtual_tables driver option, 218
rebuild .gpj conditional control statement, 921
rebuilding
 Green Hills libraries, 825
Record Pathname (Not linked) Builder option, 242
-record_eagle_signal driver option, 295

-record_entry_exit_history driver option, 133
-record_timemachine driver option, 132
Redirect Error Output to File Builder option, 252
Redirect Error Output to Overwriting File Builder option, 320
reentrant functions, in C run-time libraries, 783
Reference Builder option, 282
:reference driver option, 282
__REG_BIT, 739
__Reg_Is_32, 747
__Reg_Is_64, 747
Register Allocation by Coloring Builder option, 289
register usage, 28
registers
 allocator, 344
 I/O, 74
 implied usage of, 853
 size of, 747
 storing global variables, 70
-reject_duplicates driver option, 319
.rel program section, 886
.rela program section, 886
relative position of data objects, 846
-relobj driver option, 227
relocatable assembler expression, 400
Relocatable Module Base Image Builder option, 268
Relocatable Module Builder option, 268
relocatable object files
 creating, 227
 debug information for, 227, 298
 format, 874
 relocation types, 888
relocation directories
 for ELF object files, 888
relocation types, 888
-relprog driver option, 227
--remap option, 42
remapping host paths, 42
Remarks Builder option, 251
--remarks driver option, 251
Rename Section Builder option, 97
repeat block directives, 412
Report an Error When Multiple Libraries on a Link Line Contain a Definition of the Same Symbol Builder option, 234
report viewer
 gsar Options, 366
 launching from MULTI Project Manager, 366
 launching from the command prompt, 366
.rept assembler directive, 413
reserved symbols, 393

- restrict driver option, 214
- restrict keyword, 671
- restrict Keyword Support Builder option, 214
- restrictions
 - memory optimization, 854
- Retain Comments During Preprocessing Builder option, 305
- Retain Symbols for Unused Statics Builder option, 209
- retainlocals linker option, 427
- __REV16() intrinsic function, 766
- __REV32() intrinsic function, 766
- __REV64() intrinsic function, 766
- revision tracking, 419
- __REVSH() intrinsic function, 766
- __RIR() intrinsic function, 763
- .rodata assembler directive, 413
- .rodata data section, 451
- .rodata directive, 453
- .rodata program section, 61, 453, 886
- ROM
 - putting data in, 451
- ROM linker section attribute, 446, 449
- .ROM program section, 453
- ROM_NOCOPY linker section attribute, 446
- .rominfo program section, 887
- RTOS Include Directories Builder option, 307
- RTOS Library Directories Builder option, 115
- rtos_include_directory driver option, 308
- rtos_library_directory driver option, 116
- RTTI, 216
- __RTTI, 745
- rtti driver option, 216
- .run output file type, 16
- run-time
 - copy and clear sections, 463
 - environment, 73
 - error checking, 138, 753
 - libraries, 774
 - memory checking, 139
 - type information
 - predefined macro, 745
- run-time code
 - customizing, 451
 - default, features that depend on, 835
 - ind_call.a64, 830
 - ind_condvar.c, 830
 - ind_cov.c, 830
 - ind_crt1.c, 830
 - ind_dots.a64, 830
 - ind_errn.c, 831
 - ind_except.c, 831
 - ind_exit.c, 833

- ind_gpid.c, 831
- ind_heap.c, 832
- ind_io.c, 833
- ind_iob.c, 833
- ind_lock.c, 831
- ind_manprf.c, 831
- ind_reset.c, 830
- ind_stackcanary.c, ind_stackcanary2.c, 831
- ind_stackcheck.c, 831
- ind_thrd.c, 832
- ind_thrd11.c, 832
- libsys.a, 451, 829
- special program sections, 451
- run-time error checking
 - Builder and driver options for, 138
 - enabling and disabling, 54
 - #pragma, 753
- Run-Time Error Checks Builder option, 51, 135, 753
- run-time memory checking
 - Builder and driver options for, 139
 - #pragma, 753
- Run-Time Memory Checks Builder option, 53, 55, 138
- run-time type information (see RTTI)
- Run-Time Type Information Support Builder option, 216
- RUNDIR context sensitive variable, customization file, 971

S

- S driver option, 11, 14, 391
- .s file extension, 8, 924
- .s files, 607
- S-Record format
 - generating, 228, 532
- Safe Commands to Execute After Associated Command (via Shell) Builder option, 281
- Safe Commands to Execute After Associated Command Builder option, 280
- Safe Commands to Execute Before Associated Command (via Shell) Builder option, 278
- Safe Commands to Execute Before Associated Command Builder option, 277
- saferc driver option, 150, 151, 152, 153, 154, 155, 156, 157, 158
- .sbss assembler directive, 417
- .sbttl assembler directive, 415
- scalar variables, 289, 290
- .scall assembler directive, 420
- Scan source files to augment native debug info Builder option, 299
- scan_source driver option, 299
- __schedule_barrier() intrinsic function, 767

- ul style="list-style-type: none; padding-left: 0;">
- scxx header file directory, 775
- .sdbase program section, 453, 887
- Search for DBA Builder option, 298
- Search Path for .dbo Files Builder option, 298
- search_for_dba driver option, 298
- .secinfo program section, 454, 888
- SECOND_PASS_OPTION context sensitive variable, customization file, 971
- .section assembler directive, 414
- section driver option, 97
- section GNU attribute, 644
- section maps, 434
 - (see also linker directives files (.ld))
- Section Overlap Checking Builder option, 246
- section_prefix driver option, 309
- section_suffix driver option, 310
- sections
 - control directives, 413
 - defining, 434
 - #pragma directive, 62
- SECTIONS linker directive, 434
- :select Builder option, 282
- selectone_optional.gpj conditional control statement, 922
- selectone_required.gpj conditional control statement, 922
- Self-Dependency Builder option, 275
- sentinel GNU attribute, 645
- sequence related undefined behavior checking, 596
- .set assembler directive, 397, 417
- Set Linker Message to Error Builder option, 261
- Set Linker Message to Silent Builder option, 261
- Set Linker Message to Warning Builder option, 261
- Set Maximum DBO Not Found Warnings Builder option, 300
- Set Message to Error Builder option, 260
- Set Message to Remark Builder option, 260
- Set Message to Silent Builder option, 260
- Set Message to Warning Builder option, 260
- setenv(), 809
- setjmp.h
 - functions, 822
 - __SEV(); intrinsic function, 768
- sh_addr ELF section header field, 882
- sh_addralign ELF section header field, 882
- sh_entsize ELF section header field, 882
- sh_flags ELF section header field, 882, 884
- sh_info ELF section header field, 882, 885, 889
- sh_link ELF section header field, 882, 885, 889
- sh_name ELF section header field, 882, 892
- sh_offset ELF section header field, 882
- sh_size ELF section header field, 882, 892
- sh_type ELF section header field, 882, 885
- SHA-1 Builder option, 111
- sha1 intex option, 112
- Shadow Declarations Builder option, 257
- Shared Libraries Builder option, 52, 106
- SHF_ALLOC ELF section attribute flag, 884
- SHF_EXECINSTR ELF section attribute flag, 884
- SHF_GHS_ABS ELF section attribute flag, 884
- SHF_WRITE ELF section attribute flag, 884
- SHFLAGS linker section attribute, 446
- shift operators, 853
- SHN_ABS ELF section index, 883
- SHN_COMMON ELF section index, 883
- SHN_GHS_SMALLCOMMON ELF section index, 883
- SHN_GHS_TINYCOMMON ELF section index, 883
- SHN_GHS_ZERO_COMMON ELF section index, 883
- SHN_UNDEF ELF section index, 883, 885
- short data type
 - size of, 739
 - variable arguments, 605
- short_enum driver option, 27, 206, 604
- Show Header Trace for Diagnostics Builder option, 254
- __SHRT_BIT, 739
- .shstrtab program section, 886
- SHT_NOTE ELF section type, 884
- SHT_NULL ELF section type, 883
- SHT_PROGBITS ELF section type, 883
- SHT_REL ELF section type, 884, 885
- SHT_RELA ELF section type, 884, 885
- SHT_STRTAB ELF section type, 884
- SHT_SYMTAB ELF section type, 884, 885
- .si files, 607
- SIBLINGS context sensitive variable, customization file, 972
- SIBLINGS[type] context sensitive variable, customization file, 972
- signed_chars driver option, 204
- __SIGNED_CHARS__, 739
- signed_fields driver option, 204, 601
- signed_pointer driver option, 205
- Signedness of Bitfields Builder option, 204, 595, 601, 695
- Signedness of Char Type Builder option, 204, 595, 694
- Signedness of Pointers Builder option, 205
- SIMD vectorization, 346
- Simplify C Print Functions Builder option, 288
- single_threaded driver option, 105
- .size assembler directive, 420
- Size of time_t Builder option, 207
- sizeof linker expression function, 443
- slash_comment driver option, 146
- small driver option, 131
- small_memory_model driver option, 81

- `__SoftwareDouble__`, 740
- `__SoftwareFloat__`, 740
- source assembler option, 392
- source code
 - interleaving with assembly code, 225
- Source Directories Relative to This File Builder option, 117
- Source Directories Relative to Top-Level Project Builder option, 273
- source files for customization, 73
- source level debugging
 - problems, 855
- source listing, 392
- Source Listing Generation Builder option, 59, 224, 225
- Source Listing Generation Output Directory Builder option, 224
- Source Root Builder option, 114
- source scanning, 42
- :sourceDir Builder option, 117
- :sourceDirNonRelative Builder option, 273
- sourceroot command, 42
- .space assembler directive, 408
- specifying individual functions to inline, 331
- spl instruction, 859
- split_data_sections_by_alignment driver option, 65, 97
- srec driver option, 110, 228
- st_info ELF symbol table entry, 890
- st_name ELF symbol table entry, 890
- st_other ELF symbol table entry, 890
- st_shndx ELF symbol table entry, 890, 891
- st_size ELF symbol table entry, 890
- st_value ELF symbol table entry, 890, 891
- stack
 - generating frame to support traces, 296
- stack characteristics, 26
- Stack Limit Checking Builder option, 93
- .stack program section, 455, 888
- stack_check driver option, 93
- stack_protector driver option, 94
- stackcheck attribute, 645
- Standard C++, xxx, 669 (see C++ language)
- standard I/O package, 779
- Standard Include Directories Builder option, 306
 - `__STANDARD_CXX`, 737
 - `__STANDARD_CXX_HEADERS`, 737
- standard_vtbl driver option, 217
- Start Address Symbol Builder option, 74, 229, 429
- Start File Directory Builder option, 239
- Start Files Builder option, 239
- start-up module
 - rebuilding, 825
- startfile_dir driver option, 239
- startfiles driver option, 239
- startup code
 - crt0.o, 451, 827
 - default, features that depend on, 835
 - ind_crt0.c, 828
 - ind_mcpy.c, 829
 - ind_mset.c, 829
 - ind_reset.c, 829
 - libstartup.a, 451, 828
- static
 - constructors and destructors, 691, 839, 841
- static address elimination optimization, 355
- static analysis (see DoubleCheck)
- static assertions, 772
- STB_GLOBAL ELF symbol binding, 890
- STB_HIPROC ELF symbol binding, 891
- STB_LOCAL ELF symbol binding, 890
- STB_LOPROC ELF symbol binding, 891
- STB_WEAK ELF symbol binding, 891
- STD driver option, 143, 669
- std driver option, 144, 669
- std namespace, 745
- std_cxx_include_directory driver option, 307
- <stdarg.h>, 605, 848
- `__STDC__`, 735
- `__STDC_HOSTED__`, 736
- `__STDC_VERSION__`, 735
- stderr driver option, 252
- stderr, 839
- stderr_overwrite driver option, 320
- stderr=errfile librarian option, 470
- stdin, 839
- stdinc driver option, 306
- stdio, 779
- stdl driver option, 145, 737
- stdle driver option, 145, 737, 777
- stdlib driver option, 315
- stdlib.h
 - functions, 822
- stdout, 839
- sterror(), C implementation, 629
- Straight-line Spectre Assembler Mitigation Builder option, 83
- Straight-line Spectre Compiler Mitigation Builder option, 82
- Straight-line Spectre Linker Mitigation Check Builder option, 83
- straightline_spectre_assembler_mitigation driver option, 83
- straightline_spectre_mitigation driver option, 82
- strength reduction optimization, 342
- streq .gpj conditional control statement, 920

strftime(), C implementation, 633
 __STRICT_ANSI__, 736
 -strict_overlap_check driver option, 246
 .string assembler directive, 408
 string constants, 394
 string literals, 671
 string tables, 892
 strings not referenced, 892
 -strip driver option, 228
 -strip_application intex option, 111
 -strip_eagle_log driver option, 133
 -strip_eagle_signal driver option, 295
 -strip_entry_exit_history driver option, 133
 -strip_timemachine driver option, 132
 strong_fpnr attribute, 646
 strprefix .gpj conditional control statement, 920
 .strtab program section, 886
 .struct assembler directive, 418
 struct data type, 600, 671
 packing, 743
 portability, 847
 structure packing, 30, 31
 packing all structures, 33
 packing one structure type, 32
 pointing to packed structures, 34
 portability, 34
 warnings, 34
 STT_FILE ELF symbol type, 891
 STT_FUNC ELF symbol type, 891
 STT_HIPROC ELF symbol type, 891
 STT_LOPROC ELF symbol type, 891
 STT_NOTYPE ELF symbol type, 891
 STT_OBJECT ELF symbol type, 891
 STT_SECTION ELF symbol type, 891
 __STXR16() intrinsic function, 764
 __STXR32() intrinsic function, 764
 __STXR64() intrinsic function, 764
 __STXR8() intrinsic function, 764
 .subtitle assembler directive, 415
 Support __noinline Keyword for Templates Builder option, 214
 Support floating point types in scanf Builder option, 314
 Support for C Type Information in Assembly Files Builder option, 226
 Support for Constructors/Destructors Builder option, 215
 Support for Implicit Extern C Type Conversion Builder option, 313
 Support for Implicit Typenames Builder option, 219
 Support for Old-Style Specializations Builder option, 218, 219

Support for Very Large Switch Statements Builder option, 266
 Support Variable Length Arrays Builder option, 209
 Suppress Dependencies Builder option, 275
 --suppress_vtbl driver option, 217
 switch statements
 MISRA C, 726
 Switch Tables Builder option, 266
 --switch_table driver option, 266
 Symbol Blacklist Builder option, 320
 symbol tables, 889
 -symbol_blacklist_file driver option, 321
 symbolic
 debugging directives, 419
 memory-mapped I/O, 74
 symbols
 bindings, 890
 definition directives, 415
 definition of, 455
 types, 891
 values, 891
 Symbols Referenced Externally Builder option, 124
 .symtab program section, 886
 -syntax driver option, 14
 __SYS() intrinsic function, 770
 -sys_include_directory driver option, 307
 __SYS_XZR() intrinsic function, 770
 .syscall program section, 455, 830, 888
 -syslib driver option, 106
 System Include Directories Builder option, 307

T

-T driver option, 231
 -t librarian command, 468
 -T linker option, 427
 tail call optimization, 353
 Tail Calls Builder option, 291
 .target file extension, 924
 Target Processor Builder option, 265
 target-based timing profiling
 debugging information, 49
 -target_dir command line option, 527
 Tektronix hexadecimal format
 generating, 228, 532
 templates
 builder and driver options for, 218, 219, 220
 class, 685
 dependent name processing, 219
 entity, 685
 function, 685

- function inlining, 334
- guiding declarations, 218
- instantiation and namespaces, 688
- instantiation of, 685
- link-once, 685
- #pragma directives, 686
- Temporary Assembly File Directory Builder option, 270
- Temporary Assembly Files Builder option, 270
- Temporary Output Builder option, 270
- Temporary Output Directory Builder option, 269
- Temporary Preprocessor Output Builder option, 271
- Temporary Preprocessor Output Directory Builder option, 271
- .text assembler directive, 414
- .text program section, 61, 886
- third-party tools
 - compilers, 40, 44
 - limitations when using, 43
 - MULTI for Eclipse, 978
- Thread Safe Initializations Builder option, 222
- Thread-Local Storage Builder option, 147
- thread_local_storage driver option, 147
- thread_safe_initializations driver option, 222
- thread-safety
 - customizing libraries, 787
- thread-safety library functions
 - library functions, 783
- .ti output file type, 15
- .time output file type, 16
- __TIME__, 736
- time.h
 - functions, 823
- time32 driver option, 207
- time64 driver option, 207
- timer_profile driver option, 50, 135
- __TIMESTAMP__, 743
- .title assembler directive, 415
- tmp driver option, 269
- tmp=dir librarian option, 470
- tmpfile(), C implementation, 625
- toolchain_mem_funcs driver option, 81
- Top Project directives, for .gpj files, 914
- Top Project headers, .gpj files, 911
- TOPPROJECT context sensitive variable, customization file, 972
- Toyota Motor Corporation, 595
- trace_includes driver option, 23
- trace_includes driver option, 23
- Transition C
 - variable arguments, 605
- translating
 - debugging information, 40
 - automatically, 41
 - limitations to, 43
 - manually, 41
 - source scanning feature, 42
 - transparent_union GNU attribute, 650
 - Treatment of RTTI as const Builder option, 217
 - Treatment of Virtual Tables as 'const' Builder option, 218
 - Trigraphs Builder option, 258
 - two-pass inlining, 331
 - .txt file extension, 925
 - TYP __builtin_speculation_safe_load intrinsic function, 771
 - type combinations, 401
 - type qualifiers, 597
 - treatment in GNU C, 592
 - typographical conventions, xxvii

U

- U driver option, 140
- u driver option, 241
- UA driver option, 242
- unalign_debug_sections linker option, 428
- unaligned loads and stores, handling, 34
- unary, 397
- .undef assembler directive, 412
- Undefine Preprocessor Symbol Builder option, 140
- undefined assembler expression, 400
- undefined behavior, 613
- undefined driver option, 240
- Undefined Preprocessor Symbols Builder option, 258
- Undefined Symbols Builder option, 240
- underscore linker option, 428
- uninit DoubleCheck property type, 365
- union data type, 600, 671
 - portability, 847
- unique_strings driver option, 209
- Uniquely Allocate All Strings Builder option, 208, 694
- __unix_asm, 738
- Unknown Pragma Directives Builder option, 256, 748
- unknown_pragma_errors driver option, 256, 748
- unknown_pragma_silent driver option, 256, 748
- unknown_pragma_warnings driver option, 256
- Unroll Loops More Builder option, 124
- unsafe_predefines driver option, 305, 734
- unsaved DoubleCheck property type, 365
- unsigned_chars driver option, 204, 595
- unsigned_fields driver option, 205, 595, 601
- unsigned_pointer driver option, 205
- unspecified behavior, 613
- unused GNU attribute, 651

- unweak linker option, 428
- Use Smallest Type Possible for Enum Builder option, 27, 206, 595, 604, 695
- used GNU attribute, 651
- User-Defined Literals Builder option, 222
- user_defined_literals driver option, 222
- user_make_depends driver option, 24
- using_std driver option, 213, 745
- utility programs, 474
 - gaddr2line, 475
 - gasmlist, 476
 - gbin2c, 478
 - gbincmp, 486
 - gbuild, 488
 - gcolor, 499
 - gcompare, 502
 - gdump, 505
 - gfile, 511
 - gfunsize, 511
 - ghexfile (see utility programs, gsrec)
 - ghide, 512
 - gmemfile, 513
 - gnm, 520
 - gsize, 531
 - gsrec, 532
 - gstrip, 561
 - gversion, 563
 - gwhat, 564
 - protrans, 567
- uvfd driver option, 237

V

- V assembler option, 392
- v driver option, 16
- V driver option, 252
- v linker option, 428
- V linker option, 428
- values
 - representable range, 844
- <varargs.h>, 605
- variable
 - alignment, 748
- variable arguments, 605
- variables
 - allocation, 855
 - placement of for embedded development, 62
- vectorization, 346
- version number
 - generating from assembler, 392
 - generating from compiler driver, 252

- version_info driver option, 252
- viewing reports
 - DoubleCheck, 366
- vimport linker option, 428
- Virtual Function Definition Builder option, 217
- vla driver option, 209
- volatile keyword, 598

W

- w assembler option,
- w driver option, 251, 614
- w linker option, 428
- Wait for Debug Translation Before Exiting Builder option, 300
- wait_for_dblink driver option, 300
- Warn for all floating point types Builder option, 266
- Warn for types double or long double Builder option, 267
- warn_dbo_not_found_max driver option, 300
- warn_unused_result GNU attribute, 651
- .warning assembler directive, 421
- %warning preprocessor directive, customization file, 974
- Warnings Builder option, 251, 614
- warnings driver option, 251
- WChar Size Builder option, 310
 - __WCHAR_BIT, 739
 - __WChar_Is_Int__, 747
 - __WChar_Is_Long__, 747
 - __WChar_Is_Short__, 747
 - __WChar_Is_Signed__, 740
 - __WChar_Is_Unsigned__, 740
- wchar_s32 driver option, 310
- __WCHAR_T, 745
- wchar_t type
 - predefined macro, 745
 - predefined macros, 740
 - size, 739
- wchar_u16 driver option, 310
- wchar.h
 - functions, 822
- Wconvert64 driver option, 259
- Wdouble driver option, 267
- .weak assembler directive, 419
- weak GNU attribute, 652
- weak symbols
 - defining, 419, 751, 752
 - ELF symbol binding, 891
 - resolving from libraries, 426, 458
 - testing for in linker directives, 442
- __WFE() intrinsic function, 768
- __WFI() intrinsic function, 769

- Wfloat driver option, 266
- Wformat driver option, 257
- When Rejecting Duplicate Symbols in An Archive, Certain Symbols Generated by the C++ Front End in COMMON Sections Are Ignored Builder option, 319
- When Rejecting Duplicate Symbols in An Archive, Symbols in .ghs.linkonce Sections Are Ignored Builder option, 320
- When Rejecting Duplicate Symbols, Allow Certain Symbols Added by the compiler Builder option, 234
- When Rejecting Duplicate Symbols, Allow Symbols in .ghs.linkonce sections Builder option, 235
- whitespace, 396
- whole-archive driver option, 234
- wide character functions, 810
- Wimplicit-int driver option, 257
- Wl driver option, 233
- Wno-convert64 driver option, 259
- Wno-format driver option, 257
- Wno-implicit-int driver option, 257
- Wno-shadow driver option, 257
- Wno-trigraphs driver option, 258
- Wno-undef driver option, 258
- Wnodouble driver option, 267
- Wnofloat driver option, 266
- .word assembler directive, 409
- word size, 844
- Workaround ARM64 Erratum 1183550 (DMB) Builder option, 82
- Workaround ARM64 Erratum 1183550 (NOP) Builder option, 82
- wrap sym linker option, 428
- wrap_diagnostics driver option, 254
- Write Version Info to .comment Section Builder option, 252
- Wshadow driver option, 257
- Wtrigraphs driver option, 258
- Wundef driver option, 258

X

- X driver option, 321
- x librarian command, 468
- X-Switches Builder option, 321
- xref=declare driver option, 302
- xref=full driver option, 302
- xref=global driver option, 302
- xref=none driver option, 302

Y

- Y0, driver option, 285
- Ya driver option, 285

- __YIELD() intrinsic function, 769
- Yl driver option, 286
- YL, driver option, 285

Index of Driver Options

Symbols

#, 16

@filename, 17

A

-A, 242

-act_like=2012.1, 113

-act_like=2012.5, 114

-act_like=2013.1, 114

-act_like=2013.5, 114

-act_like=2014.1, 114

-act_like=2015.1, 114

-act_like=2016.5, 114

-act_like=2017.1, 114

-act_like=2017.5, 114

-act_like=2018.1, 114

-act_like=2018.5, 114

-act_like=2019.1, 114

-act_like=2019.5, 114

-act_like=2020.1, 114

-act_like=2020.5, 114

-act_like=2021.1, 114

-act_like=5.0, 113

-act_like=5.2, 113

-act_like=latest, 114

-align_standalone_objects, 294

-allow_cxx_duplicates, 319

--allow_different_section_types, 317

-allow_linkonce_duplicates, 320

--alternative_tokens, 210

-annotate_call_return, 81

-ANSI, 142, 577, 596

-ansi, 142, 577, 595

-ansi_alias, 146

--any_output_suffix, 273

-archive, 12, 13, 466

-argcheck, 246

--arm, 144, 681

-arm64_errata_1183550, 82

--aslr_seed, 315

-asm, 225, 390

-asm3g, 226

-asmcmd, 226, 390

--asm_errors, 255, 607

-asmfile_dir, 270

--asm_silent, 255, 595, 607

--asm_warnings, 255, 607

--assembler_warnings, 313

:autoInclude, 283

-auto_large_archive, 318

-autoregister, 290

--auto_type, 222

B

-bigendian, 79

-bigswitch, 266

--bool, 213, 745

--brief_diagnostics, 254

-bsp, 36

C

-c, 11, 13

-C, 240, 305

--C++11, 144, 672

--c++11, 144, 672

--C++14, 144, 676

--c++14, 144, 676

--C++17, 144, 678

--c++17, 144, 678

-C11, 142, 585

-c11, 142, 585

-C17, 142, 585

-c17, 142, 585

-C18, 143, 585

-c18, 143, 585

-C99, 142, 583

-c99, 142, 583

-call_shared, 106

--c_and_cpp_functions_are_distinct, 313

-check, 138

-check=all, 136

-check=alloc, 53, 138

-check=assignbound, 136

-check=bounds, 137

-check=memory, 53, 138

-check=nilderef, 137

-check=nomemory, 138

- check=none, 136
- check=switch, 137
- check=watch, 138
- check=zerodivide, 137
- check_straightline_spectre_mitigation, 83
- checksum, 246
- c_include_directory, 306
- codefactor, 237, 460
- coding_standard, 253
- commons, 91
- consistentcode, 296
- coverage=count, 134
- coverage=count64, 134
- coverage=flag, 134
- coverage=none, 134
- cppfile_dir, 271
- cpu, 38
- cpu=cpu, 36, 265

D

- D, 140
- data_delete, 236
- dbg_source_root, 114
- dblink.args, 301
- dbopath, 298
- dbo_trace, 322
- dbo_version, 322
- defer_parse_function_templates, 220
- delete, 235
- dep_name, 219, 670, 688
- deprecated_nonconst_string_literal_conv, 223
- diag_error, 260
- diagnostic_header_trace, 254
- diag_remark, 260
- diag_suppress, 260
- diag_warning, 260
- directive_dir, 314
- disable_ctors_dtors, 215
- disable_noinline, 214
- discard_zero_initializers, 92
- display_diag_number, 261, 756
- dM, 13
- do_not_wait_for_dblink, 300
- dotciscxx, 9
- double_check.config, 263
- double_check.enable, 263
- double_check.ignore, 263
- double_check.level=high, 262, 359
- double_check.level=low, 262, 359
- double_check.level=medium, 262, 359

- double_check.level=none, 262, 359
- double_check.report, 262, 366
- double_check.stop_build=errors, 264, 367
- double_check.stop_build=off, 264, 367
- double_check.stop_build=warnings, 264, 367
- dual_debug|dwarf2, 303
- duplicate_whitelist, 235
- dwarf2dbo, 304
- dyload, 268

E

- e, 144, 679, 737
- E, 13, 608
- e, 229
- ee, 145, 681
- eel, 145, 737, 777
- eele, 145, 737, 777
- el, 145, 737, 777
- ele, 145, 737, 777
- elxr_diag_error, 261
- elxr_diag_suppress, 261
- elxr_diag_warning, 261
- enable_ctors_dtors, 215
- enable_eagle_log, 133
- enable_noinline, 214
- endfile, 239
- entry, 229
- entry_exit_history_inst_threshold, 134
- errmax, 251
- error_basename, 254
- exceptions, 145, 670
- extend_liveness, 297
- external, 125
- external_file, 125
- extractall, 241
- extra_filename, 299

F

- farcallpatch, 92
- fast, 130
- ffunctions, 87
- fgnu89-inline, 309
- fhard, 84
- filetype.suffix, 9
- float_associativity, 86
- float_scanf, 314
- fNaN_cmp_unordered, 86
- fnodouble, 267
- fnofunctions, 87
- fno-gnu89-inline, 309

-fnoNaNcmpunordered, 86
-fnone, 84
-fnonodouble, 267
-follow_section, 317
-force_uvfd, 316
--force_vtbl, 217
-frigor=accurate, 85
-frigor=default, 85
-frigor=fast, 85
-fsoft, 84
-full_breakdots, 297
-full_debug_info, 303
-fused_madd, 87

G

-G, 38, 40, 45, 132, 779
-g, 132
--g++, 144, 682
-ga, 299
-gcc, 143, 589
-gen_entry_exit_history, 294
-gen_timemachine, 297
-ghsmc_core_count, 301
-ghsmc_file, 300
--ghstd=2010, 253
--ghstd=last, 253
--ghstd=none, 253
-glimits, 296
-globalcheck=full, 249
-globalcheck=none, 249
-globalcheck=normal, 249
-globalcheck_qualifiers, 250
-globalreg, 71, 93, 742
-gnu99, 143
--gnu_asm, 311
--gnu_version=30300, 309
--gnu_version=40300, 309
-gs, 132
-gsize, 245
-gtws, 296
--guiding_decls, 218

H

-H, 23, 141
-half_precision_type, 88
-help, 16
-hex, 228

I

-I, 57, 114

-I-, 58
-ident, 311
-identifier_blacklist_file, 321
-identoutput, 312
-ignore_debug_references, 238
--implicit_extern_c_type_conversion, 313
-implicit_float_reg, 87
--implicit_typename, 219
-include, 304
--incorrect_pragma_errors, 256, 748
--incorrect_pragma_silent, 256, 748
--incorrect_pragma_warnings, 256
-indirect_function_call_checks, 96
-individual_attribute_data_sections, 99
-individual_attribute_function_sections, 101
-individual_data_sections, 98
-individual_data_sections_for_nameless_vars, 99
-individual_function_sections, 100
-individual_linkonce_sections, 102
-individual_pragma_data_sections, 98
-individual_pragma_function_sections, 101
-individual_section_name_extra_dot, 102
--inline_tiny_functions, 286
-inline_trivial, 238
--inlining, 308
--inlining_unless_debug, 308
--instantiate_extern_inline, 214
-instrument_prologue_epilogue, 96
--integrity_posix, 108
:integrity_posix_mode=noposix, 108
:integrity_posix_mode=single, 108
:integrity_posix_mode=system, 108
-irel, 268

J

-japanese_automotive_c, 143, 595, 596

K

-kanji, 212
-keepasmfiles, 271
-keepcppfiles, 271
-keepcppfileswithlines, 272
-keepmap, 243
--keep_static_symbols, 209
-keeptempfiles, 270
-kernel, 107
-kernel=kernel_checked, 107
-kernel=kernel_opt, 107
-kernel=kernel_vel, 107
-kernel=kernel_vel_checked, 107

-kernel=kernel_vel_opt, 107

L

-L, 59, 115
-l, 59, 116
-language, 9, 272, 776
-large_archive, 319
-large_memory_model, 82
-libbsp, 268
-link_allow_duplicate_ghs_syms, 234
-link_allow_duplicate_linkonce_syms, 235
--link_constprop, 237
--linker_link, 312
-linker_warnings, 230
--link_filter, 312
--link_output_mode_linksafe, 317
--link_output_mode_reuse, 316
--link_output_mode_safe, 316
--link_output_mode_unlink, 317
-link_reject_duplicate_lib_syms, 234
-list, 59, 224
-list_dir, 224
--list_macros, 13
-littleendian, 79
-lmulti, 45
-lnk, 232
-lnk0, 231
-lnkcmd, 233
-locals_unchanged_by_longjmp, 311
-locatedprogram, 227
--long_long, 206

M

-Ma, 243
:macro, 284
--make_depends, 24
-malloc_num_arenas, 105
-malloc_version=arena, 103
-malloc_version=legacy, 102
-malloc_version=memcheck, 103
-malloc_version=realtime, 103
-malloc_version=smallblock, 103
-Man, 243
-map, 243
-map_eofn_symbols, 245
-maplines, 243
--max_inlining, 308
--max_inlining_unless_debug, 308
-MD, 24
-memcheck_num_callers, 139

-mem_funcs=fast, 79
-mem_funcs=safe, 80
-memory, 228
-memory_access=default, 205
-memory_access=efficient, 205
-memory_access=safe, 205
-merge_archive, 13, 466
-MF, 24
-minlib, 316
-misalign_access, 208
--misra_2004=10.1, 164
--misra_2004=10.2, 164
--misra_2004=10.3, 165
--misra_2004=10.4, 165
--misra_2004=10.5, 165
--misra_2004=10.6, 165
--misra_2004=1.1, 159
--misra_2004=11.1, 165
--misra_2004=11.2, 165
--misra_2004=11.3, 165
--misra_2004=11.4, 165
--misra_2004=11.5, 165
--misra_2004=1.2, 159
--misra_2004=12.1, 166
--misra_2004=12.10, 167
--misra_2004=12.11, 167
--misra_2004=12.12, 167
--misra_2004=12.13, 167
--misra_2004=12.2, 166
--misra_2004=12.3, 166
--misra_2004=12.4, 166
--misra_2004=12.5, 166
--misra_2004=12.6, 166
--misra_2004=12.7, 166
--misra_2004=12.8, 167
--misra_2004=12.9, 167
--misra_2004=1.3, 159
--misra_2004=13.1, 167
--misra_2004=13.2, 167
--misra_2004=13.3, 168
--misra_2004=13.4, 168
--misra_2004=13.5, 168
--misra_2004=13.6, 168
--misra_2004=13.7, 168
--misra_2004=1.4, 159
--misra_2004=14.1, 168
--misra_2004=14.10, 169
--misra_2004=14.2, 168
--misra_2004=14.3, 168
--misra_2004=14.4, 168
--misra_2004=14.5, 168

--misra_2004=14.6, 168	--misra_2004=20.2, 174
--misra_2004=14.7, 169	--misra_2004=20.3, 174
--misra_2004=14.8, 169	--misra_2004=20.4, 174
--misra_2004=14.9, 169	--misra_2004=20.5, 174
--misra_2004=1.5, 159	--misra_2004=20.6, 174
--misra_2004=15.1, 169	--misra_2004=20.7, 174
--misra_2004=15.2, 169	--misra_2004=20.8, 174
--misra_2004=15.3, 169	--misra_2004=20.9, 174
--misra_2004=15.4, 170	--misra_2004=2.1, 160
--misra_2004=15.5, 170	--misra_2004=21.1, 175
--misra_2004=16.1, 170	--misra_2004=2.2, 160
--misra_2004=16.10, 171	--misra_2004=2.3, 160
--misra_2004=16.2, 170	--misra_2004=2.4, 160
--misra_2004=16.3, 170	--misra_2004=3.1, 160
--misra_2004=16.4, 170	--misra_2004=3.2, 160
--misra_2004=16.5, 170	--misra_2004=3.3, 160
--misra_2004=16.6, 170	--misra_2004=3.4, 160
--misra_2004=16.7, 170	--misra_2004=3.5, 160
--misra_2004=16.8, 170	--misra_2004=3.6, 161
--misra_2004=16.9, 171	--misra_2004=4.1, 161
--misra_2004=17.1, 171	--misra_2004=4.2, 161
--misra_2004=17.2, 171	--misra_2004=5.1, 161
--misra_2004=17.3, 171	--misra_2004=5.2, 161
--misra_2004=17.4, 171	--misra_2004=5.3, 161
--misra_2004=17.5, 171	--misra_2004=5.4, 161
--misra_2004=17.6, 171	--misra_2004=5.5, 161
--misra_2004=18.1, 171	--misra_2004=5.6, 162
--misra_2004=18.2, 171	--misra_2004=5.7, 162
--misra_2004=18.3, 171	--misra_2004=6.1, 162
--misra_2004=18.4, 172	--misra_2004=6.2, 162
--misra_2004=19.1, 172	--misra_2004=6.3, 162
--misra_2004=19.10, 173	--misra_2004=6.4, 162
--misra_2004=19.11, 173	--misra_2004=6.5, 162
--misra_2004=19.12, 173	--misra_2004=7.1, 162
--misra_2004=19.13, 173	--misra_2004=8.1, 163
--misra_2004=19.14, 173	--misra_2004=8.10, 163
--misra_2004=19.15, 173	--misra_2004=8.11, 164
--misra_2004=19.16, 173	--misra_2004=8.12, 164
--misra_2004=19.17, 173	--misra_2004=8.2, 163
--misra_2004=19.2, 172	--misra_2004=8.3, 163
--misra_2004=19.3, 172	--misra_2004=8.4, 163
--misra_2004=19.4, 172	--misra_2004=8.5, 163
--misra_2004=19.5, 172	--misra_2004=8.6, 163
--misra_2004=19.6, 172	--misra_2004=8.7, 163
--misra_2004=19.7, 172	--misra_2004=8.8, 163
--misra_2004=19.8, 172	--misra_2004=8.9, 163
--misra_2004=19.9, 172	--misra_2004=9.1, 164
--misra_2004=20.1, 174	--misra_2004=9.2, 164
--misra_2004=20.10, 174	--misra_2004=9.3, 164
--misra_2004=20.11, 175	--misra_2012=D1.1, 201
--misra_2004=20.12, 175	--misra_2012=D2.1, 201

--misra_2012=D3.1, 201	--misra_2012=R15.2, 190
--misra_2012=D4.1, 202	--misra_2012=R15.3, 191
--misra_2012=D4.10, 203	--misra_2012=R15.4, 191
--misra_2012=D4.11, 203	--misra_2012=R15.5, 191
--misra_2012=D4.12, 203	--misra_2012=R15.6, 191
--misra_2012=D4.13, 203	--misra_2012=R15.7, 191
--misra_2012=D4.14, 203	--misra_2012=R16.1, 191
--misra_2012=D4.2, 202	--misra_2012=R16.2, 192
--misra_2012=D4.3, 202	--misra_2012=R16.3, 192
--misra_2012=D4.4, 202	--misra_2012=R16.4, 192
--misra_2012=D4.5, 202	--misra_2012=R16.5, 192
--misra_2012=D4.6, 202	--misra_2012=R16.6, 192
--misra_2012=D4.7, 202	--misra_2012=R16.7, 192
--misra_2012=D4.8, 202	--misra_2012=R17.1, 192
--misra_2012=D4.9, 202	--misra_2012=R17.2, 193
--misra_2012=R10.1, 185	--misra_2012=R17.3, 193
--misra_2012=R10.2, 185	--misra_2012=R17.4, 193
--misra_2012=R10.3, 185	--misra_2012=R17.5, 193
--misra_2012=R10.4, 185	--misra_2012=R17.6, 193
--misra_2012=R10.5, 185	--misra_2012=R17.7, 193
--misra_2012=R10.6, 186	--misra_2012=R17.8, 194
--misra_2012=R10.7, 186	--misra_2012=R18.1, 194
--misra_2012=R10.8, 186	--misra_2012=R18.2, 194
--misra_2012=R1.1, 177	--misra_2012=R18.3, 194
--misra_2012=R11.1, 186	--misra_2012=R18.4, 194
--misra_2012=R11.2, 186	--misra_2012=R18.5, 194
--misra_2012=R11.3, 187	--misra_2012=R18.6, 194
--misra_2012=R11.4, 187	--misra_2012=R18.7, 195
--misra_2012=R11.5, 187	--misra_2012=R18.8, 195
--misra_2012=R11.6, 187	--misra_2012=R19.1, 195
--misra_2012=R11.7, 187	--misra_2012=R19.2, 195
--misra_2012=R11.8, 187	--misra_2012=R20.1, 195
--misra_2012=R11.9, 187	--misra_2012=R20.10, 196
--misra_2012=R1.2, 177	--misra_2012=R20.11, 197
--misra_2012=R12.1, 188	--misra_2012=R20.12, 197
--misra_2012=R12.2, 188	--misra_2012=R20.13, 197
--misra_2012=R12.3, 188	--misra_2012=R20.14, 197
--misra_2012=R12.4, 188	--misra_2012=R20.2, 195
--misra_2012=R12.5, 188	--misra_2012=R20.3, 196
--misra_2012=R1.3, 177	--misra_2012=R20.4, 196
--misra_2012=R13.1, 188	--misra_2012=R20.5, 196
--misra_2012=R13.2, 189	--misra_2012=R20.6, 196
--misra_2012=R13.3, 189	--misra_2012=R20.7, 196
--misra_2012=R13.4, 189	--misra_2012=R20.8, 196
--misra_2012=R13.5, 189	--misra_2012=R20.9, 196
--misra_2012=R13.6, 189	--misra_2012=R2.1, 178
--misra_2012=R14.1, 190	--misra_2012=R21.1, 197
--misra_2012=R14.2, 190	--misra_2012=R21.10, 198
--misra_2012=R14.3, 190	--misra_2012=R21.11, 198
--misra_2012=R14.4, 190	--misra_2012=R21.12, 198
--misra_2012=R15.1, 190	--misra_2012=R21.13, 199

--misra_2012=R21.14, 199
--misra_2012=R21.15, 199
--misra_2012=R21.16, 199
--misra_2012=R21.17, 199
--misra_2012=R21.18, 199
--misra_2012=R21.19, 199
--misra_2012=R21.2, 197
--misra_2012=R21.20, 199
--misra_2012=R21.3, 197
--misra_2012=R21.4, 198
--misra_2012=R21.5, 198
--misra_2012=R21.6, 198
--misra_2012=R21.7, 198
--misra_2012=R21.8, 198
--misra_2012=R21.9, 198
--misra_2012=R2.2, 178
--misra_2012=R22.1, 200
--misra_2012=R22.10, 201
--misra_2012=R22.2, 200
--misra_2012=R22.3, 200
--misra_2012=R22.4, 200
--misra_2012=R22.5, 201
--misra_2012=R22.6, 201
--misra_2012=R22.7, 201
--misra_2012=R22.8, 201
--misra_2012=R22.9, 201
--misra_2012=R2.3, 178
--misra_2012=R2.4, 178
--misra_2012=R2.5, 178
--misra_2012=R2.6, 178
--misra_2012=R2.7, 179
--misra_2012=R3.1, 179
--misra_2012=R3.2, 179
--misra_2012=R4.1, 179
--misra_2012=R4.2, 179
--misra_2012=R5.1, 180
--misra_2012=R5.2, 180
--misra_2012=R5.3, 180
--misra_2012=R5.4, 180
--misra_2012=R5.5, 180
--misra_2012=R5.6, 180
--misra_2012=R5.7, 180
--misra_2012=R5.8, 180
--misra_2012=R5.9, 181
--misra_2012=R6.1, 181
--misra_2012=R6.2, 181
--misra_2012=R7.1, 181
--misra_2012=R7.2, 181
--misra_2012=R7.3, 181
--misra_2012=R7.4, 182
--misra_2012=R8.1, 182
--misra_2012=R8.10, 183
--misra_2012=R8.11, 183
--misra_2012=R8.12, 183
--misra_2012=R8.13, 183
--misra_2012=R8.14, 184
--misra_2012=R8.2, 182
--misra_2012=R8.3, 182
--misra_2012=R8.4, 182
--misra_2012=R8.5, 182
--misra_2012=R8.6, 183
--misra_2012=R8.7, 183
--misra_2012=R8.8, 183
--misra_2012=R8.9, 183
--misra_2012=R9.1, 184
--misra_2012=R9.2, 184
--misra_2012=R9.3, 184
--misra_2012=R9.4, 184
--misra_2012=R9.5, 185
--misra_2012_stricter_R8_7, 203
--misra_2012_stricter_R8_9, 204
--misra_adv=error, 148
--misra_adv=silent, 148
--misra_adv=warn, 148
--misra_mand=error, 147
--misra_mand=silent, 147
--misra_mand=warn, 147
--misra_req=error, 148
--misra_req=silent, 148
--misra_req=warn, 148
--misra_runtime, 148
-Ml, 245
-MMD, 24
-Mn, 243
-Mna, 243
-monotonic_block_order, 318
-Ms, 244
-Mu, 244
-multiple, 240
--munch, 312
-mute_eagle_signal, 295
-mute_entry_exit_history, 133
-Mx, 244

N

--namespaces, 213
--narrowing_conversions, 223
--new_style_casts, 215
--no_additional_output, 228
-no_align_standalone_objects, 294
-no_allow_cxx_duplicates, 319

--no_allow_different_section_types, 317
--no_allow_linkonce_duplicates, 320
--no_alternative_tokens, 210
--no_annotate_call_return, 81
--no_ansi_alias, 146
--no_any_output_suffix, 273
--no_arm64_errata_1183550, 82
--noasm3g, 226
--no_assembler_warnings, 314
--noautoregister, 290
--no_auto_type, 222
--nobigswitch, 266
--no_bool, 213
--no_brief_diagnostics, 254
--no_c_and_cpp_functions_are_distinct, 313
--no_check_straightline_spectre_mitigation, 83
--nochecksum, 246
--no_codefactor, 237
--no_comments, 305
--no_commons, 62, 65, 67, 91
--noconsistentcode, 296
--nocpp, 312
--no_data_delete, 236
--no_debug, 132
--no_defer_parse_function_templates, 220
--no_delete, 235
--no_dep_name, 219
--no_deprecated_nonconst_string_literal_conv, 223
--no_diagnostic_header_trace, 255
--no_discard_zero_initializers, 92
--no_display_diag_number, 261
--no_dual_debug, 303
--no_dwarf2dbo, 304
--noentry, 229
--no_error_basename, 254
--no_exceptions, 146
--no_extend_liveness, 297
--nofarcallpatch, 92
--no_float_associativity, 86
--no_float_scanf, 314
--no_full_breakdots, 297
--no_full_debug_info, 303
--no_fused_madd, 87
--noga, 299
--no_gen_entry_exit_history, 294
--no_gen_timemachine, 297
--noglimits, 296
--no_globalcheck_qualifiers, 250
--no_gnu_asm, 311
--no_gsize, 245
--nogtws, 296
--no_guiding_decls, 218
--no_half_precision_type, 88
--noindentoutput, 312
--no_ignore_debug_references, 238
--no_implicit_extern_c_type_conversion, 313
--no_implicit_float_reg, 87
--no_implicit_typename, 219
--no_indirect_function_call_checks, 96
--no_individual_attribute_data_sections, 99
--no_individual_attribute_function_sections, 101
--no_individual_data_sections, 98
--no_individual_data_sections_for_nameless_vars, 100
--no_individual_function_sections, 100
--no_individual_linkonce_sections, 102
--no_individual_pragma_data_sections, 98
--no_individual_pragma_function_sections, 101
--no_individual_section_name_extra_dot, 102
--no_inline_tiny_functions, 286
--no_inline_trivial, 238
--no_inlining, 308
--no_instantiate_extern_inline, 215
--no_instrument_prologue_epilogue, 96
--no_integrity_posix, 108
--no_irel, 268
--no_japanese_automotive_c, 143
--nokeepasmfiles, 271
--nokeepcppfiles, 272
--nokeepcppfileswithlines, 272
--nokeepmap, 243
--no_keep_static_symbols, 209
--nokeeptempfiles, 270
--no_kernel, 107
--no_large_archive, 319
--no_libbsp, 268
--no_link_allow_duplicate_ghs_syms, 235
--no_link_allow_duplicate_linkonce_syms, 235
--no_link_constprop, 237
--no_linker_warnings, 230
--no_link_filter, 312
--no_link_reject_duplicate_lib_syms, 234
--no_list, 224
--no_list_macros, 13
--no_locals_unchanged_by_longjmp, 311
--no_long_long, 206
--no_make_depends, 24
--nomap, 243
--no_map_eofn_symbols, 245
--nominlib, 316
--no_misalign_access, 208
--no_misra_2012_stricter_R8_7, 203
--no_misra_2012_stricter_R8_9, 204

- no_misra_runtime, 148
- nomonotonic_block_order, 318
- no_multiple, 240
- no_namespaces, 213, 688
- no_narrowing_conversions, 223
- no_new_style_casts, 215
- non_shared, 106
- no_old_specializations, 219
- nooverlap, 246
- nooverload, 290
- no_parse_templates, 220
- nopasssource, 225
- no_posix_dll, 107
- no_precise_signed_zero, 88
- no_preprocess_assembly_files, 224
- no_preprocess_linker_directive, 229
- no_preprocess_special_assembly_files, 225
- no_quit_after_warnings, 252
- no_range_based_for, 221
- no_readonly_typeinfo, 217
- no_readonly_virtual_tables, 218
- no_reject_duplicates, 319
- no_remarks, 251
- no_restrict, 214
- no_rtti, 216
- no_scan_source, 299
- no_search_for_dba, 298
- no_short_enum, 206, 595
- no_single_threaded, 105
- no_slash_comment, 146
- no_split_data_sections_by_alignment, 97
- no_stack_check, 93
- no_stack_protector, 94
- nostartfiles, 239
- nostddef, 305
- nostdinc, 306
- nostdlib, 315, 778
- no_straightline_spectre_assembler_mitigation, 83
- no_straightline_spectre_mitigation, 82
- nostrip, 228
- no_switch_table, 266
- no_thread_local_storage, 147
- no_thread_safe_initializations, 222
- no_timer_profile, 135
- no_toolchain_mem_funcs, 81
- no_trace_includes, 141
- no_undefined, 240
- no_unique_strings, 209
- no_unsafe_predefines, 305
- no_user_defined_literals, 222
- no_using_std, 213

- no_uvfd, 237
- no_version, 252
- no_version_info, 252
- no_vla, 209
- no-whole-archive, 234
- no_wrap_diagnostics, 254

O

- o, 12, 74, 117
- O, 119
- OB, 124
- object_dir, 113
- Oconstprop, 291
- Ocse, 291
- Odebug, 119
- Oexplodejumps, 289
- Ogeneral, 119
- OI, 121, 131, 331
- Oinline_constant_math, 287
- Ointerproc, 123, 338, 340
- Oipaliaslibfuncs, 130
- Oipaliasreads, 129
- Oipaliaswrites, 129
- Oipallocvars, 130
- Oip_analysis_only, 123
- Oipconstantreturns, 129
- Oipconstglobals, 127
- Oipconstprop, 128
- Oipdeletefunctions, 126
- Oipdeleteglobals, 126
- Oiplimitinlining, 128
- Oiplocalizeglobals, 127
- Oiponesiteinlining, 126
- Oipremoveparams, 128
- Oipremovereturns, 129
- Oipsmallinlining, 128
- OL, 131, 292, 341, 342, 855
- old_specializations, 219
- Olimit=peephole, 123
- Olimit=pipeline, 123
- Olink, 122
- OM, 292, 598, 854, 855
- Omax, 124
- Omaxdebug, 119
- Omemclr, 288
- Omemfuncs, 287
- Ominmax, 292
- Omoredebug, 119
- Onobig, 124
- Onoconstprop, 291

- Onocse, 291
- Onoexplodejumps, 289
- Onoinline, 121, 331
- Ono_inline_constant_math, 287
- Onoipa, 123, 339
- Onoipaliaslibfuncs, 130
- Onoipaliasreads, 129
- Onoipaliaswrites, 129
- Onoipallocvars, 130
- Onoipconstantreturns, 129
- Onoipconstglobals, 127
- Onoipconstprop, 128
- Onoipdeletefunctions, 126
- Onoipdeleteglobals, 127
- Onoiplimitinlining, 128
- Onoiplocalizeglobals, 127
- Onoiponesiteinlining, 126
- Onoipremoveparams, 128
- Onoipremovereturns, 129
- Onoipsmallinlining, 128
- Onolink, 122
- Onoloop, 292, 341
- Onomax, 124
- Onomemclr, 288
- Onomemfuncs, 287
- Onomemory, 292
- Onominmax, 292
- Onone, 120
- Onopeephole, 293
- Onopipeline, 293
- Onoprintfuncs, 288
- Onostrfuncs, 288
- Onotailrecursion, 291
- Onounroll, 289, 345
- Onounrollmore, 124
- Onovector, 121
- Opeep, 293
- Opipeline, 293
- Oprintfuncs, 289
- option, 321
- :optionsFile, 284
- os_dir, 106
- Osize, 120
- Ospeed, 120
- Ostrfuncs, 288
- Otailrecursion, 291
- Ounroll, 289
- Ounrollmore, 124, 345
- OV, 121, 346
- overlap, 246
- overlap_warn, 246

- overload, 290
- Owholeprogram, 122, 338, 340

P

- P, 13, 608
- pack, 207
- paddr_offset, 317
- parse_templates, 220
- passsource, 225
- pending_instantiations, 221
- posix_dll, 107
- pragma_message, 306
- pragma_message_silent, 306
- pragma_message_unknown, 306
- precise_signed_zero, 88
- preinclude_asm, 304
- preprocess_assembly_files, 224, 390
- preprocess_linker_directive, 229
- preprocess_linker_directive_full, 229
- preprocess_special_assembly_files, 225
- prototype_errors, 255
- prototype_silent, 255
- prototype_warnings, 255

Q

- Q, 13
- Qfinal, 23
- Qn, 229
- Qnormalize, 22
- quiet_auto_large_archive, 319
- quit_after_warnings, 252
- Qy, 229

R

- range_based_for, 221
- rawimport, 230
- readonly_typeinfo, 217
- readonly_virtual_tables, 218
- record_eagle_signal, 295
- record_entry_exit_history, 133
- record_timemachine, 132
- :reference, 282
- reject_duplicates, 319
- relobj, 227
- relprog, 227
- remarks, 251
- restrict, 214
- rtos_include_directory, 308
- rtos_library_directory, 116
- rtti, 216

S

-S, 11, 14, 391
--saferc, 150, 151, 152, 153, 154, 155, 156, 157, 158
--scan_source, 299
--search_for_dba, 298
--section, 97
--section_prefix, 309
--section_suffix, 310
--short_enum, 27, 206, 604
--signed_chars, 204
--signed_fields, 204, 601
--signed_pointer, 205
--single_threaded, 105
--slash_comment, 146
--small, 131
--small_memory_model, 81
--split_data_sections_by_alignment, 65, 97
-srec, 110, 228
-stack_check, 93
-stack_protector, 94
--standard_vtbl, 217
-startfile_dir, 239
-startfiles, 239
--STD, 143, 669
--std, 144, 669
-std_cxx_include_directory, 307
-stderr, 252
-stderr_overwrite, 320
-stdinc, 306
--stdl, 145, 737
--stdle, 145, 737, 777
-stdlib, 315
-straightline_spectre_assembler_mitigation, 83
-straightline_spectre_mitigation, 82
-strict_overlap_check, 246
-strip, 228
-strip_eagle_log, 133
-strip_eagle_signal, 295
-strip_entry_exit_history, 133
-strip_timemachine, 132
--suppress_vtbl, 217
--switch_table, 266
-symbol_blacklist_file, 321
-syntax, 14
-sys_include_directory, 307
-syslib, 106

T

-T, 231
--thread_local_storage, 147

--thread_safe_initializations, 222
-time32, 207
-time64, 207
-timer_profile, 50, 135
-tmp, 269
-toolchain_mem_funcs, 81
-trace_includes, 23

U

-U, 140
-u, 241
-UA, 242
-undefined, 240
--unique_strings, 209
--unknown_pragma_errors, 256, 748
--unknown_pragma_silent, 256, 748
--unknown_pragma_warnings, 256
--unsafe_predefines, 305, 734
--unsigned_chars, 204, 595
--unsigned_fields, 205, 595, 601
--unsigned_pointer, 205
--user_defined_literals, 222
--user_make_depends, 24
--using_std, 213, 745
-uvfd, 237

V

-v, 16
-V, 252
-version_info, 252
--vla, 209

W

-w, 251, 614
-wait_for_dblink, 300
-warn_dbo_not_found_max, 300
--warnings, 251
-wchar_s32, 310
-wchar_u16, 310
-Wconvert64, 259
-Wdouble, 267
-Wfloat, 266
-Wformat, 257
--whole-archive, 234
-Wimplicit-int, 257
-Wl, 233
-Wno-convert64, 259
-Wnodouble, 267
-Wnofloat, 266
-Wno-format, 257

- Wno-implicit-int, 257
- Wno-shadow, 257
- Wno-trigraphs, 258
- Wno-undef, 258
- wrap_diagnostics, 254
- Wshadow, 257
- Wtrigraphs, 258
- Wundef, 258

X

- X, 321
- xref=declare, 302
- xref=full, 302
- xref=global, 302
- xref=none, 302

Y

- Y0,, 285
- Ya, 285
- Yl, 286
- YL,, 285